

AutomatedQA Corp.

Getting Started

with

AQtime 2.0



Copyright Notice

AQtime, as described in this online help system, is licensed under the software license agreement distributed with the product. The software may be used or copied only in accordance with the terms of its license.

AQtime is Copyright © 1998-2001 Atanas Stoyanov. ALL RIGHTS RESERVED.

This help file is Copyright © 1999-2001 AutomatedQA Corp. ALL RIGHTS RESERVED.

No part of this help can be reproduced, stored in any retrieval system, copied or modified, transmitted in any form or by any means electronic or mechanical, including photocopying and recording for purposes others than the purchaser's personal use.

All brands and names mentioned in this help system are the properties of their respective owners:

- Microsoft, Windows, Windows NT, Windows 95, Windows 98, Windows 2000, Microsoft Word, Excel, Visual Studio, Visual C++, Visual Basic, Microsoft Visual J++, Internet Information Server and Personal Web Server are registered trademarks or trademarks of Microsoft Corporation.
- Borland, Delphi, C++Builder, JBuilder and Borland C++ are registered trademarks or trademarks of Inprise Corporation.
- CodeSite is a trademark of Raize Software Products.
- Overseer is a freeware product. It is a part of the open source project initiated by Pavel Cisar.
- GCC (or GNU Compiler Collection) is a freeware product. It is a part of the GNU project initiated by Free Software Foundation, Inc. See <http://gcc.gnu.org>.

Table Of Contents

| | |
|---|-----------|
| INTRODUCTION..... | 7 |
| Profiling vs. Testing | 7 |
| What a Profiler Does | 7 |
| What AQttime does | 8 |
| Some elementary questions answered with AQttime | 9 |
| What's New In AQttime 2.0 | 10 |
| System Requirements | 11 |
| Supported Compilers | 12 |
| Installation Notes | 12 |
| Uninstalling AQttime..... | 13 |
| Getting On-Line Help..... | 13 |
| Getting Support..... | 13 |
| Integrating AQttime with Your IDE..... | 13 |
| Automatic Integration..... | 13 |
| Manual Integration | 14 |
| <i>Integrating AQttime with Borland Delphi or C++Builder.....</i> | <i>14</i> |
| <i>Integrating AQttime with Microsoft Visual C++</i> | <i>14</i> |
| Installing Extensions | 15 |
| GETTING STARTED | 16 |
| User Interface - Overview | 16 |
| AQttime Panels..... | 19 |
| AQttime Profilers..... | 20 |
| Preparing a Project for Profiling..... | 22 |
| Compiler Settings for Borland Delphi..... | 22 |
| Compiler Settings for Borland C++Builder | 22 |
| Compiler Settings for Borland C++ | 23 |
| Compiler Settings for Microsoft Visual C++ | 23 |
| <i>Embedded debug information</i> | <i>24</i> |
| <i>Generating debug info as an external PDB file</i> | <i>24</i> |
| <i>Generating debug info as an external DBG file.....</i> | <i>24</i> |
| Compiler Settings for Microsoft Visual Basic | 25 |
| <i>Debug info, generated as an external PDB file.....</i> | <i>25</i> |
| <i>Debug info, included into the executable file</i> | <i>25</i> |
| Compiler Settings for GCC | 26 |
| Profiling a Project..... | 27 |
| Opening a Project | 27 |
| Controlling What To Profile..... | 28 |
| Excluding "System" Files and Functions | 29 |
| Defining Areas To Profile | 30 |
| Checking Elements to Profile | 32 |
| Using Triggers | 33 |
| Setting Up Triggers | 34 |
| Selecting a Profiler | 35 |

| | |
|---|-----------|
| Doing One Profile Run | 35 |
| Analyzing Profiler Results | 36 |
| <i>Organization</i> | 37 |
| <i>Managing results</i> | 38 |
| <i>Transferring results</i> | 38 |
| <i>More usability features</i> | 38 |
| Views Implementation | 38 |
| PANELS REFERENCE | 42 |
| Call Graph Panel | 42 |
| Details Panel | 43 |
| You can arrange the Details panel the same way you can organize other AQtime panels | 43 |
| Function Profiler - Details | 43 |
| Function HitCount - Details | 45 |
| VCL Class Profiler – Details | 47 |
| VCL Reference Count Profiler – Details | 48 |
| Memory and API Resource Check - Details | 50 |
| ATL RefCount Profiler - Details | 50 |
| BDE SQL Profiler - Details | 50 |
| Unused VCL Units Profiler - Details | 50 |
| Disassembly Panel | 50 |
| Editor Panel | 52 |
| Event View Panel | 53 |
| Explorer Panel | 55 |
| Graph Panel | 58 |
| Macro Engine Panel | 59 |
| Macro Engine Plug-In | 59 |
| Macro Engine Panel | 59 |
| About Macros | 60 |
| Macro Recording and Playback | 61 |
| Window and Process Recognition | 61 |
| Monitor Panel | 62 |
| Counter View | 64 |
| Graph View | 64 |
| Histogram View | 65 |
| PEReader Panel | 66 |
| Modules Hierarchy Panel | 68 |
| Function Information Panel | 68 |
| PE Information Panel | 70 |
| <i>Headers</i> | 70 |
| <i>Sections</i> | 71 |
| Report Panel | 71 |
| Setup Panel | 72 |
| Panels How-To | 74 |
| Adding and Removing Columns in AQtime Panels | 74 |
| Column Format | 74 |
| Displaying Results in the Report, Details and Disassembly Panels | 74 |
| Graph Panel Series | 75 |
| Selecting Several Records in a Panel | 75 |
| Working With Results | 75 |
| Comparing and Merging Results | 75 |

| | |
|---|-----------|
| <i>Comparing Results</i> | 76 |
| <i>Merging Results</i> | 77 |
| Exporting Results | 78 |
| Filtering Results | 78 |
| Grouping Results | 79 |
| Inserting Profiling Results into Source Code | 80 |
| Printing Test Results from AQtime | 80 |
| Searching Results | 81 |
| Sorting Results | 81 |
| Views | 82 |
| Panel Options | 82 |
| Call Graph Panel Options | 82 |
| Details Panel Options | 83 |
| Disassembly Panel Options | 83 |
| Editor Panel Options | 84 |
| Event View Panel Options | 84 |
| Explorer Panel Options | 85 |
| Graph Panel Options | 85 |
| Macro Engine Options | 86 |
| Monitor Panel Options | 86 |
| PEReader Options | 87 |
| Report Panel Options | 87 |
| Setup Panel Options | 87 |
| PROFILERS REFERENCE | 89 |
| Static Analysis | 89 |
| Coverage Profilers | 91 |
| Function Coverage Results | 91 |
| Line Coverage (Grouped by Function) Results | 93 |
| Line Coverage (Grouped by File) Results | 94 |
| Hit Count Profilers | 96 |
| Function Profiler | 100 |
| Function Trace Profiler | 103 |
| Description | 103 |
| Displaying Parameters | 106 |
| VCL Profilers | 107 |
| Sampling Profilers | 110 |
| Platform Compliance Analysis | 115 |
| Exception Tracer | 117 |
| Unused VCL Units Profiler | 118 |
| Unused VCL Units Profiler – Description | 118 |
| Unused VCL Units Profiler – Principles of Operation | 120 |
| ATL Reference Count Profiler | 121 |
| ATL Reference Count Profiler – Description | 121 |
| ATL RefCount Profiler – Compiler Settings | 123 |
| BDE SQL Profiler | 124 |
| Memory and API Resource Check Profiler | 125 |
| General Overview | 125 |
| Description of Results | 126 |
| Profiling VC++ Applications | 129 |
| Profiling VCL Applications | 129 |

| | |
|--|------------|
| Settings Dialog | 130 |
| Profiling Memory Management Routines | 133 |
| Checking Bounds of Memory Blocks | 134 |
| Leak Filters Dialog | 135 |
| Predefined Filters | 137 |
| Leak Resources Restriction | 137 |
| Non-Existent Resources in the Report panel | 138 |
| List of Checked Functions | 138 |
| <i>COM functions (ole32.dll and oleaut.32.dll)</i> | 138 |
| <i>GDI functions (gdi32.dll and user32.dll)</i> | 138 |
| <i>Kernel functions (kernel32.dll)</i> | 140 |
| <i>Print Spooler functions (printspool.drv)</i> | 141 |
| <i>Registry functions (advapi32.dll)</i> | 141 |
| <i>System memory management functions</i> | 142 |
| Profilers How-To | 142 |
| Enabling and Disabling Profiling From Application Code | 142 |
| Getting Results During Testing | 144 |
| Calculating Percent Time With Children | 144 |
| Optimizing the Profiling Process | 145 |
| Profiling Recursive Functions | 146 |
| Overloaded Functions | 149 |
| Profiling Inline Functions | 149 |
| Profiling With Microsoft PDB or DBG Debug Info | 151 |
| Profiling Multithreaded Applications | 152 |
| Profiling Dynamic Link Libraries | 153 |
| Profiling ActiveX Controls, OLE Servers and DCOM Servers | 154 |
| Profiling IIS and PWS Applications | 155 |
| Profiling Services | 156 |
| Profiler Options | 157 |
| Coverage Profiler Options | 157 |
| Function Profiler Options | 157 |
| Function HitCount Options | 158 |
| Line HitCount Profiler Options | 158 |
| Sampling Profiler Options | 159 |
| Function Trace Profiler Options | 159 |
| VCL Profiler Options | 159 |
| Platform Compliance Options | 160 |
| Unused VCL Units Profiler Options | 160 |
| ATL RefCount Profiler Options | 161 |
| BDE SQL Profiler Options | 161 |
| Memory and API Resource Check Profiler Options | 161 |
| INDEX | 163 |

Introduction



AQtime by AutomatedQA Corp.

From specification to final delivery, professional developers constantly aim to build applications that are robust, clean-running and clear of hidden bottlenecks, resource wastage and performance limitations. AQtime is the tool that tells you at any moment during development how your application is doing. It is your kit of instruments for checking the application's health.

Profiling vs. Testing

AQtime is an integrated **profiling** toolkit. What is the difference, then, between a profiler and a test tool? A test tool records **what** each part of an application does for other parts, and what the entire application does for the user. A profiler traces **how** the application does what it does. A test tool takes **output** measurements. A profiler takes **health** measurements. Needless to say, AutomatedQA makes an excellent test tool, AQtest. But now that we've shown that profiling and testing are two different things, the rest of this documentation will be concerned only with what AQtime does, profiling.

What a Profiler Does

You can test an application by hand: feed it input, check the output. You can also profile an application by hand: use a stopwatch, use system tools to check resources before and after, etc. Somewhat better, you can insert code to do your profiling: check the system timer at the start and at the end of a code section, check resources likewise, output function calls to a log, etc. In fact, lacking the proper automated profiler, this is what you will do when you are concerned about the "health" of a section of code.

A profiler is called a profiler because it **tracks** one of these measures automatically during a run, and displays results in **comparative** format. For instance, it might time the start and end of any function call, and display results as the percent of total time used by each function. This overall comparison is the "**profile**".

Of course, taking only one kind of profile is not a way of keeping on top of the general health of the application. This is like tracking your health with a balance. A good automatic profiling application will supply many kinds of profilers, and allow you to use any number together or separately, and on varied parts of the application. Better yet, it will let you interactively pin down the crucial information you are looking for, and which may not be where you thought it would be at first.

Not only can a profiler take many kinds of measurements (how often a function is called, time spent in a given unit, events generated, memory leaks, etc.) it can also get these in different ways.

- Some of it is totally non-intrusive: the profiler requests before-and-after information from the operating system.
- Some of it is practically non-intrusive: modern operating systems switch tasks many times per second. On each task switch, which would equally occur without the profiler being present, the profiler can gather some extremely simple information; what this changes to the task switch is unmeasurable; the profiler's only practical intrusion is that it uses some memory and resources. Such a profiler is called a sampling profiler.
- Some of it is minimally intrusive: profiling operations are inserted at many spots, but they are inserted through binary instrumentation. That is, once the executable code is loaded into memory, it is modified to add the needed operations. This is better than source-code instrumentation not only for the reasons explained

below, but because in binary the profiling points can be positioned more precisely. For instance, a short (but often-called) function may spend most of its time in setup and finalization, that is, before the first line of code and after the last. Instrumenting source cannot profile those parts of it, so it yields highly misleading information.

- Some of it is awkwardly intrusive. The processor allows a soft breakpoint operation, which in principle would be the simplest way to call profiler services. So, one variation of binary instrumentation or source code modification (see below) is to insert these "made to order" soft-breakpoint instructions. Under Windows NT or 2000, however, each soft breakpoint implies a context switch, as the profiler is running as a separate process. The result is that most of the runtime will be occupied by these context switches.
- Some of it highly intrusive: the profiler requires modification of the source code, so that your profiling source is never your normal-build source. Since this implies thousands of insertions, it has to be done by an automated source-modification tool. The tool will tempt you into "avoiding" the forking by letting it undo the modifications it did. This is worse still – you can never be sure the "cleaned up" source is identical to what you had in the first place. Some people have sworn off automated profilers because of these intrusions.

As you might expect by now, AQtime never, ever modifies source code. In fact, it always uses the least intrusive method to achieve the requested results. However, since you normally expect results to refer to functions or sections of code, most AQtime profiles require that the application be compiled with debugger information, so that code points can be linked to function or unit names.

Also, AQtime does not use soft breakpoints, with their context switches. The operations added by binary instrumentation are minimal, and run in the same process as the application. It should be noted, however, that binary instrumentation is still instrumentation. The operation may be very quick, but it leaves the processor, with its pipelines and caches, in a somewhat different state than if there had been no instrumentation. Very fine operations will run somewhat differently. This is one reason AQtime also provides a sampling profiler. See *AQtime Profilers*.

A note about results – Many profiles are measures of **relative time**. In the ordinary world, relative time is time relative to total elapsed time, that is, real time. In profiling generally, it's different. You cannot do anything about the elapsed time spent waiting for user input, except to go without the input. You can somewhat easier go without some system calls, but the fact remains that you cannot improve system code. So, a profiler by default compares *profiled* times – the times your own code takes to execute. Relative time is time relative to the time taken by all *profiled* functions.

AQtime of course allows you to get profiles for each function taken alone, or including all the calls it makes to other functions ("child calls"). It also allows you to include or exclude time spent calling the system. And finally it allows you to profile functions not just relative to one another, as is the usual practice for profilers, but relative to real time, the entire elapsed time of the profile run, that is, including input and output calls.

What AQtime does

A hidden but crucial aspect of AQtime is that its architecture is **COM-based**. This means it can be used as a server by any application (the idl is supplied of course). In fact, it is used for some services (e.g. coverage) by AutomatedQA's test automation software, AQtime. More important is that all the parts of AQtime are COM objects. They can be separately plugged in or out. In fact, some of the profilers we will list below are supplied with your installation as separate plug-ins. More will be made available, or are already available on AutomatedQA's Web site, www.automatedqa.com.

Therefore, each of these profilers is a standalone object, each is built and tuned to its one purpose. We are not talking about surface "features" added to the same basic engine, we are talking about separate, professional-grade profilers. The business of making them easy to understand and run, and of integrating their results together in a flexible format, is left to the User Interface, discussed further down.

The current list of profilers is in a separate topic, *AQtime Profilers*. You should read that before proceeding – it's the heart of AQtime.

The User Interface's main tasks are to allow you to:

- Specify the application to profile (project).
- Choose profilers to run on it.
- Filter the profiler results to center on your particular points of concern.
- Display the results.
- Format reports.

Results can be filtered by time, location, etc. They can also be filtered by the **thread** in which the event occurred.

There are many display options. Most results can be shown in one or several graphical formats (e.g. histogram), or in a selection of columns.

One display mode for Function profiler results deserves special attention: the **Call Graph**. All binary-instrumented profilers in AQtime can record the caller for each call of a function. The problem is what to do with the resulting data. The Call Graph is a very easily understood, interactive display that shows each profiled function with basic timings, and arrows from the functions that called it, and to those that it called. Each arrow carries the count of calls recorded.

The most controllable form of result display is the **report**, which is a totally-configurable grid shown in the Report Panel. Besides this onscreen display, the Report Panel can **print** its contents with such options as headers, footers, colors, etc.

Finally, both what you do in AQtime and what you do in your application can be recorded, so that any AQtime session can be repeated exactly on a new build for comparison. See *Macro Recorder*.

At this point, you may like to try your hand on a small project. See *Preparing a Project for AQtime*. But first, if you do not quite see your way around the user interface, begin with *User Interface - Overview*. Once you understand the user interface, here are a few things you might try in your first projects – among others:

Some elementary questions answered with AQtime

- Which units and or files are used by my program?
- Am I linking to my program units and or files that are not really used?
- Which procedures are linked to my program?
- How many procedures are used by my program?
- How many files and or units are used by my program?
- Where in the memory address space are my procedures loaded?
- What is the longest procedure (in procedure source code lines)?
- What is the biggest procedure (in bytes)?
- What is the biggest unit and or file in compiled bytes?
- Which unit contains the greatest and or lowest number of procedures?
- What is the most used and or executed procedure?
- What is the slowest procedure in my program?
- What is the slowest area of my program?
- Which piece of code never gets executed?
- What is the execution flow of my program?

- What is the most used class in my program?
- Do I free all classes allocated in my program?
- What is the binary output produced by the compiler for my source code?

What's New In AQttime 2.0

If you've used **QTime**, AQttime's forebear, (and even if you haven't) this list will help you get up to speed quickly with the new version. .

0. QTime supported **Macro recording** of user actions for playback, so this isn't an honest "What's New" item. And there are several new features that are even more important. But the Recorder feature is easy to overlook because it is normally not visible onscreen. Be sure to see *Macro Recording and Playing Back*.
1. AQttime profiles **Visual Basic** applications. Both of Microsoft' debugger information formats, PDB and DBG, are supported. See *Compiler Settings for Microsoft Visual Basic*.
2. AQttime supports the most common debugger format for **Visual C++** applications, Program Data Base (PDB). See *Compiler Settings for Microsoft Visual C++*.
3. AQttime profiles **GNU C++ (GCC)** applications. See *Compiler Settings for GCC*.
4. AQttime is an **OLE server**. You can control the profiling process from any application, including the application under test. See *Enable/Disable Profiling from Your Application*. This also allows AQttime to provide services to AQttest, AutomatedQA's test automation and management tool.
5. AQttime includes **triggers**. These only apply with the function profilers (all three). Any function can be defined as an on-trigger, and the effect will be that it will turn profiling on, for their thread, on entry and off on exit. Off-triggers will suspend profiling for their thread while they run (including everything they call, directly or indirectly). See *Using Triggers*.
6. AQttime integrates into **development IDEs** – Microsoft Visual C++, Borland Delphi, Borland C++Builder, Borland C++. See *Integrating AQttime into your IDE*.
7. The new **Call Graph** panel displays the flow of execution graphically, in a hierarchical scheme that provides you with an overall view of where the application spends its time, then lets you dig down into the chain of functions calls to any level of detail, always with full timing information. See *Call Graph Panel*.
8. The new **Events** panel targets every kind of event that occurs during the run – UI events, messages, exceptions, module loading and releasing, thread creation and termination, process creation and termination, etc. These are displayed in an expandable tree with information about parameters, results, and of course precise time of occurrence. See *Event View*.
9. The new **Memory and API Resource Profiler** is two monitors wrapped in one. They both work and display in real time, but also can output reports at the end of the run. The Memory Resource profiler tracks memory allocations made by the application and signals leaks and overwrites (e.g. of "just released" memory). The API Call monitor tracks all calls to the Windows API and shows their parameters and return values, especially errors. See *Memory and API Resource Check Profiler*.
10. The new **Platform Compliance Profiler** uses a database of all APIs supported by any Windows platform to tell you what platforms the application will run on without modifications, and every call that would have to be modified to make it run on other platforms. See *Platform Compliance Analysis*.
11. The new **PEReader Profiler** reads the structure of the application's executable and reports what module actually got linked in, what functions it exports, which it imports and what dll's it links statically. See *PEReader*.

12. The Function Profiler now has an option to figure time percentages relative to **total elapsed time**, rather than relative only to total profiled time. The difference is that profiling is off during system calls and calls to other functions you choose not to include. Relative-to-profiled-time tells you how expensive a call is relative to the other calls you are profiling. Relative-to-total-elapsed-time tells you how expensive it is relative to the entire test run. See *% with children relative to real time* option in *Function Profiler Options*.
13. The Function Profiler lets you **drill down** more easily. It has options to order functions in the Call Graph, to show the function body in the Details panel, to show individually only functions that cost more than a given percentage of the total time, etc... See *Function Profiler Options*.
14. The Function Trace profiler can **show calls in the Events view**, can **show the values of parameters** on each call, and can display a **memory dump** from the top of the stack. See *Function Trace Profiler*.
15. The Function Trace profiler can **log its results to file**, to a **display** panel in real time, or to **both**. See *Function Trace Options*.
16. All relevant profilers identify the thread a call occurs in, and can display results by thread. Threads can also receive meaningful names from the application. See *Profiling Multithreaded Applications*.
17. AQttime can display **results before the end of the process** being profiled. This allows profiling processes that cannot be ended without major inconvenience, such as Windows NT services or IIS Applications. See *Getting Results During Testing*.
18. The Explorer panel can now automatically **merge new results with existing ones**. This facilitates gathering statistics over many iterations of a test. See *Auto-Merge* in *Explorer Panel Options*.
19. AQttime now records your jumps among procedures in the Report panel, so that you can easily **retrace your steps** backwards or forward, as in a browser. (The Report panel is AQttime's main output, and you are likely to spend most of your time there.) See *Report Panel*.
20. Test results can be **exported to an XML file**, viewable on any computer that has Internet Explorer 5.0 or better. See *Exporting Data*.

We should be done by now, but we're not. **More improvements** —

- Test results can be added to source code as comments. See *Inserting Results into Source Code*.
- Results from separate runs can be displayed side by side. See *Details Panel*.
- Searching and sorting of features in the Setup Panel.
- Advanced drag-and-drop mode in the Setup Panel.
- Column customization in the Setup Panel.
- Modified Add Procedure, Add Classes and Add Unit dialogs.
- Modified *Docking* mechanism.
- Records can be printed directly from the Report panel.
- Printed reports can be customized for page properties, font style and color, background, etc. See *Printing Results from AQttime*.
- User-defined file types to be recognized by the Editor for syntax highlighting in Visual C++, Visual Basic, Delphi or C++Builder styles.
- More, more...

System Requirements

AQttime 2.0 is designed for 32-bit operating systems such as Microsoft Windows 98, Windows 95, Windows NT 4.0, Windows 2000 or Windows ME.

System requirements:

- Pentium 90 MHz processor or faster (Pentium 166 recommended).
- 64MB RAM or more (96MB or more recommended).
- Microsoft Windows 95, Window 98, Windows 2000, Windows ME or Windows NT 4.0 with Service Pack 3 or later.
- Microsoft Internet Explorer 4.0 or later (It is necessary for Macro Recorder).
- 30 MB hard disk space for optimal performance.
- VGA or higher resolution monitor.
- Mouse or other pointing device.

AQtime consumes a lot of memory to store all the profiling information, so for very big projects, it is recommended to have as much RAM as necessary so Windows does not use the swap file.

Supported Compilers

AQtime is a universal utility. Unlike most existing means, AQtime profiles applications, created by different development tools:

- Borland Delphi
- Borland C++Builder
- Borland C++
- Microsoft Visual C++
- Microsoft Visual Basic
- GNU Compiler.

There are extensions that support Java and NET compilers.

Installation Notes

To install AQtime, execute the installation you have acquired from AutomatedQA and follow the instructions presented by the setup program. By default, the setup program installs AQtime in the <Program Files>\AutomatedQA\AQtime directory. You can specify any other path, if required.

When installing under Windows 98 with Client for Netware Networks running, an InstallShield defect forces the installation to exit or stop with a fatal error message if you press the Browse button on the directory selection dialog.

In this case install AQtime to the default directories, rather than using the Browse button.

For more information on this problem, review the following technical articles on the InstallShield and Microsoft sites:

- Search for article Q192249 at <http://support.microsoft.com/search/default.asp>
- Search for article Q102400 at <http://support.installshield.com>

Uninstalling AQtime

To uninstall AQtime from your computer, open the Control Panel folder and double-click the Add/Remove Programs icon. Select AQtime from the list and then press the Add/Remove button. Follow the on-screen instructions.

Getting On-Line Help

This manual includes only a brief description of AQtime. More information on AQtime is contained in its online help system. To call the on-line help, select **Help | Contents** from AQtime's main menu. To get help for a AQtime dialog, press F1 or click the Help button within the dialog. To get help on the currently selected profiler, press Ctrl-F1 or choose **Help On Selected Profiler** from the Report context menu.

Getting Support

The goal of our support team is to answer and alleviate any problems you might encounter when using AQtime. Feel free to contact us at:

E-Mail:

support@totalqa.com

For a list of commonly asked questions visit us on the **World Wide Web** at:

<http://www.totalqa.com/support>

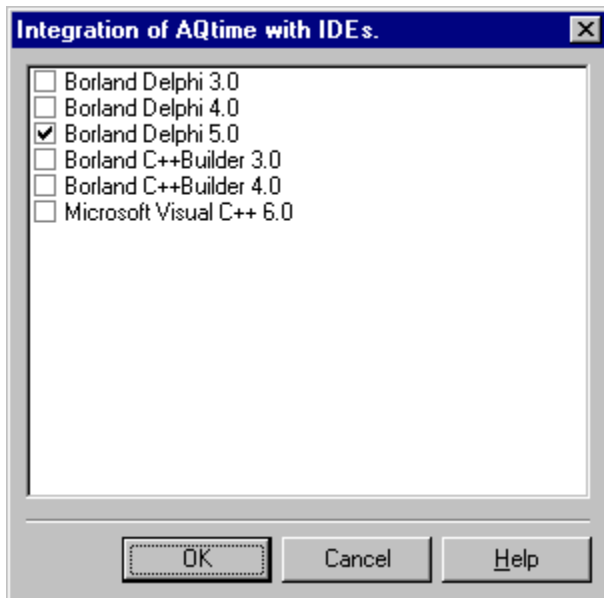
Integrating AQtime with Your IDE

For easy access, AQtime can be installed on the Tools Menu of Borland Delphi, C++Builder or Microsoft Visual C++. Once this is done selecting AQtime from the Tools Menu in the development tool will launch AQtime, compile the current application with debugger information and load it in AQtime with parameters (host application, command line, etc).

You can perform this installation by using the Integration dialog or manually, using the menu system of the development tool. The dialog does the same things for you that you would do manually, but the development tool must be restarted to make the changes effective.

Automatic Integration

The **Integration Dialog** allows automatic integration into any of the supported development tools installed on the machine. To call the dialog, select **File | Integrate AQtime...** from the main menu.



Choose the development tools you want to integrate AQtime with and press **OK**. Restart these development tools to make the changes effective.

Manual Integration


Integrating AQtime with Borland Delphi or C++Builder

To integrate AQtime with Delphi or C++Builder, perform the following steps:

1. Open the development tool.
2. Select **Tools | Configure Tools** from the Borland main menu to open the **Tools Options** dialog.
3. Press **Add** to open the **Tool Properties** dialog.
4. Press **Browse** and locate AQtime.exe on your hard drive.
5. In the **Parameters** box, select the following macros: \$TDW \$EXENAME \$PARAMS. This will instruct your IDE to launch AQtime by compiling the current program with debug information and will then pass the exe file name and command-line parameters to AQtime.
6. Type *AQtime* in the title.
7. Press **OK** to close the dialog box.

Integrating AQtime with Microsoft Visual C++

To integrate AQtime with Visual C++, perform the following steps:

1. Open Visual C++.
2. Select the **Tools | Customize...** from the VC++ main menu. This will call the **Customize** dialog.
3. Choose the **Tools** page in the Customize dialog.
4. Press  **New** at the top of the dialog to create a new item.
5. Press the ellipsis button on the right of the **Command** box and locate AQtime.exe on your hard drive.

6. Enter the following macros in the **Arguments** box: “\$(TargetPath)” \$(TargetArgs). This will instruct your IDE to launch AQtime and then pass the exe name and the command-line parameters to AQtime. **Note:** Do not forget to put \$(TargetPath) in quotes, as shown above, else any path with space characters will not be recognized.
7. Type *AQtime* in the title.
8. Press **Close** to close the dialog box.

Installing Extensions

AQtime is built on an open, COM based, architecture which allows you to write external **plug-ins** for it or install plug-ins from any source. In fact, all AQtime panels are plug-ins, but most are not left as external modules (else the UI could be very confusing).

AQtime ships with several external modules:

| | |
|--|---|
| ATL RefCount Profiler | Tracks the usage of ATL classes and COM objects. |
| BDE SQL Profiler | Measures the execution time of SQL queries and stored procedures, when processed via the Borland Database Engine. |
| Exception Tracer | Traces application exceptions in real time. |
| HexView panel | A sample panel plug-in. Displays the function binary code as hex numbers and ASCII characters. |
| Macro Recorder | Records and plays back macros that simulate user actions on the application. |
| Memory and API Resource Check Profiler | Monitors each call to API functions and tracks memory and resource usage. |
| Monitor | Traces resource usage in real time. |
| PEReader | Analyzes statically linked libraries and imported functions. |
| Unused VCL Units Profiler | Determines which Delphi modules are not actually used by the application. (Source code for this is available.) |

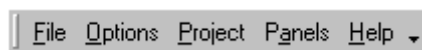
Installing a custom plug-in is extremely simple:

1. Select the **File | Install Extension** menu item. This will bring up the Install Extensions dialog.
2. In the dialog, press the **Add...** button to insert the plug-in into the list of installed plug-ins.
3. Press **OK** to save changes. This is what actually installs the plug-in.

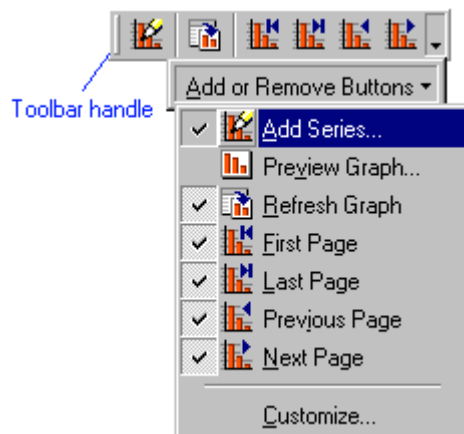
Getting Started

User Interface - Overview

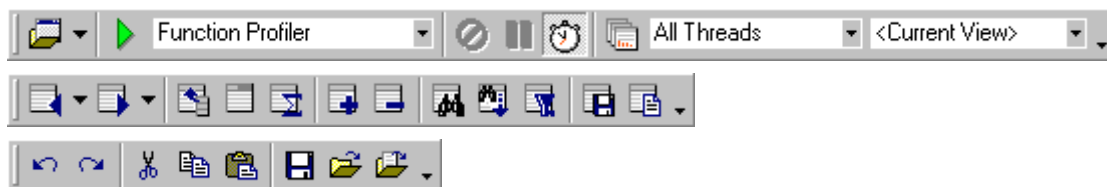
Most of AQtime's screen area is occupied by panels, and these are somewhat special. We'll return to them below. The rest of the interface is occupied by elements you already know from Microsoft Office, or from your development tool –



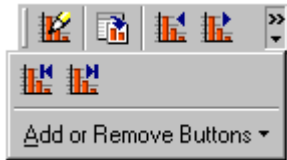
A **main menu** which is actually a toolbar with text items that open submenus. You can drag this menu around using the toolbar handle at the left, and the last item, a simple down arrow ▼, gives access to the *Add or Remove Buttons* submenu, from which you can modify not only the Main Menu, but all the other toolbars. *Add or Remove Buttons* also lets you choose which toolbars will show, and which will stay hidden:



About these options, see *Customizing Toolbars* in on-line help.



A number of other **toolbars**. Standard, Report and Editor are shown above. They all have handles at the left and *Add or Remove Buttons* at the right, as explained for the Main Menu. As you can see above, the Standard toolbar has three text boxes. Each is a dropdown list, not an edit box. It is possible to position a toolbar so that there is not enough horizontal space for it. In that case, you will get the start of the toolbar, ending with a » icon at the right, to show that you can scroll the toolbar to see what's currently hidden:

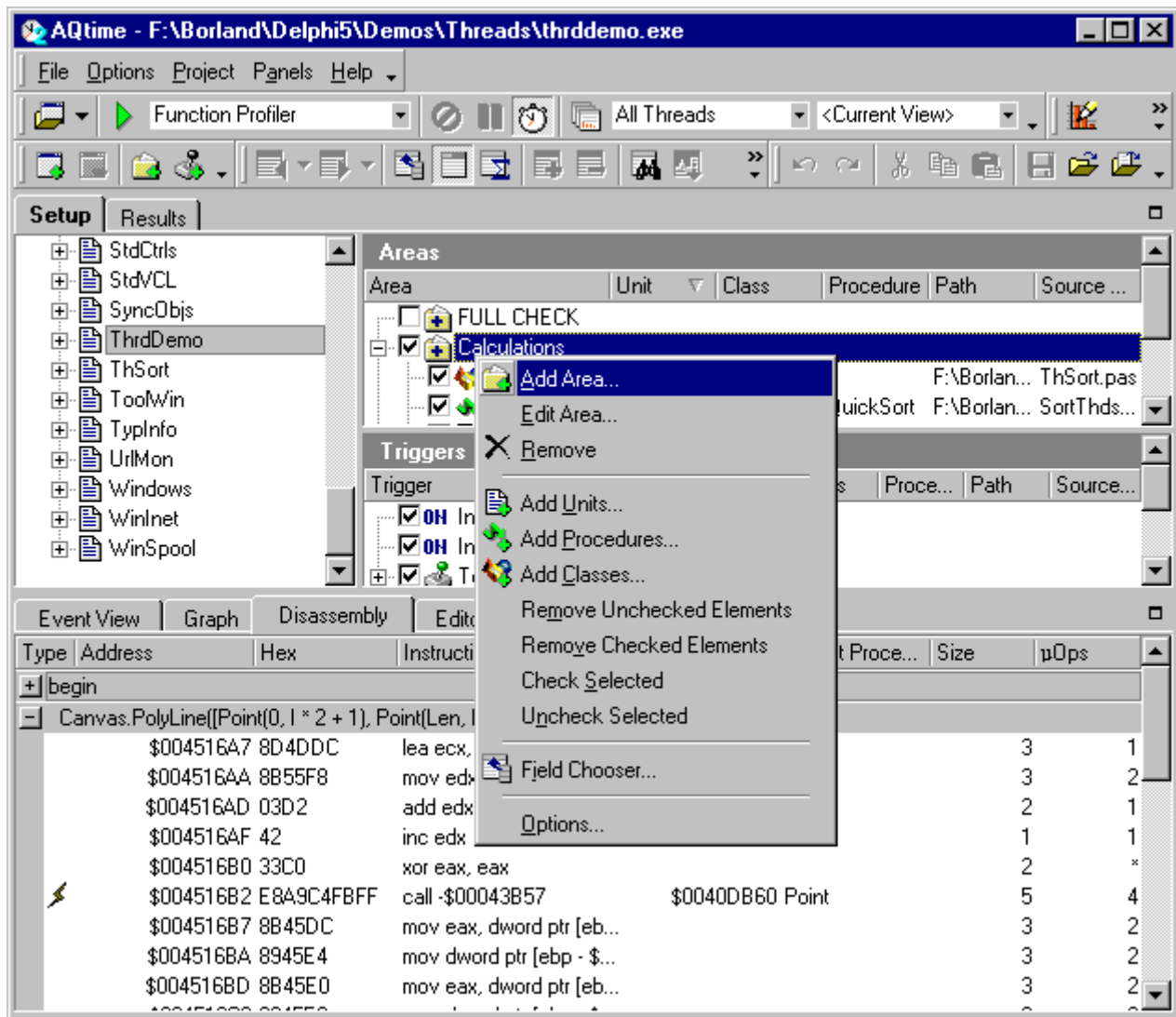


You may well set up your AQtime so that not all toolbars are showing. For information on the purpose and contents of each available toolbar, see:

- Disassembly Toolbar
- Editor Toolbar
- Explorer Toolbar
- Graph Toolbar
- Report Toolbar
- Setup Toolbar
- Standard Toolbar

Context menus that respond to a right-click (or left-click if you are lefthanded) and vary depending on where the mouse pointer is at the moment. A context menu is available everywhere except over blank space around the toolbars and main menu. There are perhaps a score of different context menus for the different spots. Microsoft often uses the term *shortcut menu* for these menus. In theory this is when all options on them are available also from the main menu. AQtime's context menus often hold options that are available only through there.

The general layout is as follows:



Most of the area, as we said, is taken up by **panels**. There are common ways of arranging columns and lines in the grids which most panels display (see Index). Beside this, the general organization of each panel has its own set of options, which you will find on the main menu – **Options | Panel Options**.

But the most important point about handling panels is how they can be moved around – **docking**. Panels are where you do your actual work and get your results in AQtime. Docking is our way of providing you with the most flexible workspace *for the particular task you're interested in*. It means that the entire work area can be reconfigured at will, even beyond what is possible with toolbars (moving, hiding, etc.). This is explained in a separate topic, which you should read: Docking.

Each panel serves a separate purpose in your work with AQtime. What is the purpose of each and how they work together is also explained in a separate topic which you should read: AQtime Panels.

Finally, since you can change so much of the organization of your workspace, it can be quite useful to save it. It's even more useful to have several favorite panel and toolbar configurations at hand that you can load depending on task. For this you have **Options | Save Desktop As...** and **Options | Load Desktop...** on the main menu. Don't hesitate to use them.

But, just in case, there is also **Options | Restore Default Docking**, and an entire **Panels** submenu so you can reach panels after you have hidden them "somewhere". So, don't be afraid to experiment!

AQtime Panels

When using AQtime --

- First you define a profiling project, which will likely involve many profile runs over several days or months.
- Then, for each profile run –
 - you first define what you wish to profile, ...
 - then execute the profile run, ...
 - which generates results when the application exits or when you ask for this through **Project | Get Results**.
- Once you have the new Results --
 - you can browse through them ...
 - or examine them in specific, targeted ways.
- This result set is automatically added to the collected result sets for the project, and, then or later –
 - you can manage the collection, ...
 - examine stored results with all the tools available for new results, ...
 - compare result sets ...
 - or merge them into new sets.

You will spend most of your time in AQtime working in its **panels**. The panels are organized to support the task list above, each panel taking care of one aspect. Of all the tasks above, only the first, defining a project is done outside a panel.

In the following picture, the latest result set from the Explorer panel is being browsed through in the Report panel, with extensive details for the current line in Report displayed below in the Details panel.



There are four major panels. They closely follow the task list above –

| | |
|-------------------|---|
| Setup | This is where you go before a profile run to define what it will profile, and when, once you have selected which profiler to use from the Profiler dropdown list. |
| Event View | This reports messages and events during profiling as they occur. In other words, this is where you track the ongoing profile run. |
| Report | After your results are generated, they are displayed here, and you can browse through them. If the profiled application used threads, the Thread dropdown list lets you choose any single thread to display the results for, or all threads. There are also ways to filter results and to organize the display in the panel. You can save a particular format for the panel and the filters as a View. The View dropdown list lets you apply a View you have saved, or one of the pre-defined ones. |
| Explorer | This is where you manage result sets from the current project, including the latest. Normally, the sets displayed are only those for the currently selected profiler, but you can also choose to have all the collections (one per profiler) presented as a treeview. Any result set can be selected and displayed in the Report panel. You can add a description to each set, and you can store it, retrieve it, merge it with others, compare it to others, save it to a separate file or read it from one. You can organize the entire collection through folders, and you can delete sets from it. |

Five more panels act as extensions to the Report panel, providing various types of information about the currently selected line in Report, or about global results:

| | |
|--------------|---|
| Graph | Shows selected results, or the entire set, in over a dozen different graphical forms. |
|--------------|---|

| | |
|--------------------|---|
| Details | For Function Profiler, VCL Class, VCL Reference Count, Memory and API Resource Check, and for some other plug-in profilers, this provides added detail for the selected line in Report, which it would be impossible to show within reasonable space as added columns in the Report panel itself. |
| Editor | Displays the source code for the line selected in Report (if available), along with optional summary results. |
| Disassembly | Displays the binary code for the line selected in Report, in assembly language, showing either source code with its line-by-line disassembly, or plain disassembly from the binary code in memory. |
| Call Graph | For the Function HitCount and Function Profilers, shows the callers for the function selected in Report, and which functions it called in turn, with statistics for each link. The call hierarchy can be browsed up or down by double-clicking on any parent or child, without returning to Report. |

AQtime keeps information about your browsing in Report and its extension panels, just as a Web browser would. There are  **Back** and  **Forward** on the Report toolbar, and you can use them to move back and forth among a sequence of functions that you are focussing on.

Most of AQtime panels hold tables of data. You can customize them as you wish: Change the column size and place, add and remove columns, sort and group records, etc. See Arranging Columns... in the Index. Exactly what each panel displays is configurable through **Options | Panel Options...** from the main menu. There are separate options for each panel.

Each panel can be undocked and moved to any other location. The **Options | Docking Allowed** menu item specifies whether the docking is active or not. If this option is on, you can undock any panel by double-clicking its header. Then, you can drag this panel to other location, e.g. you can put it to the tabbed page along with another panel. See Docking for complete description of the docking mechanism. If you ever need to bring up a panel quickly, the **Panels** submenu of the main menu has no other purpose – it's your failsafe panel retriever.

AQtime Profilers

This topic is actually an extended section of the AQtime Overview, so it is recommended you read that first. The object of this section is **what do you use the profilers for?** All profilers currently supplied with AQtime are listed, and each has a reference to its own topics for details. This topic provides the what-is-this-for overview.

The point of using any profiler is to know **what your application is doing**. There are simply so many things a Windows application does that to list them in series would be to drown you in detail. Each profiler supplied addresses a limited subset of "What is the application doing?", and uses the best technique to deal with that subset.

To begin, you want to know about how the **source** is used or left unused. For this, Static Analysis will tell you the number of modules, classes or functions, which are the largest (in source lines or in binary) and which are unused. It will also tell you what Windows versions support the API calls in the source (Platform Compliance). See it as an intelligent overview browser for the debug information linked into the executable.

The complement to Static Analysis is the PEReader. (PE is the Windows executable format.) Without source, without needing debug information, the PEReader tells you statically what is actually in the executable – modules that are part of it, functions imported or exported, statically linked dll's, etc., with full details. From a health-check perspective, this can give some important answers very quickly, without profiling any particular run.

All other profiling is done while the application runs. The Sampling profilers (function and line) are the ones described as "practically non-intrusive" in the section above. You can run them on the entire application with no perceptible slowdown. They will tell you how much time the application spends, proportionally, inside each function, class or module, and how much time it spends on some more time-consuming lines, if you use the line-

profiling option. Note that sampling is not extremely accurate, and becomes unreliable for very short functions. Also, using line profiling will cause some slowdown.

The other limitation of the sampling profilers is that they require Windows NT or 2000. Otherwise, because they provide so much information with so little trouble, they're the first run-time profilers to try.

All other profilers use **binary instrumentation** (an invisible runtime process explained in the Overview). It is seldom a good idea to run them on the entire application – they slow down the execution of small, fast functions, and they simply output too much detail. This is why AQtime has **Areas** – an Area is simply a collection of modules, classes, functions on which to turn on profiling. Outside of the specified Area, no binary instrumentation is put in. Areas can also be used with the sampling profilers.

Whether or not you first did an overall check with a sampling profiler, you will normally engage in a **dialogue** with AQtime, going from one profiler to another, tuning areas in and out. AQtime can go to near-insane detail in telling you what your application is doing. In the real world, it should tell you *what you want to know* about what your application is doing. It can't tell what you want to know unless you tell it first. And usually you will only know in the first place by circling around, and then down.

The two main binary-instrumented profilers are Hit Count and Function. The latter times each execution of each function in the selected area. The results will tell you how many times the function was called, how many times it failed to exit normally, how much time it spent on average, at a minimum or at a maximum, the average time including all calls to other functions, the two averages (alone and with calls) as a percent of total execution time, etc.

The HitCount results are the same as the Function results, except that there are no timings, just a count of calls. This holds its own lessons – functions never called, functions called much more often than expected, etc. Normally, you will use the Sampling or Hit Count profilers, or both, to pin down the points of interest before going on to the Function profiler. The Hit Count profiler also has one option the Function profiler does not have – counting "hits" for source **lines**. Very useful too (but slower).

The Coverage profiler is a variation on the Hit Count profiler. It doesn't provide counts, only yes/no values on whether the function or line was executed at all. The reason for having a subset of Function results in Hit Count, then a subset of Hit Count in Coverage, is to avoid human data overflow. No profiler results are any use beyond what *you* actually get out of them. Localized information can swamp out global relationships. Switching between Coverage, Hit Count and Function lets you directly get at what you want to know at the moment. Coverage is actually three profilers: Function, Line (grouped by function) and Line grouped by file (module). As always, profiling by line will slow execution somewhat.

The Function Trace profiler is a different animal. It does not provide statistics, it traces calls in real time. The output is the list of calls made during the profile run, in a collapsible tree display, with call times. Function Trace currently can only operate with Object Pascal source code. However, it is an industry first – it does automatically what can otherwise only be done by introducing hundreds of trace-message lines in the source.

An equally different animal is the Memory and API Resource Check Profiler. Like Function Trace, it works in real time. It follows memory allocations and API calls. For the allocations, it will tell you when (and if) they are released, whether any write goes outside the allocated block and whether any write is to an unallocated block. For the API calls it will tell you input parameters and return values (especially, errors). In report mode, it will also trace back where the call was made from in your application, where that function was called from in turn, etc. Obviously, if you see a profiler as a health-check tool, the Memory and API Resource Profiler is a crucial element.

There are two more profilers that work in a similar way to the memory part of the Memory and API Monitor. The ATL Reference Count Profiler is for calls to Microsoft's Active Template Library. The VCL Profiler is for calls to Borland's Visual Control Library. Both track object allocations by library template (<CFontNotifyImp>) or class (TLabel), and errors raised by the library. These are further extensions to the health-check capabilities available. Besides showing the relationship between your application code and its library, these two profilers can point out one particular problem – memory fragmentation due to the creation of many small objects in succession, and their later destruction.

The Real-Time Resource Monitor does for the ATL and VCL reference counters what the Function Trace profiler does for the function profiler – it follows library object allocations in real time during the application run. Very easy to use and quite instructive at times.

Finally, for Borland applications using SQL through the BDE, there is the BDE SQL Profiler, which logs and times calls to the SQL server.

Preparing a Project for Profiling

AQtime depends on debug information to profile your application. This tells it where functions start and end in memory, and what source-file functions correspond to what executable section in memory. To use AQtime, therefore, your applications must be compiled to include debug information.

When your application is ready for final delivery, remember to compile it without debug info to reduce application size.


Compiler Settings for Borland Delphi

To prepare a Delphi application for AQtime, you must first ensure that it includes TD32 debug info. Follow these steps:

1. To set compiler options, choose **Project | Options** from Delphi's main menu and select the **Compiler** tab.
2. To include symbolic debug information, in the Debugging section of the Compiler page, check **Debug information**.
3. To view variables local to procedures and functions, check **Local Symbols**:
4. Unless you don't want to use the VCL profilers (see point 6), also check **Stack Frames** in the code generation section.
5. To set linker options, now select the **Linker** tab. In the EXE and DLL options group, check **Include TD32 debug info**.
6. If you are profiling an **ActiveX** control, register the “debug” version of this control in the system (See *Profiling ActiveX Controls, OLE Servers and DCOM Servers*).
7. Now, if you do not want to use the VCL profilers, **VCL Class** and **Reference Count**, you're done. Note that the point of profiling with the VCL is not directly performance, as is the case when profiling direct application code. The point of the VCL profilers is rather to track VCL usage.

If you do want your application to support the VCL profilers, you must make sure AQtime has access to the VCL binary code – as follows.

The simplest way to support the VCL profilers is to uncheck **Build with runtime packages**, still on the **Packages** page.


If you wish to keep **Build with runtime packages** (for instance to control exe size), you can still use the VCL profilers. When you include your application in an AQtime project, you will also have to include the VCLnn.BPL file, where nn is the compiler main version number, followed by 0. For instance, with Delphi v. 6.0, you should add the VCL60.BPL file. To add a module to an AQtime project, press  **Add Module** on the **Setup** toolbar or select it from the Setup context menu.

Compiler Settings for Borland C++Builder

To prepare a C++Builder (BCB) application for AQtime, you must first ensure that it includes debug info. Follow these steps:

1. To set compiler options, choose **Project | Options** from BCB's main menu and select the **Compiler** tab.
2. To include symbolic debug information, in the Debugging section of the Compiler page, check Debug information.
Also, to refer this information to source line numbers, check Line Information:
3. Unless you don't want to use the VCL profilers (see point 6), also check Stack Frames in the code generation section.
4. To set linker options, now select the Linker tab. In the Linking options group, check Create Debug Information.
5. If you are profiling an **ActiveX** control, register the "debug" version of this control in the system (See *Profiling ActiveX Controls, OLE Servers and DCOM Servers*).
6. Now, if you do not want to use the **VCL Class**, **VCL Reference Count** or **Memory and API Resource Check** profilers, you're done. Note that the point of these profilers is not performance directly. Their point is rather to track memory and resource usage.

a) If you do want your application to support the **VCL profilers**, you must make sure AQttime has access to the VCL binary code. The simplest way to support the VCL profilers is, still on the Linker page, to check **Use debug libraries** and then, on the **Packages** page, to uncheck **Build with runtime packages**:

If you wish to keep Build with runtime packages enabled (for instance to control exe size), you can still use the VCL profilers. When you include your application in an AQttime project, you will also have to include the VCLnn.BPL file, where nn is the compiler main version number, followed by 5. For instance, with C++Builder v. 5.0, you should add the VCL55.BPL file. To add a module to an AQttime project, press  **Add Module** on the **Setup** toolbar or select it from the **Setup** context menu.

b) To support the **Memory and API Resource Check** profiler for your application, you must check the **Use debug libraries** option on the Linker page. The other option, **Build with runtime packages**, specifies what kind of memory management routines the profiler will analyze. You may check or uncheck it according to what you wish to profile. See *Profiling Memory Management Routines* and *Profiling VCL Applications* in the *Memory and API Resource Check* section.

Compiler Settings for Borland C++

To prepare a Borland C++ application for AQttime, you simply need to ensure that it includes debug info. Three steps will do it:

1. To set compiler options, choose **Project | Options** from Borland C++'s main menu and select the **Compiler** topic.
2. To include symbolic debug information, from the Compiler topic, select the Debugging subtopic. Once there, check **Generate Debug Information**.
3. To set linker options, from Project | Options now select the **Linker** topic. In the Linking options group, check **Create Debug Information**.

Compiler Settings for Microsoft Visual C++

To prepare a Visual C++ application for AQttime, you must ensure that it includes debug info and select the format under which it will be generated. Follow these five steps:

1. Choose a debug configuration, by opening your project in Visual C++, selecting **Build | Set Active Configuration...** and setting a debug configuration as active for your project. Usually it will look like *<Your_Project_Name> - Win32 Debug*.
2. Now, open the **Project Settings** dialog (press Alt-F7 or use **Project | Settings...**) and select the configuration you have set.

3. In the dialog, open the **C/C++** page and make sure that **Debug Info** is set either to “Program Database” or “Program Database for Edit and Continue”.

For more information on these options review Microsoft Visual C++ Help.

4. You now have set your project to generate debug information when compiling. Next, you must ensure that the linker saves it. From **Project | Settings...** select the **Link** page. There, first set **Category** to **General**. Then check **Generate debug info**:
5. The last step is to set **how** the linker will save the debug information. Visual C++ offers three ways; follow the links for details on how to work each:
 - The debug info can be directly embedded into the executable file
 - The Program Data Base that VC++ keeps can be generated as an external PDB file (PDB format)
 - The PDB info can be converted to the older DBG format and generated as an external DBG file

AQtime can work with all the options Visual C++ offers for generating debug information. Profiling applications for which the debug info is in an external file (PDB or DBG) requires a specific Microsoft dll – DbgHelp.dll. It is included in AQtime's installation package.

Once you have set the compiler and linker options correctly, rebuild your application and it will be ready for profiling. If you are profiling an **ActiveX** control, however, you should register the “debug” version of this control in the system (See *Profiling ActiveX Controls, OLE Servers and DCOM Servers*).

Embedded debug information

To have the Visual C++ linker include debug information as part of the executable file, simply tell it not to output it as a separate file.

- a) In the Project Settings dialog set Category to **Customize** then uncheck **Use program database**.
- b) Now, set Category to **General** and uncheck **Enable profiling**.

Generating debug info as an external PDB file

In the **Link** page of the **Project Settings** dialog:

- a) Set Category to **Customize**.
- b) Check **Use program database**.
- c) Enter the PDB file name you want into the **Program database name** edit field.

Generating debug info as an external DBG file

In the **Link** page in the **Project Settings** dialog:

- a) Set Category to **Customize** and uncheck **Use program database**.
- b) Then, select the **General** category and uncheck **Enable profiling**.
- c) Set Category to **Debug** and select either **COFF format** or **Both formats**.

The **COFF** format is necessary for AQtime to read basic debug information, but the information it holds is limited. **"Both"** provides AQtime with **line information** to complement the COFF information. The resulting dbg file will be bigger, but the AQtime line profilers will be able to work. With COFF only, there is no information below the function level, so only function profilers will work.

- d) At this point, you have directed the linker to include debug info in the exe, using the format(s) you specified. When you have rebuilt the application, the last step is to use **Rebase.exe** (a command-line utility that comes with Microsoft Visual C++) to extract the debug info into a DBG file. For instance:


```
rebase -b 0x10000000 -x . your_program_name.exe
```

Usually, Rebase is installed into the <Visual C++ directory>\BIN folder. For more information on the utility, see Microsoft documentation.

Compiler Settings for Microsoft Visual Basic

Microsoft Visual Basic can generate debug info in either of two ways:

- Debug info generated as an external PDB file
- Debug info included in the executable file

AQtime supports both forms – just follow the links. Profiling applications for which the debug info is in an external file (PDB or DBG) requires a specific Microsoft dll – DbgHelp.dll. It is included in AQtime's installation package. See also Profiling Applications with Debug Info in PDB or DBG Files.

Debug info, generated as an external PDB file

Open your project in Microsoft Visual Basic. Then:

- a) Select **Project | Project Properties...** to open the **Project Properties** dialog.
- b) Move to the **Compile** tab and check **Create Symbolic Debug Info**.

When the **Create Symbolic Debug Info** option is active and the LINK environment variable is not set, Visual Basic generates debug info as an external PDB file. It's that simple!

Debug info, included into the executable file

Compiling an executable file with embedded debug info is achieved in exactly the same way as compiling it with external debug info, except that the LINK environment variable must be set. Setting this is a totally separate operation, which you need to carry out outside of Visual Basic before compiling.

So, simply follow the steps in the Debug info, generated as an external PDB file section to set Visual Basic to generate debug info. Here are the additional steps for setting the LINK environment variable so VB integrates the debug info into your executable. The first recipe below will work no matter what your operating system. The other recipes are simpler, but they vary according to operating system:

Windows NT, Windows 95, Windows 98, Windows 2000

1. Open an MS-DOS Prompt. You can use **Start | Programs | Command Prompt**.
2. Move to the folder that contains Microsoft Visual Basic. Do this by typing the disk name, for instance "c:", then Enter, then "cd", a space and the full path – for instance, "cd \visual basic". then Enter
3. Type the following command, followed by Enter:
`set link=/pdb:none`
4. Launch Visual Basic from the MS-DOS command prompt by typing the name of the exe file, for instance "vb6", then Enter.

Windows NT only

1. Open Control Panel.
2. Double-click **System** to open the **System Properties** dialog.
3. Within the dialog, select the Environment tab.

4. In the **Variable** field enter the word `link`
In the **Value** field, enter: `/pdb:none`
5. Press **Set** to add the variable to the environment.
6. Press **Apply** and close the dialog.
7. Start Microsoft Visual Basic in the normal way.

Windows 95/98 Only

1. Make a backup copy of the `c:\autoexec.bat` file, so that later you will be able to undo changes quickly.
2. Open the `autoexec.bat` file with any text editor.
3. Add the following line to the `autoexec.bat` file:
`set link=/pdb:none`
4. Save the `autoexec.bat` file and close the text editor.
5. Reboot your computer.
6. Start Microsoft Visual Basic.

Windows 2000 Only

1. Open Control Panel.
2. Double-click **System** to open the **System Properties** dialog.
 - Within the dialog, select the **Advanced** tab.
 - Click on **Environment Variables...** to open the **Environment Variables** dialog.
 - In this dialog, in the **System Variables** group, press **New...** to open the **New System Variable** dialog.
 - In the **Variable Name** field enter the word `link`.
In the **Variable Value** field, enter: `/pdb:none`
6. Press **OK** and close the dialogs.
7. Start Microsoft Visual Basic.

When you re-compile your application, remember that **Create Symbolic Debug Info** must be checked. . Also, if you wish to profile an ActiveX control, you must register the “debug” version of this control in the system (See *Profiling ActiveX Controls, OLE Servers and DCOM Servers*).

Compiler Settings for GCC

The current AQtime version must get debug information in the **stab** (symbol table) format, so you must use the GDB extension for stab.

To generate debug information in stab format, use either the **-g** or the **-ggdb** compiler option.

-g means "debug information in the format native to the operating system", and stab is OS-native for Cygwin.

-ggdb means "debug information in the format native to the compiler", and again this is stab for Cygwin.

Remember that future versions of gcc may use a different default debug information format. For more information on gcc arguments controlling the creation of debug information, see the *Options for Debugging Your Program or GNU CC* topic of the GCC Compiler Guide.

Profiling a Project

Throughout the AQtime Help system, we will use the generic term ***profiling*** for the use of any of AQtime's test tools. Usually, but not always, a complete profiling operation involves the following steps:


- **Compiling** your application with debug information
- **Opening** the project (application) in AQtime
- Controlling **what to profile**
- Selecting the **profiler** to run
- Doing one **profile run**
- **Analyzing** the results

Opening a Project

Your AQtime **project** is simply your current "work site" in AQtime – application, modules, profiling parameters, etc. Recent results for the project are kept reference, and you can save them permanently also. So the project is also your set of available past results.

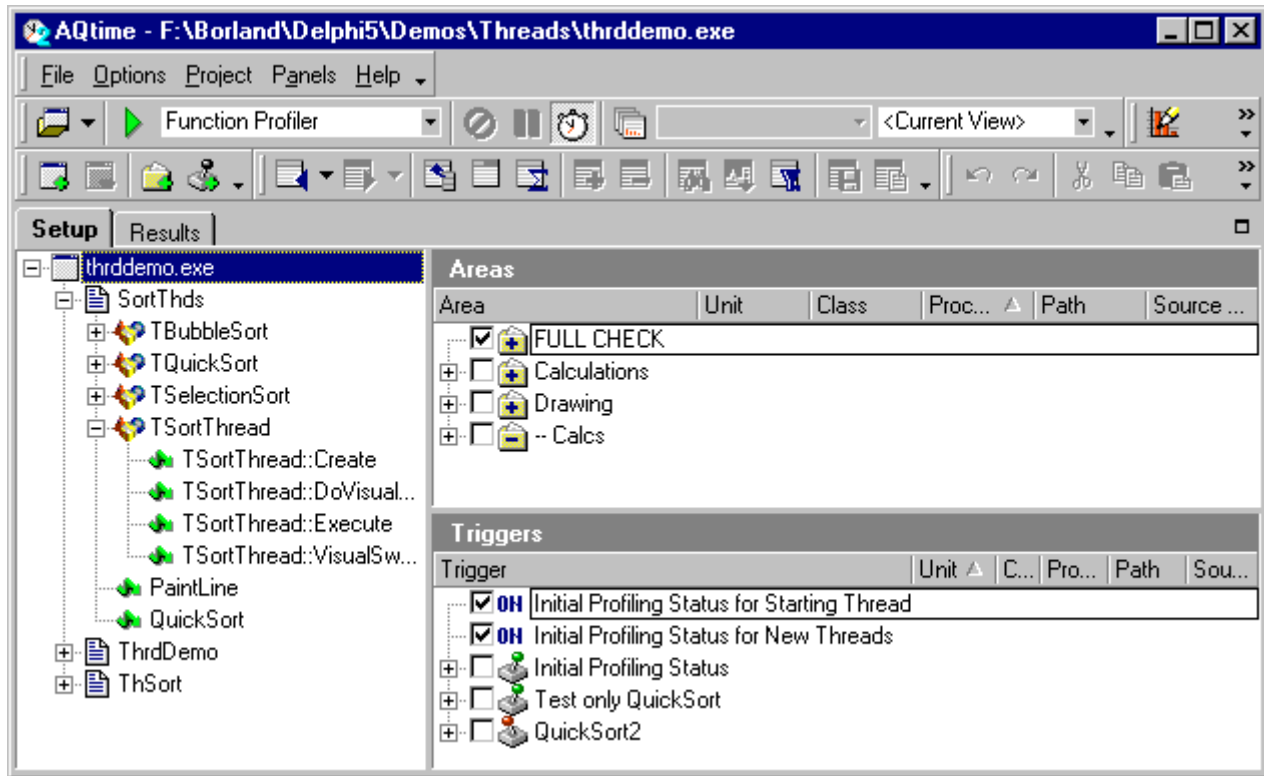
Opening a project means specifying the application you want to profile now.

Before you open a project, make sure you have compiled your application with Debug Information.

Then, just use **File | Open...** on the main menu, or the  Open button on the toolbar, or simply Ctrl-O. Then browse for your application in the standard file-open dialog. To the right of the Open button, there is a down arrow which will open a list of the applications you most recently used for projects in AQtime. This saves browsing.

AQtime profiling can extend to statically or dynamically linked **DLLs**, as well as **ocx**, **bpl** or **dpl** modules. To profile within a dll or other module, set your project to your main exe, then go to the Setup panel and use the context (right-click) menu. You'll find **Add Module...**, which leads to a standard multi-file Open dialog. You can add as many modules, from as many folders, as you wish.

Once your project is open in AQtime, you can see a list of all object modules (units) and their functions in the left-hand pane of the Setup panel:



Controlling What To Profile

Profilers yield a mass of information. The trick in using them is to, as well as you can, ask only for the kind of information you need at the moment, get it, then begin over again, using what you've just learned to refine the question you are asking. In other words, you **dig down** progressively.

So, even more important than **what** information a profiler can provide is **what about** it will provide it. A good profiling tool is one of which you can ask very restricted questions easily, get answers, then re-tune your question. Otherwise, the important information gets lost in the mass of results which, at the present moment, are of no importance or, worse, a distraction. (Also, a few profilers seriously add to execution time, so you don't want to wait while they gather information you won't need.)


A lot of the advantages of AQtime reside in the ease of use, variety and flexibility of the means it provides you with for **controlling** what gets profiled in any given run. All of them work on the **exclusion** principle: If a given means says something will not be profiled, it will not be. If it doesn't say that, or it says the code "will" be profiled, then the code will only actually be profiled if all the other means permit. From the very general to the very local, these means group into four categories:

Means for excluding **code from profiling**. Code can be excluded by source file or by function and, as always, any number of files or functions can be excluded. Once code is excluded in this way, it will never be profiled until you change the settings. This overrides the three other categories. The general name for these means of exclusion is "system files"; it is a bit precise, but it's uncomplicated. See Excluding "System" Files and Functions.

Means for defining **code areas to profile**. *Areas* are a central concept in AQtime. Any number of files, classes or functions can be included in an area, and any number of areas can be checked (or unchecked) for profiling in a given run. Furthermore, each element in a checked area can also be checked or unchecked. Areas are a primary tool for progressively refining what you want profiled. As noted, code correctly checked-in this way only gets profiled if all of the other three ways permit it. But what is *not* checked doesn't get profiled on this run, period. Because sometimes you may want to put an entire class or unit in an area, *except* one or two elements, in addition to the

normal Including areas there are Excluding areas. Since nothing gets profiled in any case if it is not in an Including area, the point of Excluding areas is only this, to provide an easy way of removing some sub-elements from a larger element added to an Including area. See *Defining Areas To Profile* and *Checking Elements to Profile*.

Means for defining **when code will be profiled**. *Triggers* are another central concept in AQtime. They apply only to the three function profilers, and they are the only means of controlling profiling on a **thread** basis. There are on-triggers and off-triggers. An on-trigger is a function that turns profiling on when it begins and turns it off (unless another trigger is running) when it ends. Code that is correctly checked in the Area system, and not excluded as a "system file", will be profiled only when it is called from a trigger function in the same thread, directly or indirectly. On-triggers always get profiled themselves even if outside any profiling Area. Off-triggers are the opposite. While they're running, whatever profiling would be going on in their thread is turned off. If there are no trigger functions, then profiling is always on by default (the application is the trigger). See *Using Triggers and Setting Up Triggers*.

Means for **turning profiling on and off during the run**. There are two such means. The  Enable/Disable Profiling button on the toolbar button can turn profiling off at any time while the application is running. When it is "on" (the default), of course it only enables profiling as restricted according to the three points above. This is a really quick, no-fuss, no-mess way to restrict profiling to a given trouble spot – once you know where it occurs. Its drawback is that you can never repeat the run exactly; for run-to-run comparisons, Triggers are the tool to use. See *Enable/Disable button details*. The other means of doing this is from application code. See *Enabling and Disabling Profiling From Application Code*. For what goes on during a profiling run see *Doing one profile run*.

Excluding "System" Files and Functions

All the means listed in *Controlling What To Profile* are actually means of excluding code from profiling, or of restricting profiling to certain times during the run of the application. The object of this topic is the means of excluding files or functions that you will "never" want to profile (not quote marks), either in any application, or in the current project. In other words, here we're talking about exclusions that are global AQtime settings, or whole-project settings. More-controlled exclusions are better defined through the Areas facility.

The general terms for these overall exclusions are "system files" or "system functions".

But first, note that AQtime will not profile files for which it has no source information. This is a default setting, and can be overridden by modifying the Registry. From that point, your results may vary. Unless you have a serious and overriding motive, leave the default the way it is. In the opposite case, see *Exclude Files with No Source Info* in on0line help.

So, by strong default, the Win API, for instance, is excluded. But you might wish to generally exclude code for which the source is present, but which you don't mean to touch, such as the runtime libraries of your various development tools. These are what we, in approximative fashion, call *system files*.

The first three items on the Options submenu from the main menu deal with this. Item two is System Files and item three is System Functions. Click on the links for details. The point of System Functions is that there are files you might wish to profile, but always skipping some large functions which clog up the profiling results, and which you don't normally mean to profile.

Both of these settings are global to AQtime, they work equally for all projects. There are times where you might wish they weren't so restrictive. Rather than try to undo and redo them, you may use the first item on the Options submenu, **Ignore System File Settings**, which is normally off but can be checked on — and then back off to restore normal behavior. If you choose Ignore on a serious project, make sure that you enable other means of restricting what gets profiled. See *Controlling What To Profile*.

Finally, you can also define "system" files to exclude from the current project only. This is done through the **System Files** item of the **Project** submenu. See *Project System Files*.

Defining Areas To Profile

AQtime offers several means of **restricting** what parts of an application get profiled. **Areas** (with the associated **checking**) are perhaps the most important of these means. But they work in association with the other means. In the end, what gets profiled is what currently falls under no restriction of one kind or another. See *Controlling What To Profile* for full details.

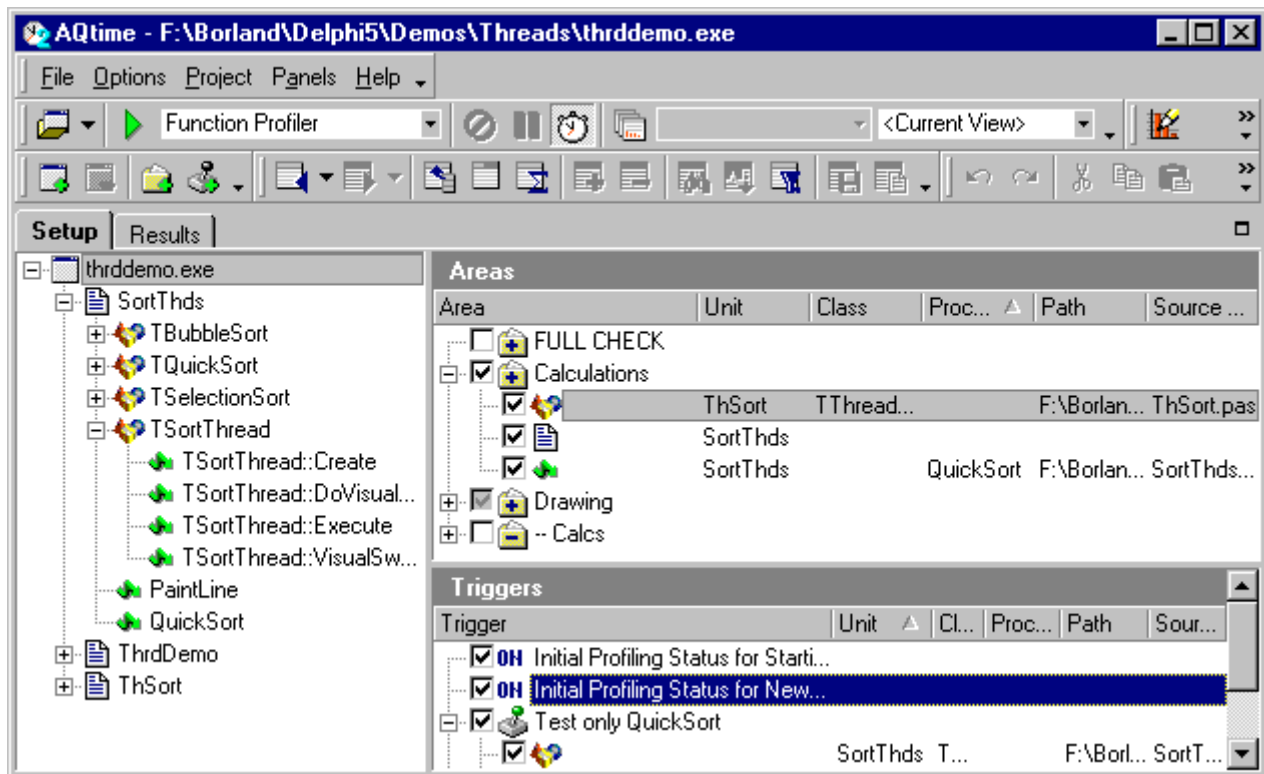
But first note that certain profilers ignore Areas and checking: VCL Class, Reference Count, Platform Compliance, ATL Reference Count, Memory and API Resource Check and BDE SQL profilers. They always profile the entire project. The rest of this topic only applies to the other profilers.

Once a project is open (that is, an application has been selected for profiling, see *Opening a Project*), the Setup panel is where you set and change most parameters that will apply across profilers. These are Areas and Triggers.

Areas are collections of **elements** to profile. Elements may be source files, classes or single functions. An element may be lodged in more than one area. The reason for having several areas is that an area can be turned on or off by checking or unchecking it. Each area represents, if you wish, a "typical **profiling interest**" for you. Each profile run can take in one or several areas. (If it takes in none, it profiles nothing, unless it's using one of the whole-application profilers listed above.)

By default, an Area is **Including** – it adds elements to the profiling list. However, sometimes you may one to include *almost* all the methods from a class, or might wish to include a unit, but skip two functions in it. You can always do this by adding the wanted elements one by one to an (Including) area. But, for convenience, you also have the option of adding the entire class or unit to an Including area, and then defining a special **Excluding** area to prevent the profiling of the few sub-elements you wish to skip.

So, an Area by itself doesn't define what you will profile, but what you might wish to profile. What will actually get profiled in a given run, barring other restrictions, is only the checked elements within the checked areas, barring their also being checked in an Excluding area. Checking is the object of another topic, *Checking Elements to Profile*. The object of this one is how to build up and maintain the collections of checkable elements, that is, Areas themselves.



The right-hand pane of the Setup panel holds two lists, Areas and Triggers, with Areas on top. The separator between the two is moveable; make sure you do see the Areas list. Here is how you manage it –

When you begin a new project, there is one preset area, FULL CHECK – the entire project, with no separate elements at all. This requires no management – check it if you want everything, leave it unchecked otherwise.

For any normal area, first you need to add it to the list. (It will start out empty.) Click on **Add area...** in the context (right-click) menu and give the area a name in the edit box provided. By default, this will be an Including area, but you can click the Excluding radio button in the dialog to make it an Excluding area. Once you close the dialog, the new area will appear in the Areas list, with a big + in its icon if it is an Including area, else a big -.

Add elements to the area. The easiest way to do this is to go to the left-hand panel. This displays a treeview of the entire application, any part of which you can expand or contract. You can make single or multi- selections in the view, using click, shift-and-drag or Ctrl-click. You can for instance select an entire unit, then unselect parts of it by Ctrl-clicking them off. You add the selected elements to the area you want by dragging from the tree-view and dropping onto the area in the right-hand pane.

The other way to add elements to an area is to open the context menu on the area and use Add Units (source files), Add Procedures (or functions, or methods) or Add Classes.

Both of these methods can be used to add elements at any time to any existing area.

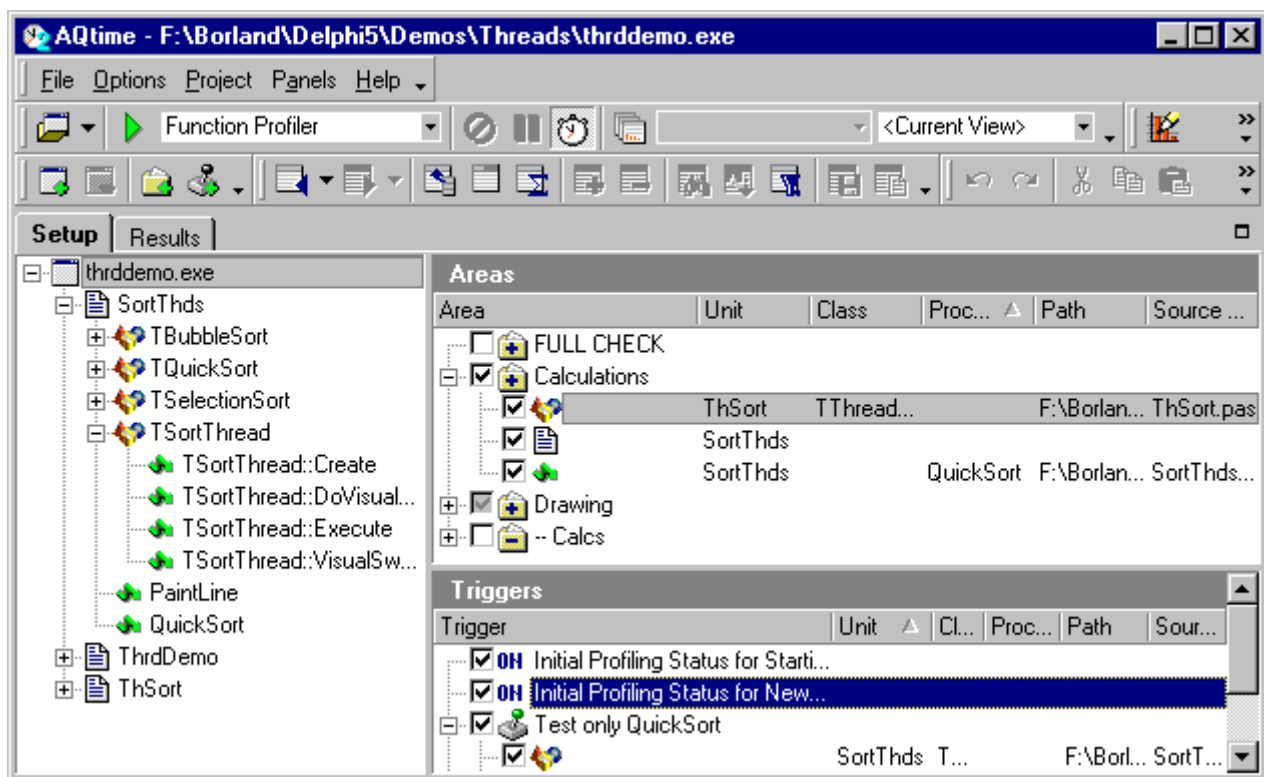
Elements can also be removed at any time. They can be dragged away from the area, and dropped onto the left-hand pane. More conveniently, you can uncheck the elements you wish to remove, then use Remove Unchecked Elements from the context menu. Alternatively, you can check only the elements to remove and use Remove Checked Elements.

Remember that you don't have to use an area as-is. The point of adding or removing elements is to create a "stored definition", which you can later trim simply by unchecking elements. It is perfectly reasonable to run a profiler on an entire area, then begin unchecking elements for each successive runs, as you eliminate the uninteresting parts.

Be careful that when you profile a function, part of its execution time will be spent calling other functions, which we call "child" functions (it's the calls that are child calls, actually). Unless the child functions are part of the profiling area, you will not be able to narrow down your profiling to find bottlenecks outside the main function. Profiling results can discriminate between a function's own execution time and its overall execution time, "with children". But if you want to know about the children themselves, they have to be added to the profiling area. See Function Profiling Restriction.

Checking Elements to Profile

Areas are collections of elements (units, classes or functions) which you keep together because you might want to profile them. In other words, the Areas list is a repository of profileable elements grouped into collections called Areas, and an element may belong to more than one collection.



Normally, as you start profiling an application, you use simple profilers (such as Hit Count) over broad areas. Once you better know what you want to track, you may use profilers that yield more detailed information (such as Tracing), but run them over smaller areas. So, in an ordinary profile test, the pattern is that, from run to run, you set Areas so as to profile less and less.

On any run, only the Areas you have checked will be profiled. If no Area is checked, no profiling will occur. But very often you will also want to trim down *within* an Area. Open any Area in the tree view, and you see the elements you put in it. You can temporarily remove any element from the profiling by unchecking it individually.

You can also use the common multi-select commands (Shift-drag and Ctrl-click) to select a larger number of elements at once, then use the context (right-click) menu's **Check Selected** or **Uncheck Selected** commands.

When you begin a new profiling session, remember to open the Areas you have checked, so as to see what elements are currently checked or unchecked within them. If you check an Area, only the currently checked

elements in it will be profiled. If an Area is not checked, then none of its elements appear checked. Checked elements only show up once the Area is checked.


All of the above is meant for the normal, Including, areas. The elements of Excluding areas only become excluded if their area is checked, and they themselves are checked. Normally, you will simply check any Excluding area when you check the Including area for which it holds exceptions.

Using Triggers




Triggers are an AQtime facility allowing better control of profiling under the three function profilers, Function HitCount, Function Trace and Function Profiler. "Triggers" include three linked settings, On-Triggers, Off-Triggers and Initial Profiling Status (Starting and New).

AQtime offers several means of **restricting** what parts of an application get profiled. Triggers provide a crucial means to fine-tune exclusions, but they work in association with the other means. In the end, what gets profiled is what currently falls under no restriction of one kind or another. See Controlling What To Profile for full details.

Especially, since Areas and Triggers are rather similar, it is important to understand that, no matter what Triggers and Initial Status may decree, nothing will get profiled if it is not checked in the Areas section. The one exception is the on-trigger function itself, which doesn't need to be checked in Areas to get profiled. But the functions it calls will not be profiled if they are not checked.

The purpose of Triggers is to allow profiling to come on when certain functions (on-triggers) are executing (including during their calls to "child" functions), or on the contrary to be turned off (off-triggers). With Areas and checking, you set what you **never** want profiled during the current run (i.e., everything but the checked elements). With Triggers and Initial Status, you set in **what part of the execution path** you want to enable profiling, or to disable it. It's like having a very smart robot to press the  Enable/Disable Profiling button at the right moments – perhaps thousands of times in a run.

Technically, what triggers do is very simple to explain –

- For any given thread, at any given moment, **profiling status** is "enabled" or "disabled". Profiling actually operates in a thread if, one, status for that thread is "enabled" and, two, nothing else excludes the current function from profiling.
- The  Enable/Disable Profiling button is an onscreen way of controlling profiling status by hand while the application runs. If it is pressed-in () , profiling status is as set otherwise. When you unclick it () , profiling is turned off for all threads. When you press it back in, you re-enable profiling, according to whatever is otherwise set for the application *at that point* (not as it was when you disabled it).
- If there are triggers of any kind, the Initial Profiling Status settings define whether profiling is enabled or not before any trigger starts executing. After that, they have no effect. Likewise if there are no triggers at all. There are two Initial Status settings, one for the application's starting thread, and one for any new thread created in the course of execution. In other words, **Initial Profiling Status for Starting Thread** "triggers" profiling on or off at application launch, for its starting thread, and **Initial Profiling Status for New Threads** triggers it on or off on the creation of each new thread.
- When an **on-trigger** function begins executing, it saves the current profiling status for its thread and enables profiling for that thread. It also turns profiling on for itself, independent of Area settings. These still apply to everything it calls. When it reaches the end of its execution (after perhaps hundreds of calls, sub-calls and sub-sub-calls), it restores the thread's profiling status as it found it.
- When an **off-trigger** function begins executing, it saves the current profiling status for its thread and turns profiling off for that thread. When it reaches the end of its execution, it restores the thread's profiling status as it found it.

Suppose, Proc_B is an off-trigger routine, profiling is currently enabled and the FULL CHECK area is used:

```
Proc_A;
Proc_B      // off-trigger routine
    Proc_D;    // Proc_D and Proc_E are child routines of Proc_B,
    Proc_E;    // that is, they are called within Proc_B.
              // Proc_D and Proc_E are not profiled.
Proc_C;
```

As profiling is enabled, and FULL CHECK is on, AQtime profiles Proc_A. When the application enters Proc_B, profiling is disabled for that thread. So Proc_D and Proc_E are not profiled. When Proc_B exits, AQtime restores the profiling status as it was – enabled -- so Proc_C is profiled.

This topic has explained the **use** of triggers. It has not explained **options** for triggers. And it has not explained **how to set them up**. Both questions are the object of the next topic: *Setting Up Triggers*.

Setting Up Triggers

Before proceeding to set up triggers, make sure you have read Using Triggers. A few reminders --

- Triggers work in conjunction with the other means of selecting what to profile. Execution is actually profiled only when all means say "yes". See Controlling What To Profile.
- Of all means of selecting what to profile, Triggers are the only one that can select on a **thread** basis.
- "Triggers" include three linked settings, On-Triggers, Off-Triggers and Initial Profiling Status (Starting and New).
- Triggers only apply to the function profilers: Function HitCount, Function Trace and Function Profiler.

Triggers are defined and controlled in the Setup panel. The right-hand pane holds two lists, Areas and Triggers, with Triggers at the bottom. The separator between the two is moveable; make sure you do see the Triggers list.

In that list, a "trigger" is a collection of *potential* trigger functions of one type, either on- or off-. No triggers operate at all unless one such collection (at least) is **checked**. And in the collection only those triggers operate which are checked also.



Up to this point, the mechanics of managing Triggers, including checking them on or off, is identical to that of managing Areas. We refer you to Defining Areas and Checking Elements for details.

Now for the points that are **different** with setting up Triggers:

- Note that if one element of a Trigger is a unit, for instance, and you check it, then each single function in that unit becomes a trigger. Uncheck it and no function in the unit acts as a trigger, unless it also belongs to some other checked collection, and it is checked in there. In other words, you should be more conservative when including elements in Triggers than when including them in Areas. After a certain point, more Triggers simply mean more confusion, when the whole purpose of Triggers is to clarify profiling results.
- A "trigger" collection is defined as holding **on** or **off** triggers (one type per collection), not just triggers.
- There are two default triggers always at the top of the list, which set the profiling status at the beginning of threads, before specific triggers click in. (After that, they have no effect.) These are name **Initial Profiling Status for Starting Thread** (that is, for the application's main thread) an **Initial Profiling Status for New Threads**. (that is, for secondary threads – one status for all threads). These can be set on (profiling allowed until a trigger operates) or off (no profiling until a trigger operates).

- Except for the Initial Status triggers, each trigger can be tuned regarding **after how many calls** a trigger will start operating, and then **for how many calls** (after which it will become inoperative). These two options are themselves affected by another option – is the call count taken over all threads, or over each thread individually? This yes-no option, which sets the meaning of the other ones explained below, is **For All Threads**. Remember that, whether or not calls are counted over all threads, triggers only act on their own thread.
- The **Pass Count** option allows the trigger function or functions to be called a certain number of times before they act as triggers. For instance, this lets you skip the startup phase of your application. Pass Count 3 means "only act as trigger on the fourth call". Default is 0, operate from first call.
- The **Work Count** option then sets how many consecutive calls will act on the trigger (to allow profiling or to disallow it), before the trigger stops acting again. This is a way to limit the amount of data gathered at the point where it just repeats what's known. 0 is again the default, but here it doesn't mean "never activate", it means "stay activated to end of run".
- Finally, the **Cycling** yes-no option (default no) sets whether the pass-count-work-count cycle will repeat after work count + 1 is reached and the trigger is deactivated. A cycling trigger is a way to sample application behavior through various phases without amassing data for every single call.

Selecting a Profiler

One AQtime "run" is one execution of the application under one profiler. The profiler to be run is set from the  dropdown list on the standard toolbar, "Select a profiler", just to the right of the  **Run** button.

The dropdown list is actually a treeview. Individual profilers are listed when you open a branch.

See also *AQtime Profilers*.

Doing One Profile Run



Before doing your first run of your application under an AQtime profiler, you must have completed these **preliminary steps** –

- **Compiling** your application with debug information
- **Opening** the project (application) in AQtime
- Controlling **what to profile**
- Selecting the **profiler** to run





Between each run, you are likely to change what to profile, and fairly often to change profilers also. And, during the life of the project, you will of course frequently recompile your application after making changes.

Now, with the four preliminary steps completed, there are a few more **checks to go through** before starting the run:

- Check that you have set the **running conditions** as they need to be. For an exe, these are the (possible) runtime arguments, for a DLL, the (necessary) host application. Most often, you don't have to do anything, because you are testing an exe that takes no parameters, or because you simply want to keep the existing settings. To check or change conditions, use **Project | Parameters...** from the main menu. This leads you to the Run Parameters dialog, which has a box both for parameters and for host application. See *Profiling Dynamic Link Libraries* in on-line help.

- Make sure that the **necessary modules** (such as statically linked DLLs) can be loaded.
- Check that the  **Enable/Disable Profiling button** on the Standard toolbar is in its normal pressed-in state (as shown), unless you want to start with profiling turned off, overriding your Trigger settings.
- You launch the application by pressing  **Run** on the Standard toolbar, or F9 on the keyboard.

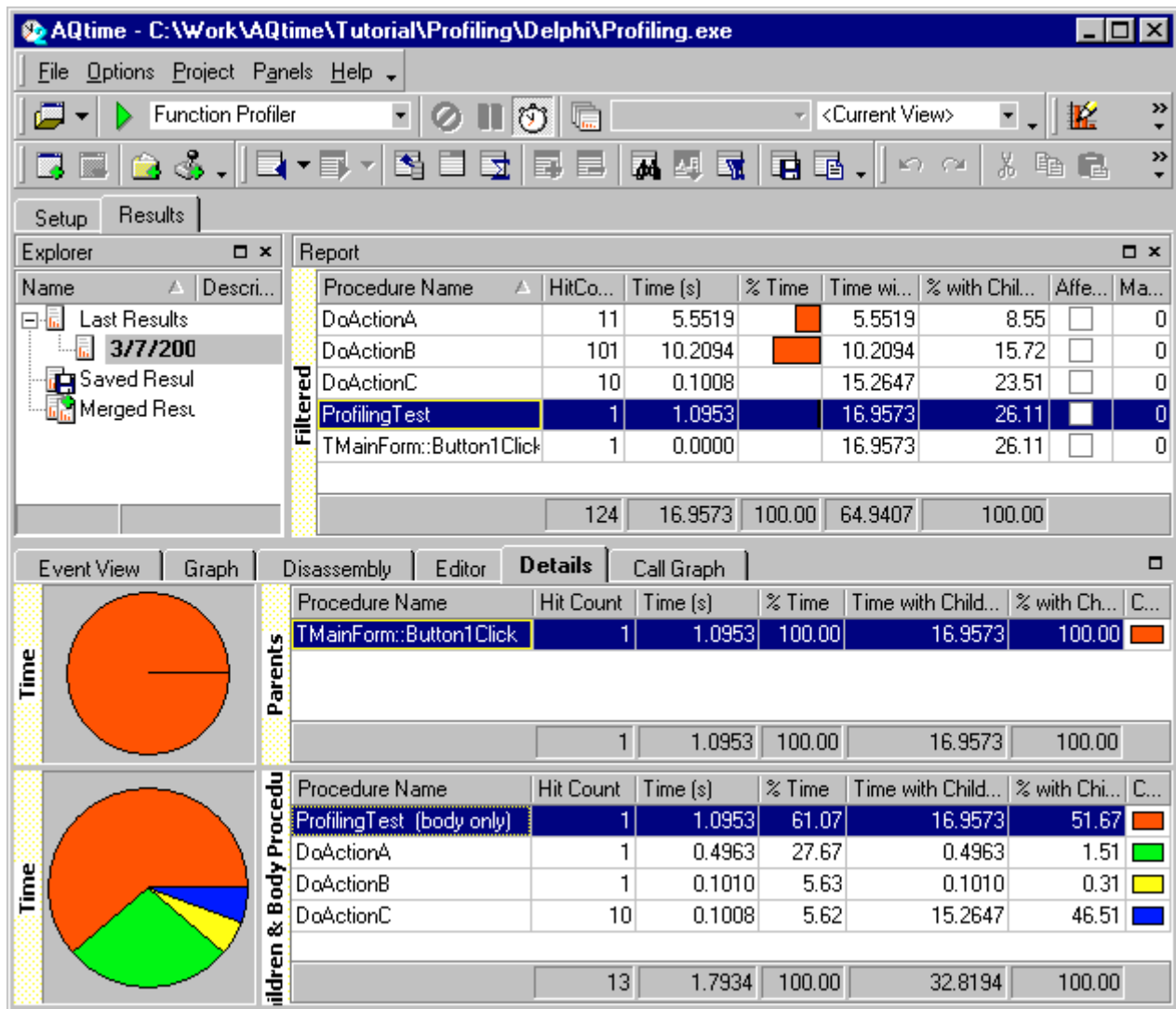
Some points while executing the application:

- Run the operations that you need to profile (for instance, those where you suspect a bottleneck). You might **plan your run** before starting, to be sure you hit all the high (or rather, low) points.
- If you need time to reflect on what to do next, or simply to check screen output, don't hesitate to use the  **Pause Program** button on the Standard toolbar. Press  **Resume** to resume execution.
- You can use the  **Enable/Disable Profiling** button to suspend profiling, but not execution, while you run through parts of the application you don't need to profile. See Controlling What To Profile for the caveat.
- If you want to force and end to the application process, exit the profile run and get results, press  **Reset**. This may be needed if the application cannot normally be ended without rebooting, logging off or some other drastic intervention, or if it simply has stopped responding.
- Note that some profilers, with some over-enthusiastic settings, may **slow** application execution to the point where you mistake this for a crash.
- Once you have gone through the operations you wanted, **exit** the application. Do not accumulate needless profile data.

Once you exit, the resultant profile will be displayed in the AQtime Report panel.

Analyzing Profiler Results

After profiling your application, results are displayed in the Report panel, usually on the Results tab:



Organization

This panel shows a table where each row corresponds to a function or line that has been profiled. When you select a row, other panels are updated to display information for that function or line. (Not all panels apply to all profilers).

- The Call Graph panel displays the call information for the selected function.
- The Details panel holds additional profiling results for the function, e.g. the call stack, "child" and "parent" functions, etc.
- The Disassembly displays binary code generated for the function or line.
- The Editor panel displays the source code of the selected function or line.
- The Event View panel is unaffected; it simply logs events that occurred during the profile run.
- The Graph panel is unaffected; it displays overall results in a comparison graph.

For **multithreaded** applications, AQtime stores profiling results for each thread as well as results for whole application. To view thread results, select the desired thread from the Explorer panel or from the **Threads** dropdown list on the Standard toolbar. See *Profiling Multithreaded Applications*.

Managing results



- You can **sort** results by any column.
- You can **group** results by one or several columns.
- You can **search** results using the Find dialog or the Incremental Search feature.
- You can **add** summary fields.
- You can **filter** results.
- Better yet, you can apply a **view** from the many pre-defined ones, or from those you define yourself. A view combines a filter, a layout for the Report panel and one for the Graph panel.
- You can also **compare** current results with previous ones to find the effects of changes you made in the application (or in the way you ran it).
- And you can **merge** results accumulated over several tests to constitute a benchmark.

Transferring results

Profiling results can be –

- **copied** to the Clipboard,
- **printed** using the Print Preview Form,
- **exported** to text, Excel, html or xml formats (see Exporting Data).
- **inserted** into application source code (see Inserting Results Into Source Code).

More usability features

- While you use the Report or the Details panels, AQtime records your movements from item to item. You can come back to something you selected previously, and then return to where you jumped back from, as with an Internet browser. Use the  **Back** and  **Forward** buttons on the Report toolbar. See AQtime Panels.
- AQtime's visual means for arranging grids apply to the display of results, of course. Especially, you can:
 - **change** column width and ordering,
 - **hide or show** columns (not all columns are displayed by default).


Views Implementation

A **View** is a group of settings for displaying results. Creating and using views (including the many pre-defined ones) is a great way to accelerate, simplify and clarify the analysis of your results. A view can serve not simply as a preset format, but as a preset question to which you get an immediate, clear answer by switching to the view. Many of the pre-defined views are of that kind, and you can define more for your own frequently asked questions.

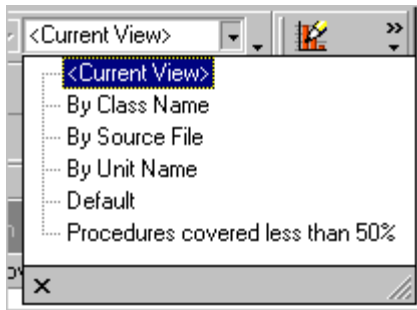
A view is defined for one specific profiler, and stored in the profiler's single .qtview file, which groups all the views currently defined for it. By default, the stored settings are for:

- filter expression,
- column layout for the Report table,
- Graph panel layout.

Some later plug-in profilers may add to the items that are stored in their views.

You can add a new view to store your current filter and panel settings, for the current profiler, simply by pressing the  **Define Views** button on the Standard toolbar and using the ensuing **Define View** dialog. In that dialog, you will also find that you can export or import views between profilers or between different AQtime installations.

The views available for the current profiler are displayed in the **View** dropdown list on the Standard toolbar:



Until you add your own views, this will display the views that are shipped with AQtime, and which are defined for all applicable standard profilers. The rest of this topic is devoted to capsule explanations of these standard views. They are:

Current

Default

By Class Name

By Source File

By Unit Name

Top 10 Procedures

Top 20

Covered

Uncovered

Leaked Classes Only

Unreleased Classes Only

Top 10 Procedures (Net Time)

Top 10 Procedures (Time w. Children)

Top 10 Executed Procedures (Hit Count)

Top 10 % (Net Time)


Top 10 % (Time w. Children)

Top 10 %

Top 5 %

Procedures Covered less than 50%

Default View is what AQtime uses when executing AQtime for the first time. If you change a series of parameters for the **Report** table or **Graph** diagram and then select Default View, AQtime automatically restores its standard settings.

Current View is simply the name for whatever settings are currently active. It is not a stored view, but you can make it so by saving it under some name using the  Define View button, right to the left of the list. You will then be able to retrieve it for re-use. Note that it applies only to the current profiler (unless you later use export and import in the Define Views dialog).

By Class Name, By Source File, By Unit Name allow you to group the Results table by the corresponding fields. These views are available for all profilers except VCL Class Profiler and Reference Count Profiler.

The **Top 10 Procedures** view is enabled only for the Function HitCount, Function, and Function Sampling profilers.

- For **Function HitCount** Top 10 shows the ten most used procedures.
- For the **Function Profiler** Top 10 shows the ten procedures that take the longest time to execute – a quick performance measurement.
- For **Function Sampling** Top 10 shows the ten most used procedures – a different quick performance measurement.

The **Top 20** view is available only for the Line HitCount and Line Sampling profilers.

- For **Line HitCount** Top 20 shows the twenty lines that have been executed the most times.
- For **Line Sampling** Top 20 shows the twenty lines of code which take the longest time to execute. (The 20 slowest lines of code based on processor execution).

The **Covered** and **Uncovered** views are available for the Function Coverage profiler only. The Covered view displays all executed procedures. The Uncovered view displays all procedures that did not execute.

The **Leaked Classes Only** view is for the VCL Class Profiler. It filters the results to show only the objects not freed after program termination. It is named "classes" because the Class Profiler displays the class in the Report panel and the instance in Details.

The **Unreleased Classes Only** view is for the Reference Count Profiler. It filters results to show only classes implementing interfaces for which the reference count has not been brought back to 0.

The **Top 10 Procedures (Net Time)**, **Top 10 Procedures (Time w. Children)**, **Top 10 Executed Procedures (Hit Count)**, **Top 10 % (Net Time)** and **Top 10 % (Time w. Children)** views are available for the Function Profiler only.

Top 10 Procedures (Net Time) displays the ten procedures that execute the slowest in their own code (net), independent of the functions they call.

Top 10 Procedures (Time w. Children) displays the ten slowest procedures, counting all time spent between entry and exit, including "child" calls.

Top 10 Executed Procedures (Hit Count) displays the ten procedures called most often.

Top 10 % (Net Time) is the same as Top 10 (net), except that the list is extended to the top tenth of all procedures (typically a hundred or more).

Top 10 % (Time w. Children) is the same as Top 10 (w. Children), except that the list is extended to the top tenth of all procedures (typically a hundred or more).

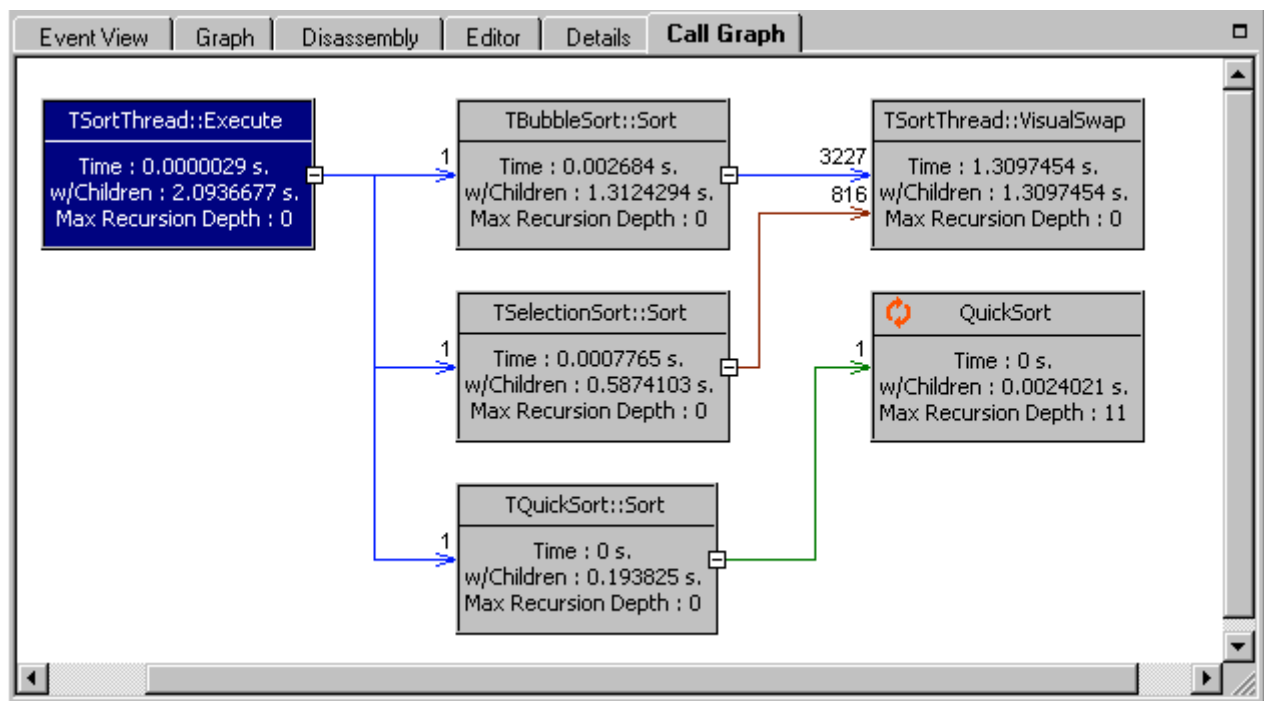
The **Top 10%** and **Top 5%** views are available only for the Function Sampling and Line Sampling profilers. They display the 10% or 5% worst-performing functions or lines, that is the ones that occupy the most CPU time in their own code).

The **Procedures Covered less than 50%** view is available only for the Line Coverage profiler. It displays procedures where fewer than half the source lines were executed.


Panels Reference

Call Graph Panel

The **Call Graph** panel is used with the Function Profiler and the Function HitCount profiler (when its *Call relationship tracking* option is enabled) to display the hierarchy of calls for the function double-clicked in the Report panel:



Each function is represented as a rectangle and the arrows show the sequence of function calls. In the upper part of the rectangle there is the function name, in the lower one, profiling results. Profiler option *Display in Call Graph* sets what results will be shown in the lower part of the function rectangles. (See Profilers Options - Function Profiler and Profilers Options – Function HitCount).

The numbers at the arrow starts (from parent functions) specify how many times the highlighted function was called from the parent one. Those at the arrow heads (towards child functions) specify how many times a child was called from the highlighted function. Recursive calls are marked with .

The number of "child" and "parent" levels in the chart is specified by options of the same name (*Number of child levels* and *Number of parent levels*). Increase these settings to show more of the function call hierarchy. Decrease them to have a simpler display.

Double-clicking on a function rectangle in the panel has the same effect as double-clicking on that function in the Report panel – the Call Graph panel updates to highlight the clicked function and show its parents and children, and other panels update accordingly (Editor, Disassembly, etc. -- see *AQtime Panels*).

The context (right-click) menu has **Go to Child Procedure**, **Go to Parent Procedure** and **Go to Current Procedure** to help navigating the hierarchy when it is large. It also has **Zoom In**, **Zoom Out**, **No Zoom** and **Fit** to help you get the view you want.

Details Panel

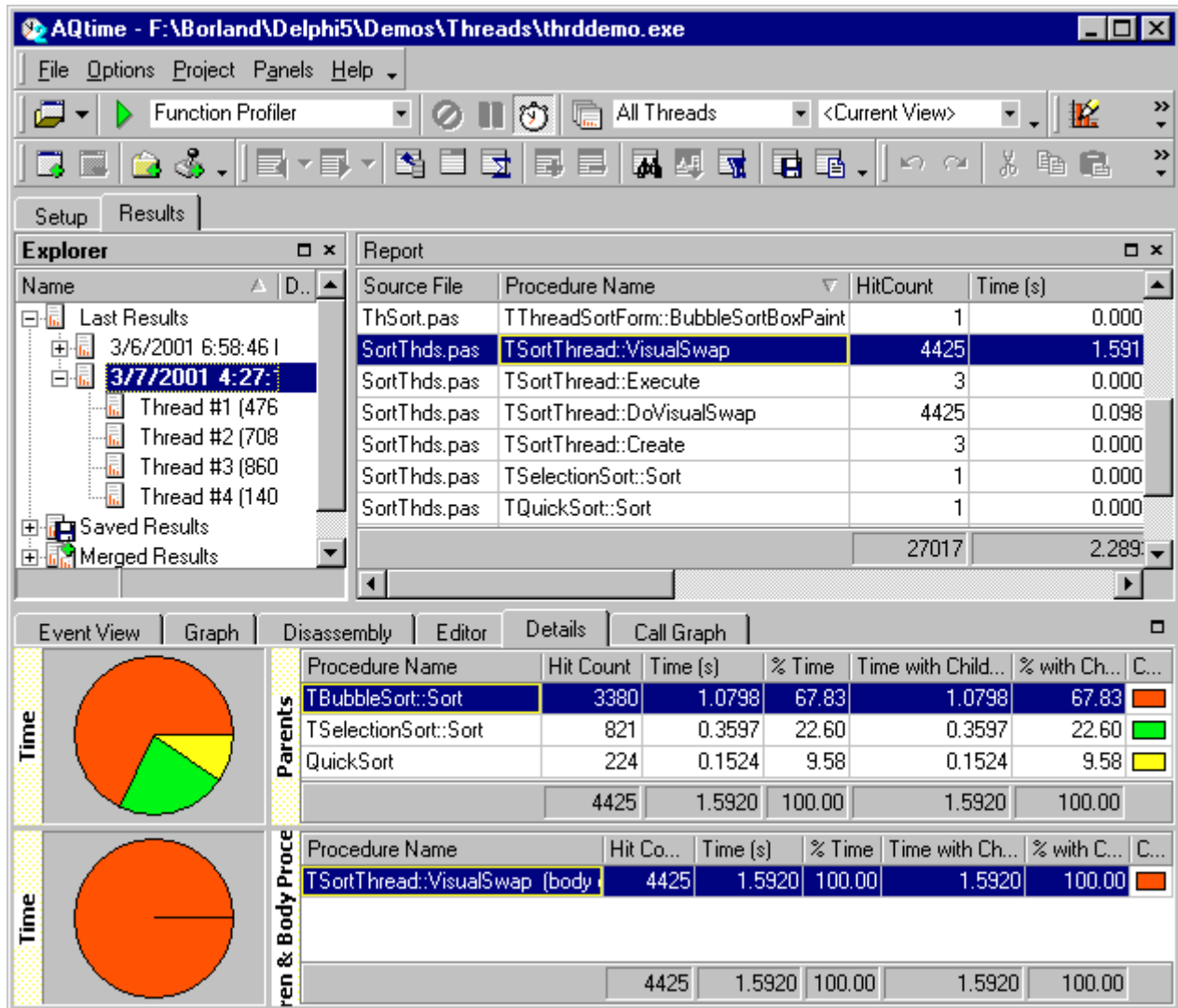
The **Details** panel displays additional results for the line selected in the Report panel with a number of profilers. The details displayed depend on the profiler. See

- Function HitCount Profiler - Details
- Function Profiler - Details
- VLC Class Profiler - Details
- VCL Reference Count Profiler - Details
- Memory and API Resource Check - Details
- ATL RefCount Profiler - Details
- BDE SQL Profiler - Details
- Unused VCL Units Profiler - Details

You can arrange the Details panel the same way you can organize other AQtime panels.

Function Profiler - Details

When using the Function Profiler the Details panel looks like this example:



There are two panes. The upper, *Parents*, displays procedures that call the function double-clicked in the Report panel; The lower, *Children*, displays the functions it calls.

As always, the only functions shown are those that were profiled. If you have a ParentFunction procedure that calls a child function and the profiling area includes only the ParentFunction, the child function is not profiled and the Details panel does not display its results. If a function is costing you some time, make sure you include its child calls, for instance by making the function a Trigger. (See also Function Profiling Restriction). Otherwise, you'll lose the ability to tell how much of that time cost goes to child calls.

In the panel, on the left hand of each pane is a chart, which is either a pie chart or a bar chart, depending on the *Chart settings* option of Details panel. This chart compares one result for each function on the right side of the pane. You can select which result is compared by using the context (right-click) menu on the chart section. Each function line on the right has a **Color** column that lets you select the color used in the chart for that particular line.

Apart from Color, the columns are similar to those in the Report panel. However, there are some differences:

Parents pane



- The **Time (s)** and **Time with Children** columns do not report totals for all calls of the named function, as they would in the Report panel. They report totals for those calls which called the currently selected function (in Report).

- **Time %** and **% with Children** are calculated against the totals for Time (s) and Time with Children in this pane, not in Report. They give you the relative speed of the parent functions in those calls where they call the selected function.

Children pane

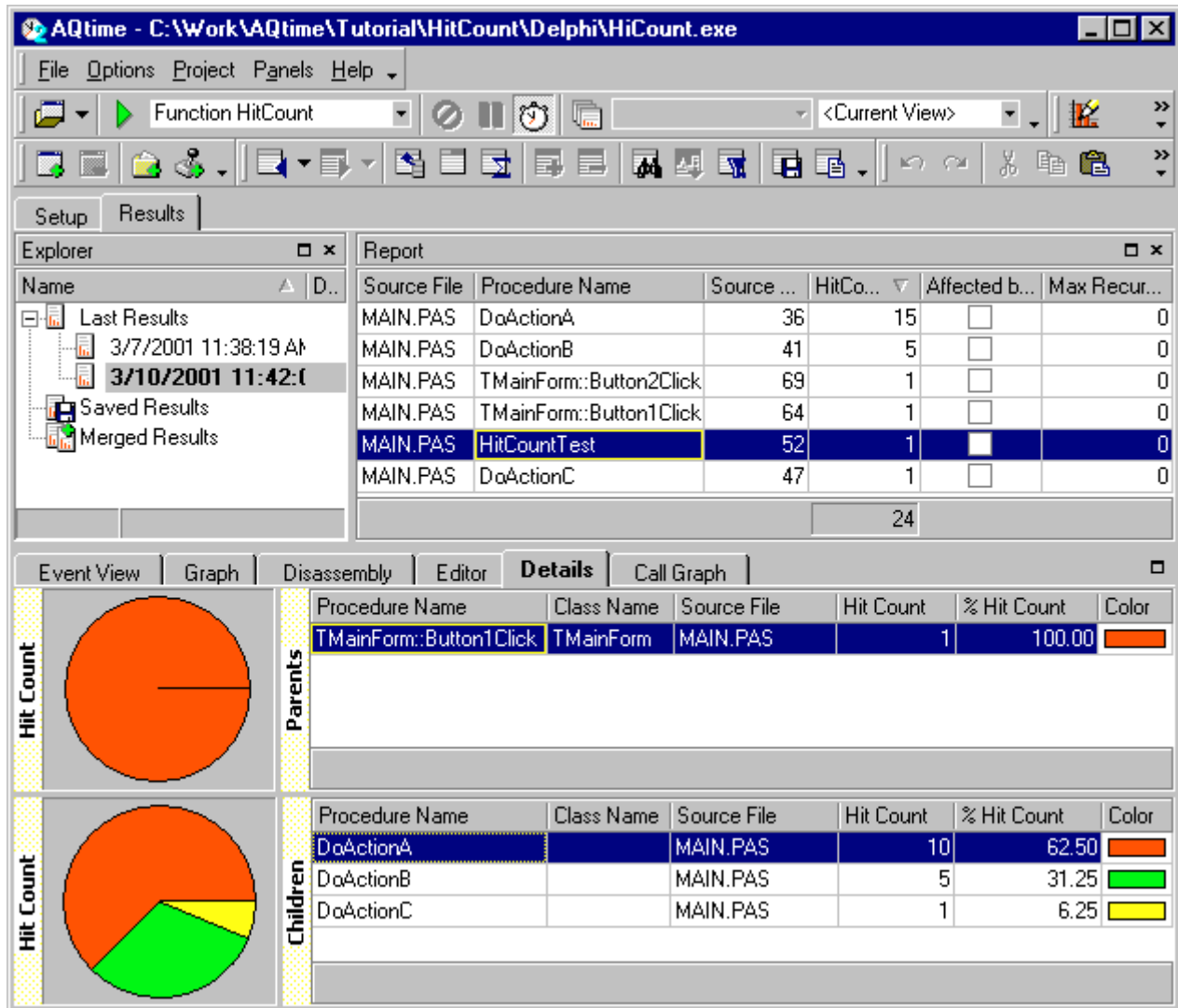
- The **Time (s)** and **Time with Children** columns do not report totals for all calls of the named function, as they would in the Report panel. They report totals only for those calls that came from the currently selected function (in Report).
- **Time %** and **% with Children** are calculated against the totals for Time (s) and Time with Children in this pane, not in Report. They give you the relative speed of the child functions among the other child functions of the selected function.

If the *Show the function body in Details* option is set for the Function Profiler, there is a line with the name of the selected function with "(body only)" added. This gives the results for the part of the function that was *not* child calls – in other words, the function's own code.

When you double-click a line in Details, the corresponding function is selected in the Report panel and the Details panel will display the parent and children of the new selected function. Movements between routines are tracked, as they would be between pages in a Web browser. The  **Back** and  **Forward** buttons on the Report toolbar allow you to retrace your steps.

Function HitCount - Details

The Function HitCount profiler will gather results to display in Details only if its *Details* option is on (see Profilers Options – Function HitCount). When it is in use, the Details panel holds two panes. The upper, *Parents*, displays procedures that call the *examined* function, that is, the function double-clicked in the Report panel. The lower, *Children*, displays the functions called by the examined function:



On the left hand of each pane is a chart, which is either a pie chart or a bar chart, depending on the *Chart settings* option of Details panel. This chart compares the hit count for each function on the right side of the pane. Each line on the right has a **Color** column that lets you select the color used in the chart for that particular function (press the ellipsis button).

The same columns appear in Details as do in the Report panel, with a few differences. Color, as just noted, only exists in Details. Affected By Trigger is absent from Details. **Hit Count** and **% Hit Count** are calculated differently from the same columns in the Report panel.

In the **Parents** pane, **Hit Count** only counts those calls of the function on that line, where the function in turn called the examined function. **% Hit Count** is this "calling" hit count calculated as a percentage of the sum of all counts in that column. In other words, % Hit Count tells you where the examined function is most often called from, where it is most rarely called from, etc..

Likewise, in the **Children** pane, **Hit Count** only counts the calls made to child functions by the examined function, excluding all the hits where they were called from elsewhere. **% Hit Count** is this value calculated as a percentage of the sum of values in the Hit Count columns. In other words, it tells you which child function the examined function called most frequently, least frequently, etc.

VCL Class Profiler – Details

For the VCL Class Profiler, the Report panel shows a list of all VCL classes for which instances were created during the run. For the class currently selected in Report, the Details panel shows those instances that were **leaked**, that is, never freed, and gives the state of the **call stack** at the point where they were created (Note that the call stack is traced only if the *Stack / Show call stack* option is enabled):

The screenshot displays the AQtime VCL Class Profiler interface. The 'Report' panel shows a table of VCL classes with columns: Class Name, Current (Leaked), Total Created, Total Size, Peak Created, and Instance Size. The 'Instances' panel shows a table of leaked objects with columns: Class Name, Instance Address, #, and Parent Thread. The 'Call Stack' panel shows a table of the call stack with columns: Unit Name, Source File, Class Name, Procedure Name, and Line No.

| Class Name | Current (Leaked) | Total Created | Total Size | Peak Created | Instance Size |
|-----------------|------------------|---------------|------------|--------------|---------------|
| TObject | 100 | 100 | 400 | 100 | 4 |
| EInvalidPointer | 0 | 1 | 0 | 1 | 16 |
| EOutOfMemory | 0 | 1 | 0 | 1 | 16 |
| TApplication | 0 | 1 | 0 | 1 | 272 |
| TBits | 0 | 1 | 0 | 1 | 12 |
| TBrush | 0 | 8 | 0 | 8 | 24 |
| TButton | 0 | 2 | 0 | 2 | 512 |
| | 100 | 299 | 400 | 294 | 3924 |

| Class Name | Instance Address | # | Parent Thread |
|------------|------------------|-----|---------------|
| TObject | \$00E41C14 | 124 | 1280 |
| TObject | \$00E41C20 | 164 | 1280 |
| TObject | \$00E41C40 | 166 | 1280 |



| Unit Name | Source File | Class Name | Procedure Name | Line No |
|-----------|-------------|-------------|----------------------|---------|
| System | system.pas | | ClassCreate | 4157 |
| System | system.pas | TObject | TObject.Create | 3517 |
| Main | MAIN.PAS | | VCLclassUsageTest | 50 |
| Classes | Classes.pas | TPersistent | TPersistent.AssignTo | 2473 |

The **Instances** pane holds the list of leaked objects (class name, memory address, object number in creation order for the class). The **Call Stack** pane shows the state of the call stack at the moment the currently selected object was created. An object is selected by double-clicking on its line in the Instances pane. The Call Stack pane displays the following columns:

- Procedure Name** Caller.
- Class Name** If caller is a method, name of its class.
- Module Name** Module (exe, dll, etc.) where the calling code is located.
- Unit Name** Unit where the caller is declared.

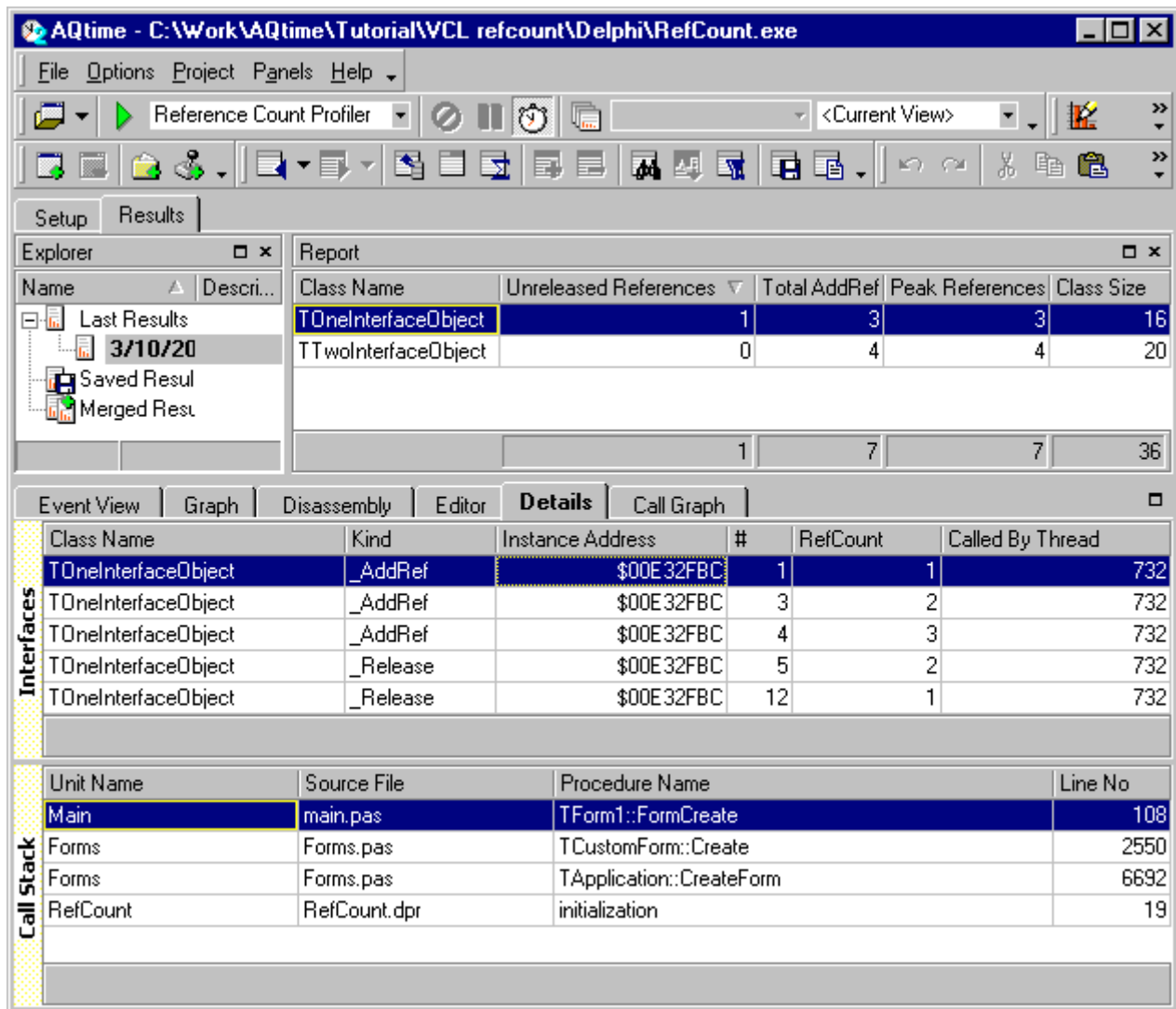
| | |
|-----------------------|--|
| Source File | File holding the source for the caller. |
| Line No | First line of the implementation of the caller in Source File. |
| Stack Entry No | Caller rank in the call stack. The topmost caller has index 1. |

The topmost line details the function which called Create for the selected object. The next line is the function that had called this one, and so on down the list.

Double-clicking a function in the Call Stack panel will update the Editor and the Disassembly panels. You can then simply move to those to inspect the function's source or binary code (see *AQtime Panels*). The  **Back** and  **Forward** buttons on the Report toolbar let you move back and forth between functions.

VCL Reference Count Profiler – Details

For the VCL Reference Count Profiler, the Report panel shows a list of all interfaced classes for which references were added during the run. There is an Unreleased References column. If a class is selected in Report and Unreleased References is not 0 for this class, then the Details panel will show its `_AddRef` and `_Release` history throughout the run, and the state of the **call stack** on each of those calls. Note that the call stack is traced only if the *Stack / Show call stack* option is enabled:



Report

| Class Name | Unreleased References | Total AddRef | Peak References | Class Size |
|---------------------|-----------------------|--------------|-----------------|------------|
| TOneInterfaceObject | 1 | 3 | 3 | 16 |
| TTwoInterfaceObject | 0 | 4 | 4 | 20 |
| | 1 | 7 | 7 | 36 |

Details

| Class Name | Kind | Instance Address | # | RefCount | Called By Thread |
|---------------------|----------|------------------|----|----------|------------------|
| TOneInterfaceObject | _AddRef | \$00E32FBC | 1 | 1 | 732 |
| TOneInterfaceObject | _AddRef | \$00E32FBC | 3 | 2 | 732 |
| TOneInterfaceObject | _AddRef | \$00E32FBC | 4 | 3 | 732 |
| TOneInterfaceObject | _Release | \$00E32FBC | 5 | 2 | 732 |
| TOneInterfaceObject | _Release | \$00E32FBC | 12 | 1 | 732 |

Call Stack

| Unit Name | Source File | Procedure Name | Line No |
|-----------|--------------|--------------------------|---------|
| Main | main.pas | TForm1::FormCreate | 108 |
| Forms | Forms.pas | TCustomForm::Create | 2550 |
| Forms | Forms.pas | TApplication::CreateForm | 6692 |
| RefCount | RefCount.dpr | initialization | 19 |

The **Interfaces** pane lists all calls to `_AddRef` and `_Release` for the selected (reference leaking) class. It holds the following columns:



| | |
|-------------------------|---|
| Class Name | The class name. |
| Kind | Reference call -- <code>_AddRef</code> or <code>_Release</code> |
| Instance Address | Address of the object with an unreleased reference. |
| # | Instance number of the unreleased object (in creation order for the class). |
| RefCount | Number of references to the object after this call was executed. |

The **Call Stack** pane shows the state of the call stack at the moment the call currently selected in the References pane (by double-clicking). It uses the following columns:

| | |
|-----------------------|---|
| Procedure Name | Caller. |
| Class Name | If caller is a method, name of its class. |

| | |
|-----------------------|--|
| Module Name | Module (exe, dll, etc.) where the calling code is located. |
| Unit Name | Unit where the caller is declared. |
| Source File | File holding the source for the caller. |
| Line No | First line of the implementation of the caller in Source File. |
| Stack Entry No | Caller rank in the call stack. The topmost caller has index 1. |

The topmost line details the function where the `_AddRef` or `_Release` call occurred. The next line is the function that had called this one at that point, and so on down the list.

Double-clicking a function in the Call Stack panel will update the Editor and the Disassembly panels. You can then simply move to those to inspect the function's source or binary code (see *AQtime Panels*). The  **Back** and  **Forward** buttons on the Report toolbar let you move back and forth between functions.

Memory and API Resource Check - Details

See *Memory and API Resource Check Profiler*.

ATL RefCount Profiler - Details

See *ATL Reference Count Profiler*.

BDE SQL Profiler - Details

See *BDE SQL Profiler*.


Unused VCL Units Profiler - Details

See *Unused VCL Units Profiler*.

Disassembly Panel

The purpose of the the **Disassembly** panel is to allow you to check the exact binary source for the results reported by most of AQtime's profilers, independent of the compiler, version or library behind this code.

When you switch to it, the Disassembly panel shows the last function double-clicked in one of the main AQtime panels - Report, Details, Event View or Call Graph:

| Event View Graph Disassembly Editor Details Call Graph | | | | | | | |
|---|-------------------------|-----|---------------------------------|---------------------------|------------------|------|------|
| Type | Address | Hex | Instruction | Target | Target Procedure | Size | μOps |
| + begin | | | | | | | |
| + with FBox do | | | | | | | |
| + Canvas.Pen.Color := clBtnFace; | | | | | | | |
| + PaintLine(Canvas, FI, FA); | | | | | | | |
| + PaintLine(Canvas, FJ, FB); | | | | | | | |
| - Canvas.Pen.Color := clRed; | | | | | | | |
| | \$004414A4 8B45F8 | | mov eax, dword ptr [ebp - \$... | | | 3 | 1 |
| | \$004414A7 8B8020010000 | | mov eax, dword ptr [eax + \$... | | | 6 | 1 |
| | \$004414AD 8B4010 | | mov eax, dword ptr [eax + \$... | | | 3 | 1 |
| | \$004414B0 BAFF000000 | | mov edx, \$000000FF | | | 5 | 1 |
|  | \$004414B5 E83638FDFF | | call -\$0002C7CA | \$00414CF0 TPen::SetColor | | 5 | 4 |
| | | | | | | 22 | 8 |
| - PaintLine(Canvas, FI, FB); | | | | | | | |
| | \$004414BA 8B45FC | | mov eax, dword ptr [ebp - \$... | | | 3 | 1 |
| | \$004414BD 8B4840 | | mov ecx, dword ptr [eax + \$... | | | 3 | 1 |

The panel's context (right-click) menu also has an **Open Binary File** item that lets you use it to check any executable independent of what is showing in other panels.



When **Header Information** is checked in its context menu, the panel displays a header listing the names of the source file, unit and function, as well as source file line number, function address in memory, and function length.







If the *Display source code* panel option is disabled, the main part of the panel displays assembler instructions, one per line. If it is enabled, source is displayed first and each source line becomes a node that can be expanded into its assembler instructions. For other formatting options, see *Arranging Columns...* in the Index.

The panel columns are as follows:

| Column | Description |
|--------------------------|---|
| Address | Starting address of the instruction in memory. |
| ANSI Value | Alternative translation of the binary code into ANSI-string format instead of disassembly. Quickly marks out string constants embedded in the code. |
| Hex | Hexadecimal string of bytes composing the instruction. |
| Instruction | Assembler instruction matching the Hex string. |
| Instruction No | Number of the assembler instruction from the beginning of the function. |
| Instruction Notes | An editable field for notes concerning this assembler instruction. The notes are stored in a single file used by all AQtime project. To edit, use Instruction notes from the |
| Line No | Source line number in source file. |
| Mops | Count of micro-instructions (micro-operations) in the assembler instruction. |
| Size | Instruction's size in bytes (i.e., length of its hex string). |
| Source Line | Contents of the source line for the instruction. |
| Type | Icon identifying certain instructions, such as jump , return , call , dd , dw and db . |
| Target | Next address to be executed after a jump, a call, etc. |

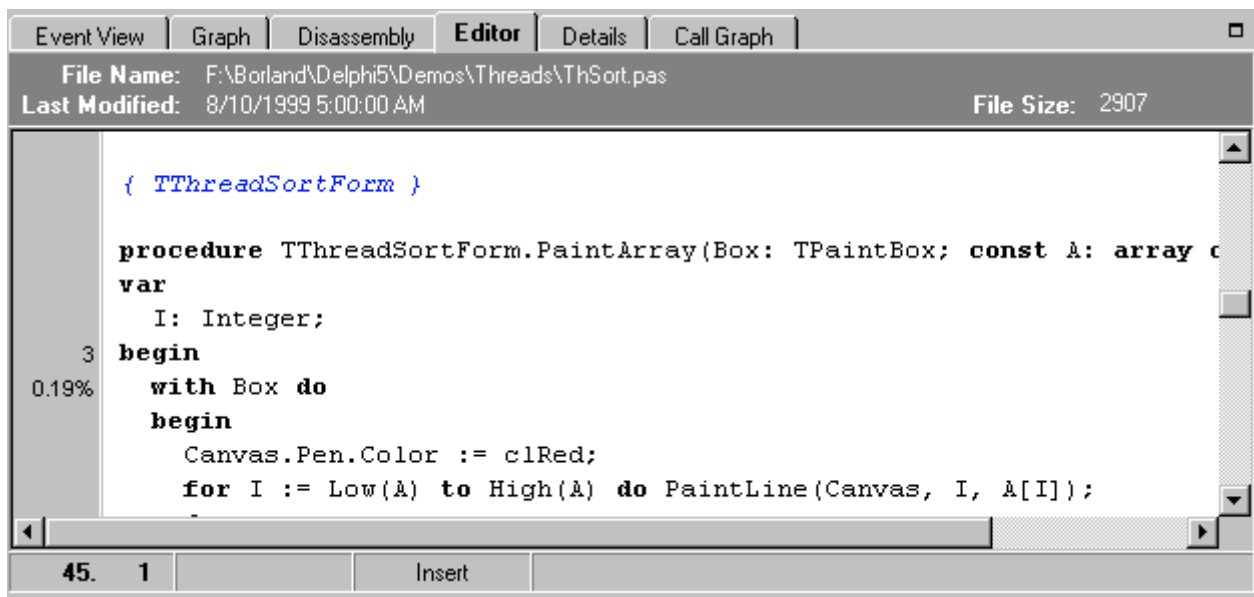
Target Procedure Name of the function at the Target address.

You can copy instructions to the clipboard by selecting them and clicking  **Copy** from the Disassembly toolbar or Disassembly context menu. You can also save them to a text file by using  **Save to File...**. The usual Ctrl and Shift commands allow for multiple selections (see *Selecting Several Records in a Panel*).


By clicking on the  jump,  return or  call icons, you can switch to the disassembly for the target function. (The  icon indicates instructions identified as data.) When using these direct jumps, the Disassembly panel tracks your movements among functions. The  **Back** and  **Forward** buttons on the Disassembly toolbar or context menu let you move back and forth among those you have previously viewed.

Editor Panel

The **Editor** panel displays source code. With most AQtime profilers, its gutter area also displays selected results for the current function or line. The panel updates to display the source for the last function double-clicked in the Report, Details, Call Graph or Event View panels, unless its source file cannot be found on the search path. (Note that some compilers, e.g. Visual Basic, can compile the executable without saving its sources to a disk file). You can also open any source file simply by dragging it onto the panel from Windows Explorer, or by using **Open...** on the panel's context (right-click) menu. **Open Project Files** on the context menu gives faster access to the source for the current project.



By default, the Editor works in read-only mode. This is controlled by the **Read Only** checkbox on its context menu. The current state (Read Only or Read/Write) is always displayed on the status bar at the bottom of the panel.

You can search the file currently displayed by using  **Find...** on the Editor's context menu or on the Editor toolbar.

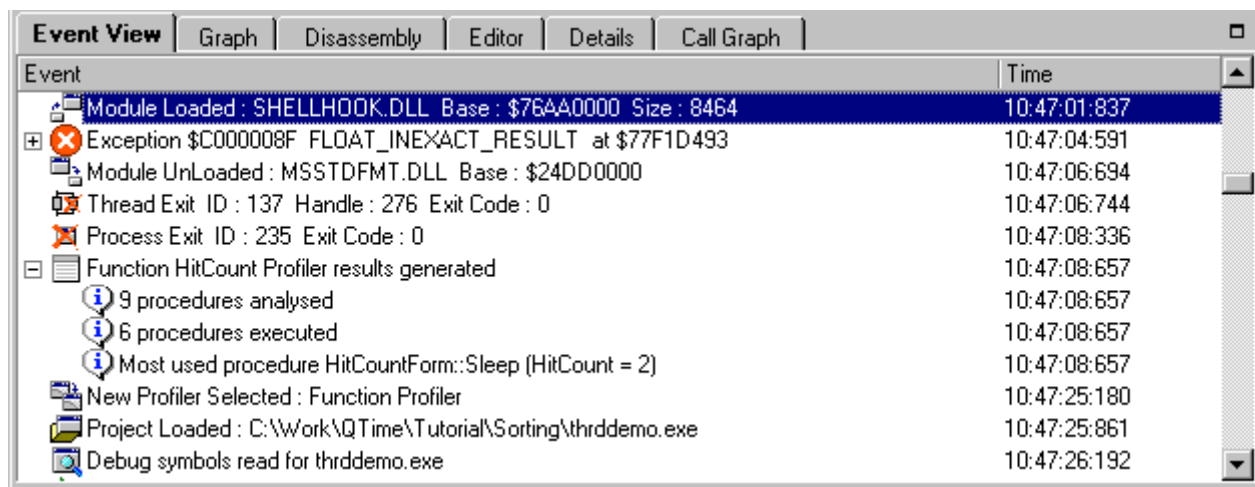
The Editor panel uses syntax color highlighting – different fonts and colors for different code elements, making them easier to distinguish and to locate. The settings are those of the development tool linked to the extension of the

file being displayed. The *File extensions for highlighting* Editor panel option specifies which extensions will be displayed according to your settings for Microsoft VC++ or VB, or for Borland Delphi or C++Builder. If an extension appears in more than one list, the priority goes from Delphi to C++Builder to VC++ to VB. Delphi's is the default color highlighting for file extensions linked to no tool.

Note that these settings set both font appearance and font language. If you have set VC++ for Scandinavian fonts, then the files you associate with it will use the VC++ highlight and Scandinavian fonts.

Event View Panel

The **Event View** panel displays events that occur within AQtime and the profiled application during profiling. It also displays brief profiling results and other profiling-related information. Each event is displayed as a parent node. Some events have child nodes that provide you with additional information about the event.





There are two columns. **Event** holds the event description, **Time**, the time of occurrence. If the *Time from application start* Event View panel option is enabled, Time is counted from the beginning of the current profile run. Else, it is the system time. See Arranging Columns... in the Index for the display options Event View shares with other panels.



















To restrict the type of event displayed (see the list below), select **Messages Filter...** from the context (right-click) menu. If the panel already has a filter, you can also click on the word *Filtered*, on the left side. Either command will bring up the Message Filter dialog in which you can exclude event types by unchecking them.

The context menu also holds **Add Comment...**, which lets you add comments as "events" into the event list. The menu also lets you copy selected events to the clipboard or to a file (tab-delimited) through its **Copy to Clipboard** and **Save to File...** items. You can use the usual Ctrl and Shift commands to multi-select events. (See *Selecting Several Records in a Panel*).

The event types displayed are as follows:

-  **Exception** Logs an exception raised by the profiled application, with exception code, name and address, and the call stack, displayed as child nodes. See Exceptions in the Event View Panel. The panel's *Max consecutive exceptions* setting defines the point beyond which it will stop logging more exceptions until the next non-exception event.
-  **Exception Stack** These items are simply successive functions in the call stack at the moment of

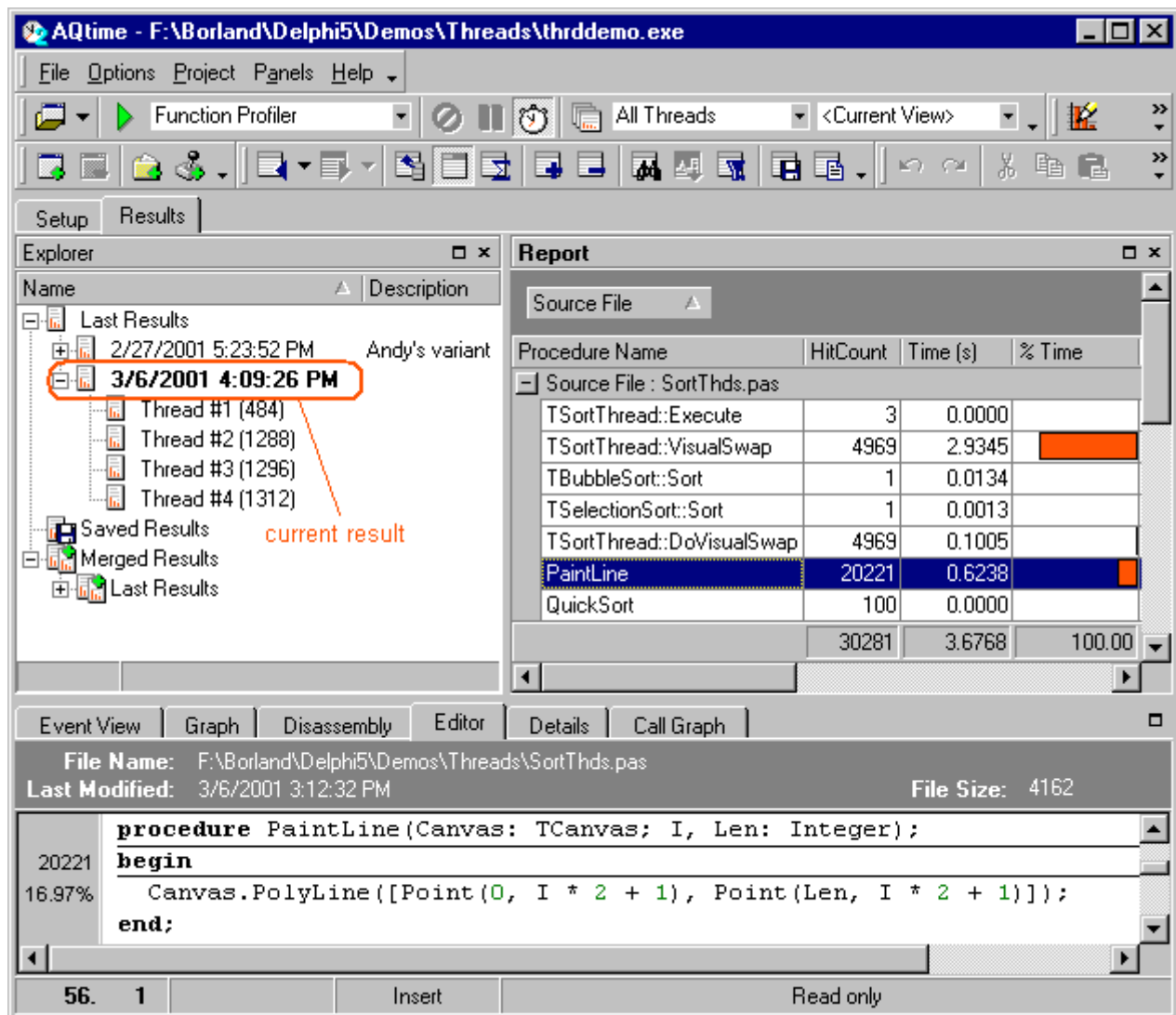
an exception. For each, the address, module name, procedure name, etc. are given. If there is enough information to retrieve them, the best substitute is displayed, for instance the module address in memory when the name is unavailable. Double-clicking on any function in the stack will update the Editor panel to its source code, and likewise the Disassembly panel to its compiled code, when you switch to either.

| | |
|---|---|
|  Project Run | Logs the start of a profile run. |
|  Project Suspend | Logs the moment a profile run is suspended from the AQtime interface. |
|  Project Resume | Logs the moment a profile run is resumed after being suspended from the AQtime interface. |
|  Change Profiler | Logs a change of profilers. |
|  Results Generated | Logs the end of result generation after a profile run, and displays a summary of the results. |
|  Debug Symbols Read | Logs the end of the debug-info reading process at the start of a profile run. |
|  Module Load | Logs the loading of a module (dll, external executable, etc.) by the main application and gives its base address in memory once loaded and its size in bytes. |
|  Module Unload | Logs the release of a module by the main application and gives its base address before unloading. |
|  Project Loaded | Logs opening a project in AQtime. |
|  String Received | Logs and shows a string message generated by AQtime or one of its profilers or plug-ins for the event list. |
|  Create Thread | Logs the creation of a secondary thread in the profiled application and gives its thread ID and handle. |
|  Exit Thread | Logs the closing of a secondary thread in the profiled application and gives its thread ID, handle and exit code. |
|  Create Process | Logs the creation of the profiled application's process and gives its process ID, handle and base address, as well as the and a handle of its primary thread. |
|  Exit Process | Logs the closing of the profiled application's process, with process ID and exit code. |
|  Leak | Warns that not all the memory or resources allocated by the application have been released at the end of its process. This event is generated by profilers that track memory and resource usage such as VCL Class, Reference Count Profilers, Memory or API Resource Check. |
|  Comment | This is the equivalent of a String Received event, but for a comment added by the user via the Add Comment dialog. |
|  Enter Procedure | Logs an entry into a function, when using the Function Trace profiler with <i>Event View</i> included in its <i>Hierarchy display location</i> setting. |
|  Leave Procedure | Logs an exit from a function, when using the Function Trace profiler with <i>Event View</i> included in its <i>Hierarchy display location</i> setting. |

Explorer Panel

AQtime's **Explorer** panel serves to manage the profiling results. It supports the following operations:

- Save the current profiler results for future use.
- Load previously saved results and display them in the Report panel.
- Delete previously saved results when they are no longer of use..
- Merge two or more result sets.
- Compare two or more result sets.
- Export results to a text or binary file.
- Import results from a text or binary file.
- Collect related result sets and organize them into folders on the Windows Explorer model.



The organization of this display can be modified, as with other panels (see Arranging Columns... in the Index). If Explorer's *Show results for all profilers* option is disabled (the default, shown above), it only displays results for the currently selected profiler. Else it displays a tree with a branch for each profiler, and in each a sub-tree of folders identical to the tree shown above. See Explorer Options.

All items inside folders are names for result sets, and they can be edited in place. The default name is simply the date and time that the results were generated. For multithreaded applications, separate results per thread are kept as sub-items of the whole-application results. Double-clicking on the main results will open the thread-result list.

There are a number of main branches to the results tree. Individual result sets can be dragged from one to the other, or click-dragged to copy them. They can be deleted by using the Del key or **Delete** on the context (right-click) menu. Or they can all be deleted by using **Clear All**. Generally, the same manipulations are possible in the Explorer tree as in Windows Explorer, but some are forbidden for obvious reasons – you cannot drag or copy into Last Results or Merged Results, for instance.

The folders are:

Last Results – The most recent results are automatically kept, up to a certain number, with each new result set expelling the oldest from the list. The number of result sets kept here is set by the *Number of recent results to keep* Explorer panel option, and is five by default.

Saved Results – This stores any result set you have moved or copied to it, or saved by using **Save Current** (not Save) from the context menu. Results are not removed from the store until you do it yourself.


Merged Results – The store of result sets obtained by *merging* (explained below).

Optional Folders – The **New Folder** item on the context menu lets you add as many main branches as you wish. All behave like Saved Results. In other words, Saved Results is the default folder, and the folders you add and name yourself let you put more organization into your store of saved results. Result sets are added by dragging of Ctrl-dragging.

Merging Results

As long as you are profiling the same build of your application, or builds where the code of interest hasn't changed, combining result sets can be a major help in getting better statistics. This is called **merging**. Also, since you can merge your results later, you are free to do shorter and simpler profile runs on separate aspects of your application.

Exactly how result sets will be merged is defined in the Merge Settings dialog which you can call through **Merge Settings...** on the Explorer context menu. Make sure you have this under control first, so that your merged results will make sense to you. Also, if you are using Borland Delphi 5, you should enable *Update procedure names to Delphi 5* in Explorer Options (and disable it if you are not using Delphi 5).

Merging itself is a two-step operation. First multi-select several result sets (even merged ones), from any folder. Use Shift-Click and Ctrl-Click to do this. Second, command the merge operation by selecting  **Merge** from the Explorer toolbar or context menu. Each merged result set stores the names of the source sets.


This can also be automated. In Explorer Options, there is an *Auto-merge* option. If this is enabled, you can also specify a *Folder name*. This is not a normal folder, but a sub-folder that will be created inside the Merged Results folder on the next profile run. Separate result sets will accumulate there as well as in Last Results, and the first will be identical to the one in Last Results. The second however will merge this first one with the results of the new profile run. The third will merge the second with the next profile run results, and so forth.

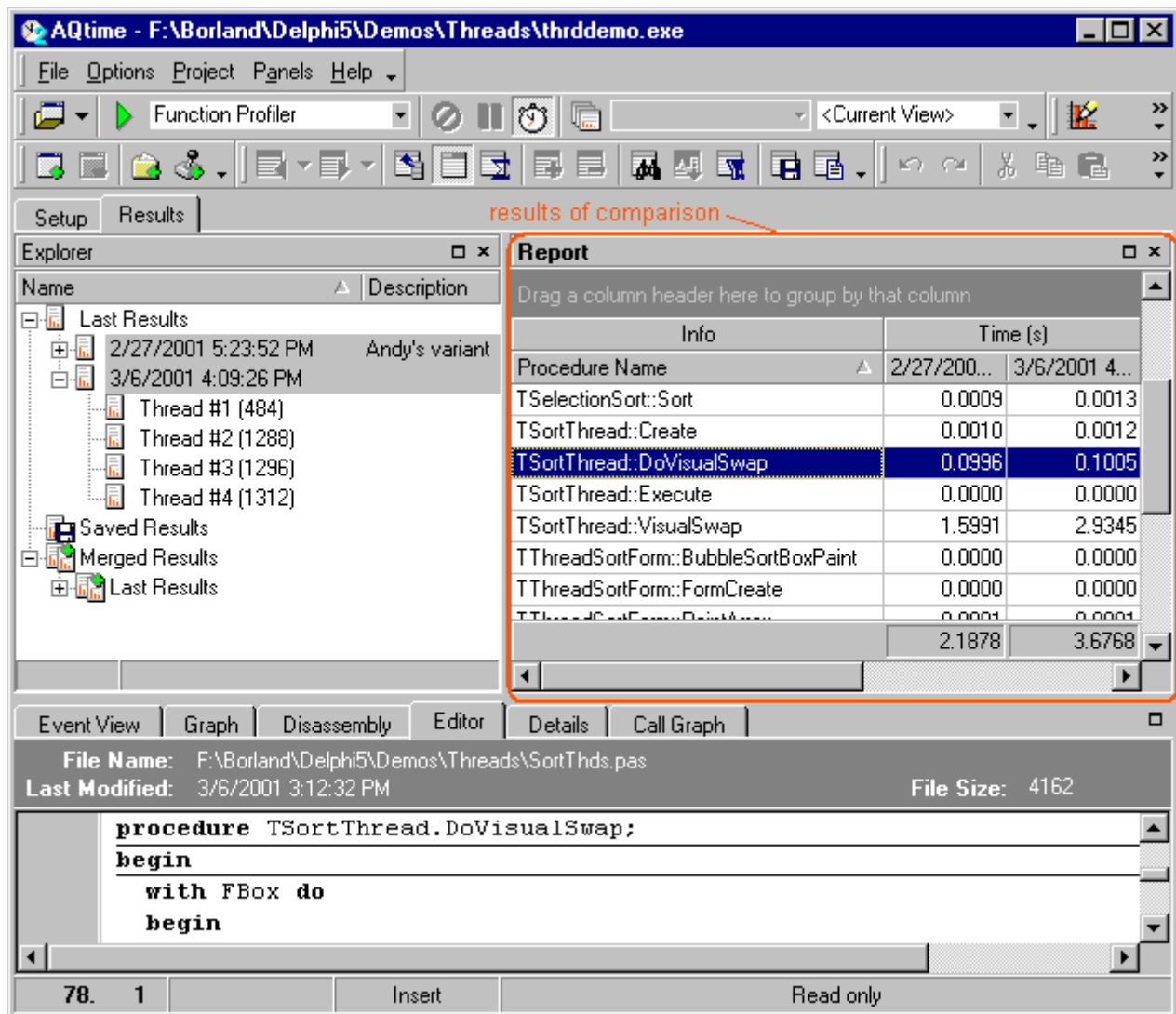
Comparing Results

The point of getting profile results is usually to improve your code. So, between different builds, it can be extremely helpful to have a simple way to **compare** result sets *on the values of interest*. This is what the Explorer panel's Compare operation allows you to do.

What values will be compared, and how the difference will be computed in the case of numeric values, defined in the Compare Settings dialog, which you can call through **Compare Settings...** on the Explorer context menu. Make sure you check this first so that the operation compares what you want and discards what is not of interest. In fact, there is an *Always set up Compare params* option in Explorer Options, which will cause the dialog to pop up whenever you command a Compare operation.

Also in Explorer Options, if you are using Borland Delphi 5 you should enable *Update procedure names to Delphi 5 i* (and disable it if you are not using Delphi 5).

To do a Compare, first multi-select several result sets from any folder. Use Shift-Click and Ctrl-Click to do this. Second, command the Compare by selecting  **Compare** from the Explorer toolbar or context menu. The Report panel will then display the result of the comparison:



The Report panel is divided into several vertical sections. The first one, **Info**, holds information that is common to all result sets and identifies one particular result line (file, function, etc.). The other sections each correspond to a value you set to be compared in Compare Settings. In each section is a column for each result set and one for the difference computed between them, if these are numeric values (the normal case).

| Info | Time (s) | Time (s) |
|-------------------------------------|---------------|---------------|
| Procedure Name | 2/27/200... | 3/6/2001 4... |
| TSelectionSort::Sort | 0.0009 | 0.0013 |
| TSortThread::Create | 0.0010 | 0.0012 |
| TSortThread::DoVisualSwap | 0.0996 | 0.1005 |
| TSortThread::Execute | 0.0000 | 0.0000 |
| TSortThread::VisualSwap | 1.5991 | 2.9345 |
| TThreadSortForm::BubbleSortBoxPaint | 0.0000 | 0.0000 |
| TThreadSortForm::FormCreate | 0.0000 | 0.0000 |
| TThreadSortForm::PaintArea | 0.0001 | 0.0001 |
| | 2.1878 | 3.6768 |

```

procedure TSortThread.DoVisualSwap;
begin
  with FBox do
  begin

```

The Report panel is divided into several vertical section. The first one, **Info**, holds information that is common to all result sets and identifies one particular result line (file, function, etc.). The other sections each correspond to a value you set to be compared in Compare Settings. In each section is a column for each result set and one for the difference computed between them, if these are numeric values (the normal case).

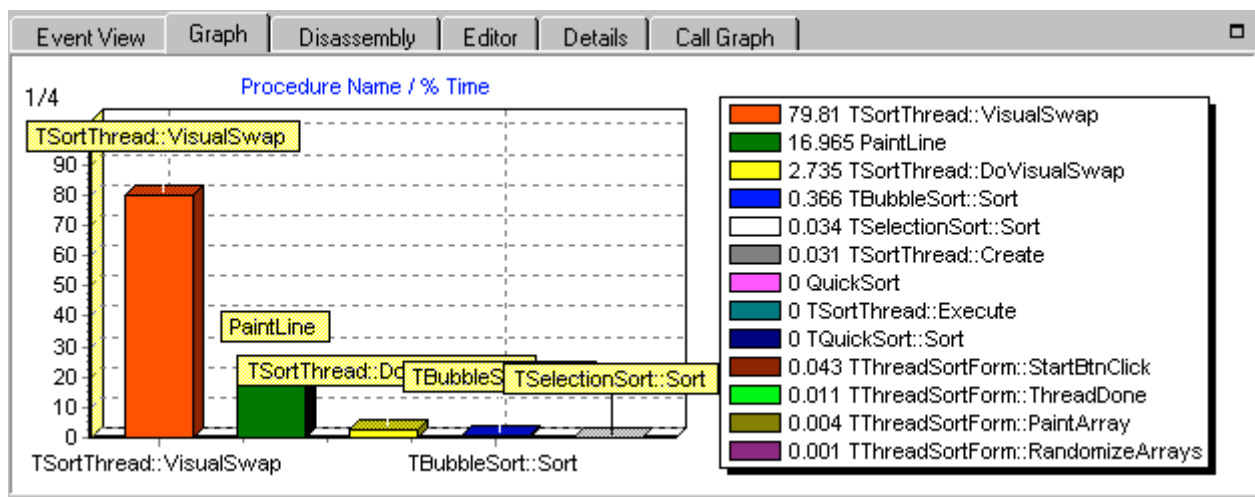
Exporting and Importing results

Result sets can be exported to a text file so that they may be included in a formal report or for some other use. This also allows transferring specific result sets from one machine to another. For more compactness in the latter use, the export format can be made binary.

Exporting any single result set is available from the context menu as **Save To File...** and importing as **Load From File...**

Graph Panel

The **Graph** panel displays profiling results from the Report panel in graphical form and offers a built-in print capability.





On the X axis (horizontal), the graph shows each line from the Report panel. For each line it shows as many graphical values as you have chosen separate values to graph. "Separate values to graph" are called **series**. For instance, the graph might show two series, Percent Time and Time in Seconds. Each profiled function would get two vertical bars, one for each value.

But note that the panel can show many other forms of graphs besides bar graphs – eleven in all, each in flat or 3D version. This is selected from the Graph context menu (right-click), **Properties | Change**. Properties in fact opens the Chart Editor, and many other aspects of the graph layout can be modified from there – colors, titles, etc. More are available from **Options...** on the context menu.

Series can be added by dragging columns in from the Report panel or by using **Add graph series** from the Graph toolbar. But for more control (esp. to remove series) use the **Series** sub menu of the context menu. The panel will automatically reject a series if it is already on the graph. (See *Graph Panel Series*.)

Refresh on the context menu will update the chart display when the Report panel has been reorganized, for instance by sorting.

Copy on the context menu lets you copy the chart to the clipboard in any of three formats, bitmap, metafile, or enhanced metafile image. Once on the clipboard, the chart can be pasted into the appropriate software (e.g. Paint for a bitmap), and modified or saved to file from there.

To print a chart with complete control, use  **Preview Graph** from the toolbar or context menu. You can print directly from there. Or, if your print parameters are already correctly set up, you can simply click on  **Print** in the context menu.

Macro Engine Panel

Macro Engine Plug-In

You have built your application, tested it and found bugs and bottlenecks. You've modified your source, and retested, repeating the exact tests you did before. You've fixed code again, and now you're going to repeat the cycle... etc., until you've actually found the solution to the application's problems. This approach to application testing is a drain on time and energy and also ensures that at some point the test sequences will vary. Robot work should be left to robots.

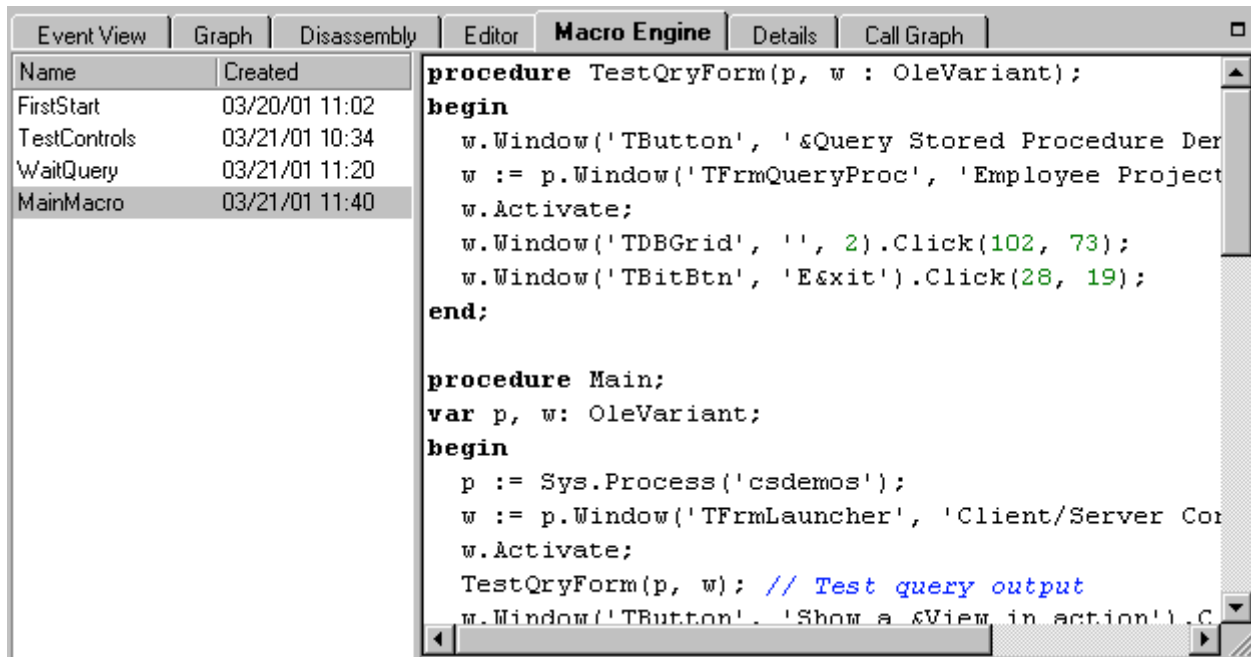
The **Macro Engine** plug-in for AQtime can record and play back any sequence of test steps (mouse clicks, key presses, etc). These events are recorded as program instructions in one of three scripting languages, at your option: DelphiScript, VBScript or JScript (see *About Macros*). These instructions make up a routine, which is called a **macro**. On the next profile run, you can execute the macro rather than repeat all test operations manually.

The Macro plug-in uses object-oriented recording and playback built on AutomatedQA's **AQtest** technology. All events are recorded as a sequence of calls to methods or properties of the same objects as AQtest uses. Macros recorded with the Macro Engine can be run in AQtest, where you may want to use AQtest's specialized facilities to test them out. On the other hand, most AQtest *scripts* will not execute as AQtime macros; the Macro Engine online supports a simple, single-purpose subset of the AQtest scripting facilities.

All recorded macros are stored in the AQtime project file (<projectName>.aqt). They can separately be saved to .mac files, and loaded back from these.

Macro Engine Panel

The Macro Engine plug-in is installed in AQtime as an additional panel. This panel includes two sections:



The left section displays a list of all recorded macros for the current test project. It holds two columns: **Description** displays the macro name, **Created**, the creation date and time. You can customize the macro list using the usual AQtime column-moving options.

The right section, **Macro Editor**, holds the source code of the macro selected in the list. The Editor includes a full range of code-editor features; see Editor Options. Remember that all macros must include a *Main* procedure, since this is what is called when you run the macro. See *About Macros*.

To run or record a macro, use the context (right-click) menu or the Macro Engine toolbar. See *Macro Recording and Playback* for details.

About Macros

A macro is a series of program instructions written in one of three scripting languages: DelphiScript, VBScript or JScript. The language for new macros is specified by one of *Macro Engine Options*.

The DelphiScript support is built into the Macro Engine plug-in. VBScript and JScript are supported by dynamic link libraries of the same name shipped with Windows or Internet Explorer. To run VBScript and JScript macros, you must have Internet Explorer 4.0 or higher. Else, you can install the Microsoft scripting components directly. The latest version of these DLLs (as a part of Windows Script Components) is available at <http://www.microsoft.com/msdownload/vbscript/scripting.asp>.

All three supported languages are limited to variables of OleVariant compatible type, and none can use pointers. Types specified in variable or function declarations are ignored. For detailed information, see

- DelphiScript *DelphiScript Description* in on-line help
- JScript <http://www.microsoft.com/jscript>
- VBScript <http://www.microsoft.com/vbscript>

Program instructions within a macro can be divided into several procedures. When recording starts, the Macro Engine creates a new procedure called *Main* and then records to it all events that occur

- in AQtime,


- in the profiled application and
- in the system.


Later you can add other procedures manually using Macro Editor, split the recorded code among them, or add your own code. Procedures can call each other within the same macro. Note that each macro must have its *Main* procedure, as this is what is called when you start playback.

All events are recorded as a sequence of calls to methods or properties of certain objects. The objects, methods and properties all belong to a subset of those in **AQtest**. Macros recorded with the Macro Engine can be run in AQtest, where you may want to use AQtest's specialized facilities to test them out. You can copy macros from the Macro panel to AQtest and run and debug them as scripts there. But most AQtest scripts go beyond the capacities of the Macro Engine, and will not play back on it. For detailed information on objects, methods and properties used by the Macro Engine plug-in, see the AQtest Documentation. Some basic principles are mentioned in *Window and Process Recognition*.


Macro Recording and Playback

Before recording or playing back a macro, review the settings in the *Macro Engine Options* dialog and make sure they are as you expect.

To record a new macro, select  **Record New Macro** from the context (right-click) menu or from the Macro Engine toolbar. To rewrite an existing macro, select it in the Macro Engine panel and then choose **Replace Existing Macro** from the context menu.

Choose  **Stop** from the context menu or from the toolbar to stop recording. Once the recording is stopped, you can switch to the Macro Editor panel and edit the name and contents of the macro. Note that the Macro Engine always records instructions into the *Main* procedure. Later, you can create new procedures and organize source code among them.

All recorded macros are stored in the .aqt file for the AQtime project. You can also save and load a macro from an external file (with a .mac extension): Just select **Save to File...** or **Load From File...** from the Macro Engine context menu.

To play a macro, select it in the Macro Engine panel and then choose  **Run** from the context menu. To abort the playback, press Stop. Note that each macro must have a *Main* procedure since Macro Engine calls this procedure when you press the Run button.

It may be more convenient to use keyboard **shortcuts** for Macro Engine commands instead of context menu items. For instance, to cancel the recording or playback you can simply press the Stop shortcut, whereas using the context menu items you'll have to switch to AQtime's Macro panel and select Stop from the context menu. Note that Macro Engine's shortcuts are used as the system global ones; they work in all applications, whether or not the Macro plug-in is in record or playback mode. To change shortcuts, use the *Macro Engine Options* dialog.

Window and Process Recognition

Since the Macro Engine is built on the **AQtest** technology, it uses the same objects and object hierarchy as AQtest. Processes, windows, controls, etc. are represented as program objects in macro code. According to the object hierarchy, processes are "children" of the system, windows are "children" of processes or other windows, etc. To get the complete idea, see the *AQtest documentation*. Here is some essential information on how to work with window and process objects in source code.

The `Process(...)` and `Window(...)` function methods return proper program objects for the desired process or window. The Macro Engine, as well as AQtest, uses several attributes meant to distinguish one window from others currently open in the system. The window handle is meaningless, since it is changed from one run to the next. That is why the Macro Engine uses more stable attributes: Class name, caption and index (instance

number), i.e. `Window(Classname, Caption, Index)`. Both the class name and the caption can hold wildcards (* and ?).

Nevertheless, no window attribute is guaranteed to be stable. On the contrary, applications can change window attributes frequently. For instance, MDI applications often change the parent MDI window caption according to the active child window. To warn you away from relying on such a caption, Macro Engine automatically records the string "*" for any caption that includes a hyphen.

In some applications, the window class names can be changed from one execution to the next. Typically, this applies to MFC windows. For instance, in the class name "Afx:400000:b:10008:6:1027b" the last digits can change from run to run. To solve the problem, specify wildcards in the window class name. In our example, you might use "Afx:400000:b:*" instead of the full class name.

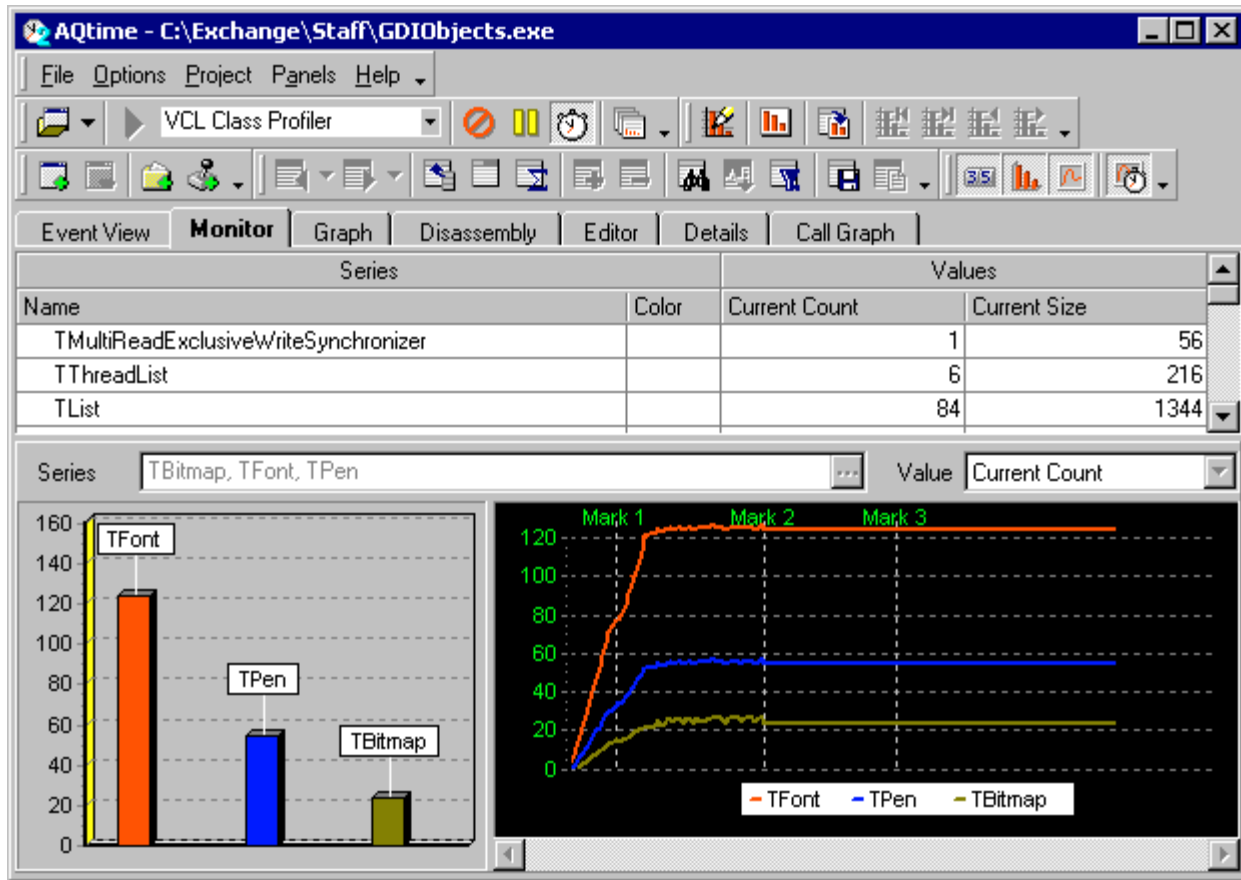
The `Process(...)` function uses two parameters: The name of the executable that launches the process (extension is excluded) and the process index, that is, `Process(ProcessName, ProcessIndex)`. If the tested application is run and Macro Engine cannot find it, make sure the process name and index are specified correctly. Note that unlike the window caption and class name parameters, the process name does not support wildcards.

Monitor Panel

The **Real-Time Monitor** is a tool aimed at monitoring real-time resource usage and reporting the results in different ways: grid records, charts and graphs. The contents of the posted reports are determined by the profiler used and by the application being profiled. Currently, Real-Time Monitor works with the following profilers:

- ATL Reference Count Profiler
- VCL Class Profiler
- VCL Reference Count Profiler
- Memory and API Resource Check

The Real-Time Monitor is organized as an AQttime visual plug-in (a panel) with a context menu (right-click) and a toolbar which lets you tune what to display in the panel and how to do that. Here is an example of the Real-Time Monitor report for the VCL Class profiler:



The Real-Time Monitor can display result data using three views:

- Counter
- Graph
- Histogram









They can be used simultaneously in any desired combination. Splitters that separate them are movable; thus you can change the view sizes. To hide or show a view, simply select an appropriate item in the Real Time Monitor toolbar or the context menu.

Each profiler holds a constant list of values to be calculated for each series. These values are columns in the **Values** section of the **Counter** view. There is no need to start the project in order to obtain this list: You simply select the necessary profiler. Then you can hide some of these columns or display them again. See the *Values* dialog in on-line help. One of the columns, displayed in the Counter view, is used to form the results on the Graph and Histogram views: You select it from the **Values** dropdown list of the Real-Time Monitor panel.

A series list is set in a different way. It is modified during the profiling and therefore it is empty before executing the application (though all series will be visible after you have started the application for the first time). After this, the Real-Time Monitor saves all information on the used series in the current project. Before starting the application for the second time, you can specify which series to display. To do this, use the *Series* dialog. To choose series for two other views, use the *Select Series Series* dialog and the results of this selection will be displayed in the **Series** field of the Real-Time Monitor panel.

Counter View

In the Counter view, all the data are displayed as table records called series. In comparison with the other views that graphically illustrate the results and give their common image, the Counter view is suitable for precise analysis of the results. When Counter is not alone on the panel, this view occupies the topmost part of it.

| Series | | Values | |
|----------------|---|---------------|--------------|
| Name | Color | Current Count | Current Size |
| TMenuItemStack |  | 1 | 8 |
| TMouse |  | 1 | 40 |
| TScreen |  | 1 | 152 |
| TMonitor |  | 1 | 12 |
| TFont |  | 11 | 396 |
| TApplication |  | 1 | 272 |
| TIcon |  | 2 | 88 |
| TIconImage |  | 2 | 48 |

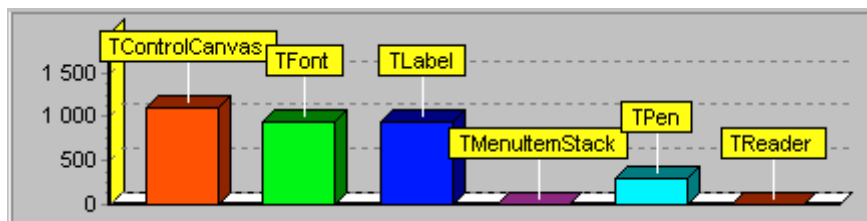
The table is divided into two sections. The first section (**Series**) displays the names of series where you will get the report (the **Name** column) as well as the color chosen for the specific series (the **Color** column). You can specify which series to be displayed in this section for the current profiler using the *Series* dialog. Through this dialog you can also assign colors for these series to distinguish them in the **Graph** view, although only the colors of the series, displayed in Graph, will be displayed in the grid of the Counter view.

The second section (**Values**) holds columns to display numerical values for the chosen series. Each supported profiler has its own list of columns for this section. Through the *Values* dialog you can assign the columns to be displayed.

The *Refresh Interval / Counter* option sets the time interval, after which AQtime updates the Counter view. If this value is 0, Counter is refreshed immediately after a series value has been modified. See *Monitor Options*.

Graph View

This view displays series within a chart and visually demonstrates the difference between series values. When Graph is accompanied with all other views on the panel, this view occupies the lower left part of it.



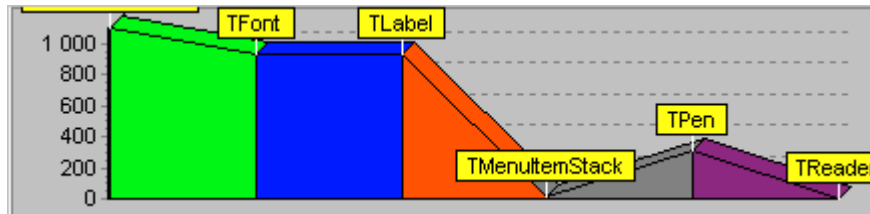
The Graph view represents the results of a sole value: It is the column selected in the **Value** dropdown list. This control holds the names of the columns checked in the *Values* dialog. The **Series** field determines the series for which the results are displayed in the Graph view. It lists the names of the series that are checked in the Select Series dialog. Those can be the series provided by the profiler as well as the sum of all series and the sum of all visible series.

The *Refresh Interval / Graph* option sets the time interval, after which AQtime updates the Graph view. See *Monitor Options*.

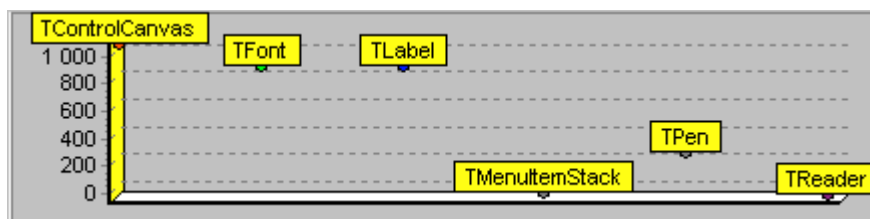
The Graph view applies definite colors to draw the chosen series. Use the Series dialog to set these colors for the ordinary series or the *Options* dialog to set the colors for summarized series (**Sum of All Series** and **Sum of All Visible Series**).

After termination of profiling the Graph view contents can be scaled. Diagrams within Graph can be displayed in one of several supported styles (lines, points, areas, etc.). You can choose an appropriate style if you select **Graph Style** from the context menu:

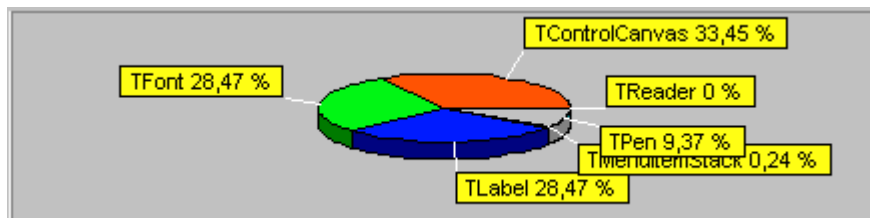
Area



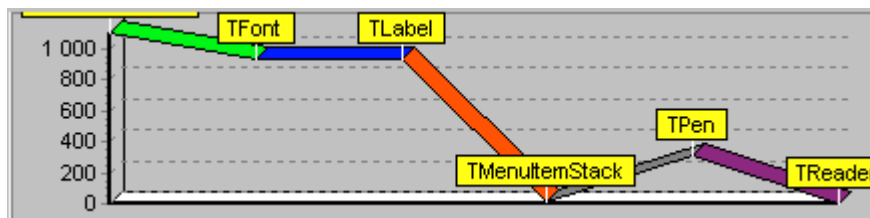
Point



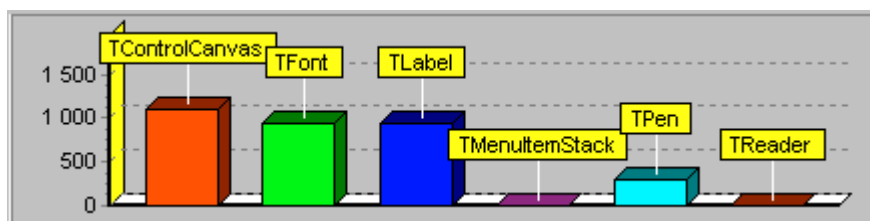
Pie



Line

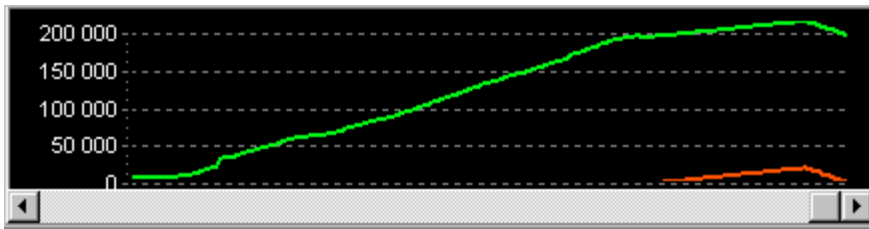


Graph



Histogram View

The **Histogram** view displays series within a histogram during the application functioning. This allows you to compare certain series by their values at definite moments. Multiple display styles are available when using this view. When Histogram is accompanied with all other views on the panel, this view occupies the lower right part of it.

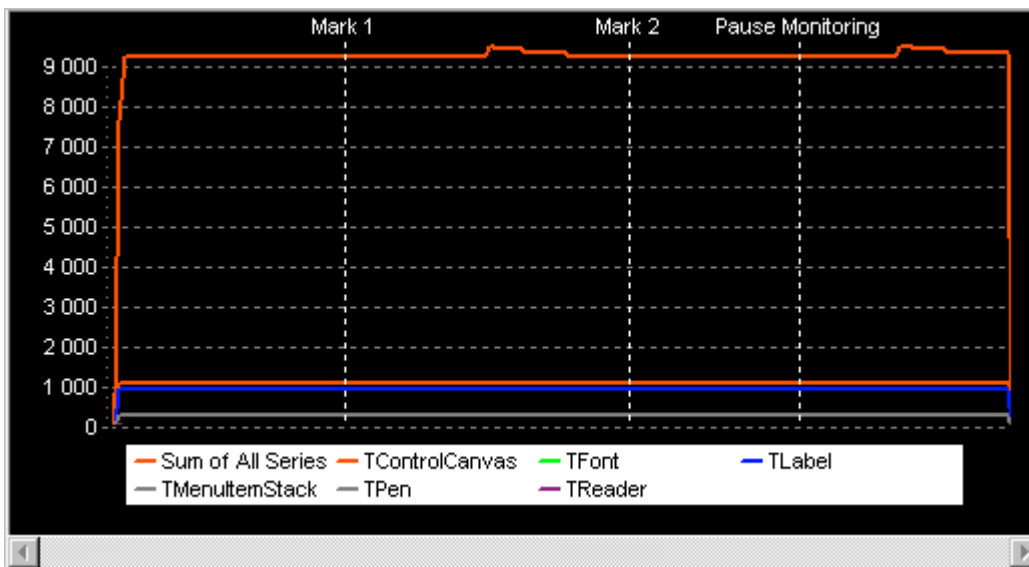


The **Histogram** view represents the results of a sole value: It is the column selected in the **Value** dropdown list. This control holds the names of the columns checked in the Values dialog. The **Series** field determines the series for which the results are displayed in the Histogram view. It lists the names of the series that are checked in the Select Series Series dialog. Those can be the series provided by the profiler as well as the sum of all series and the sum of all visible series.

The *Refresh Interval / Histogram* option sets the time interval, after which AQtime updates the Histogram view. See Monitor Options.

The Histogram view applies definite colors to draw the chosen series. Use the Series dialog to set these colors for the ordinary series or the Options dialog to set the colors for summarized series (**Sum of All Series** and **Sum of All Visible Series**).

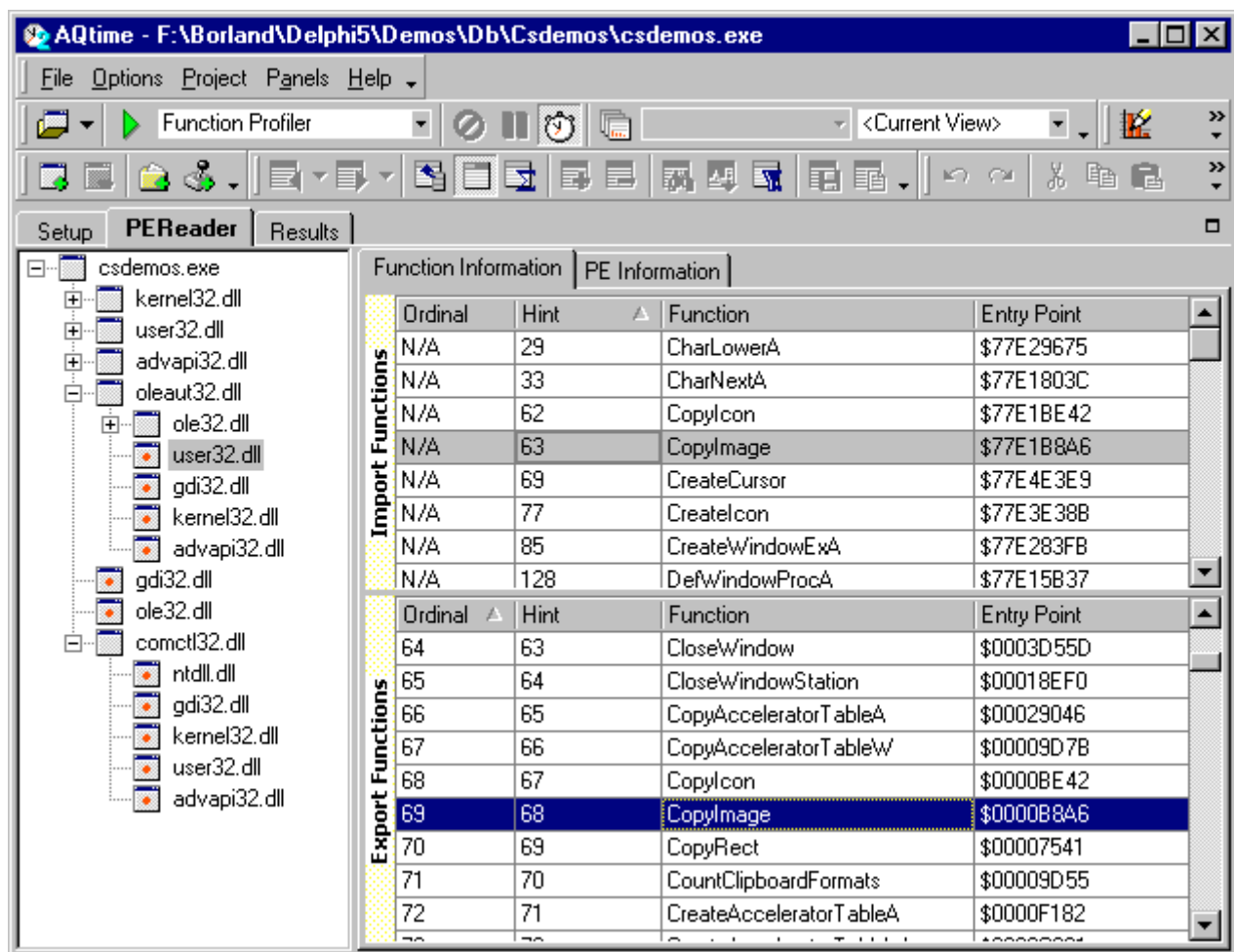
When executing a project, you can place marks within the histogram to quickly isolate the desired sections of the graph (simply click within the histogram where you want).



After termination of profiling the Histogram view contents can be scrolled and scaled.

PEReader Panel

The **PEReader** plug-in is intended for analyzing modules' relationship in the profiled application. This plug-in is installed as an additional AQtime panel (see Installation Notes). Upon loading a project, PEReader analyses modules statically linked to the application and displays detailed information about these modules, imported functions, modules' headers, etc.



The PEReader panel is divided into several sections -

- Modules Hierarchy** Displays the hierarchy of modules used by an application.
- Function Information** Holds tables of imported and exported functions.
- PE Information** Displays information about headers and sections of the module selected in the Modules Hierarchy pane.

Contrary to other AQtime analysis means, PEReader does not require the application to be compiled with the debugger information. It simply analyzes the application code. It helps you to -

- Determine statically linked modules required for the application running.
- Determine defective files.
- Determine what functions each module imports and exports.
- Examine detailed information about functions used by the application (entry points, function addresses in a module).
- Find correspondence between imported and exported functions quickly.


- Examine detailed information about modules used by the application (operating system version, module version, image file type, debug info existence, entry point, image base address, processor type, etc).
- Determine whether a function belongs to a module, etc.

Specified URL cannot be opened. A web browser is not set on your computer properly. To open a URL, launch your web browser and type the address manually.

Modules Hierarchy Panel

The **Modules Hierarchy** pane of the PedReader panel displays the hierarchy of all modules statically linked to the application. It is a convenient instrument to explore correlation between different modules, used by an application.

When AQtime loads a project, PedReader scans all application modules recursively beginning with the main one. If a module, say it is a dynamic link library, imports some functions from another dll, PedReader analyzes the latter dll and displays it as a child of the "superior" dll in the Modules Hierarchy pane. The recursion continues until all used modules are processed. The results of analysis are displayed in the Function Information and PE Information panes.

Normally, modules use each other. If during the recursion PedReader meets a module that has already been reviewed, it does not check its child modules. PedReader marks the duplicated modules using the  sign and displays them without "children" in the Modules Hierarchy pane. The sign means that the module has been analyzed somewhere before. To view child modules for the duplicated one, simply double click it in Modules Hierarchy or select **Find First Occurrence** from the context menu.

Besides the "duplication" mark, modules can be displayed with one of the following signs:



Ordinary module.



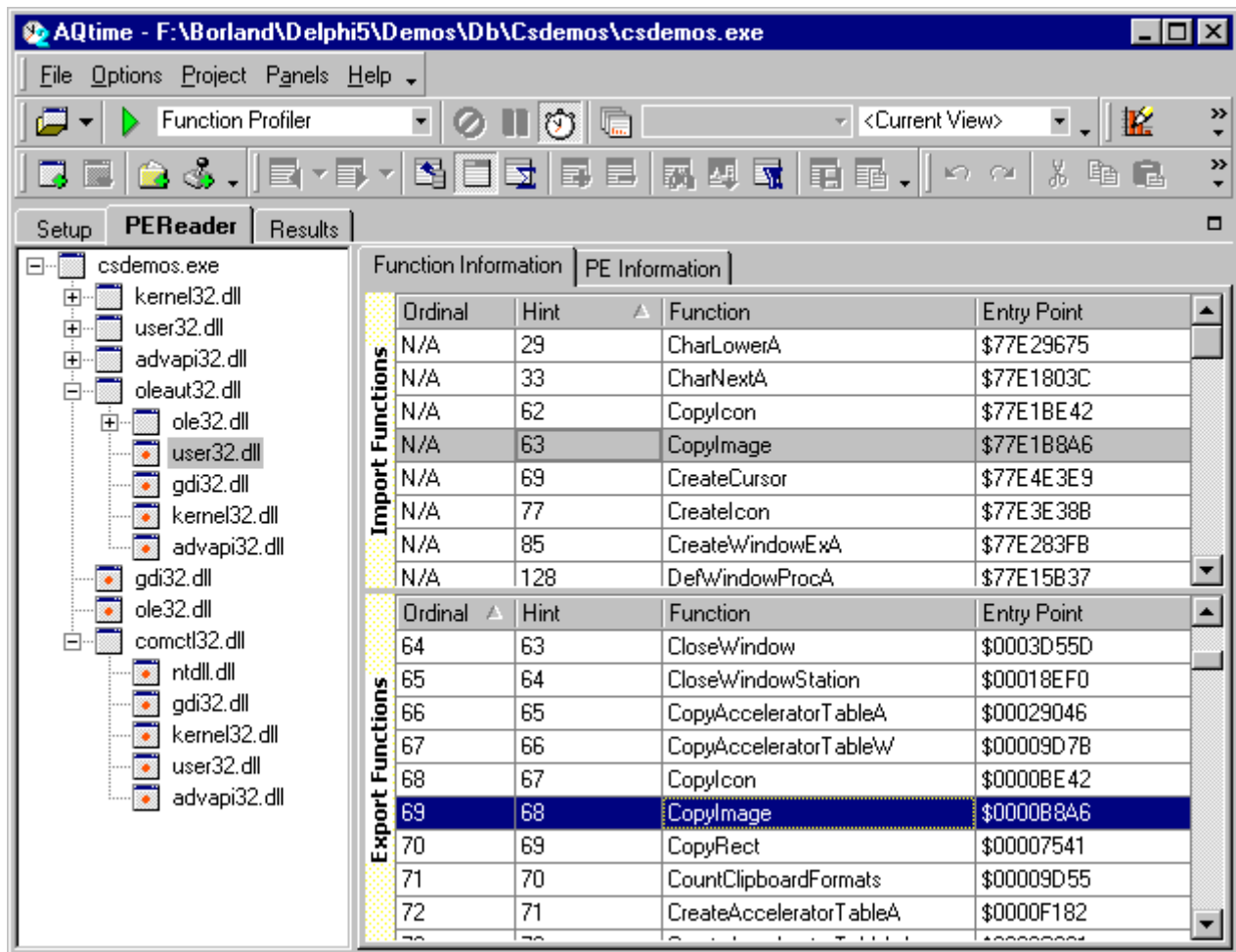
Absent module. PedReader displays this icon when it can not locate a module used by the application. To obtain the list of all modules necessary for the application running, call the Statically Linked Modules dialog (select **Show Modules List...** from the context menu).



Defective module. PedReader reports that the module is defective if it can not be executed by certain reasons.

Function Information Panel

The **Function Information** pane of the PedReader panel displays two lists of functions: The lower one holds functions exported by the module currently selected in the Modules Hierarchy pane; The upper list holds functions called by the "parent" module from this "child".



Function Information includes the following columns -

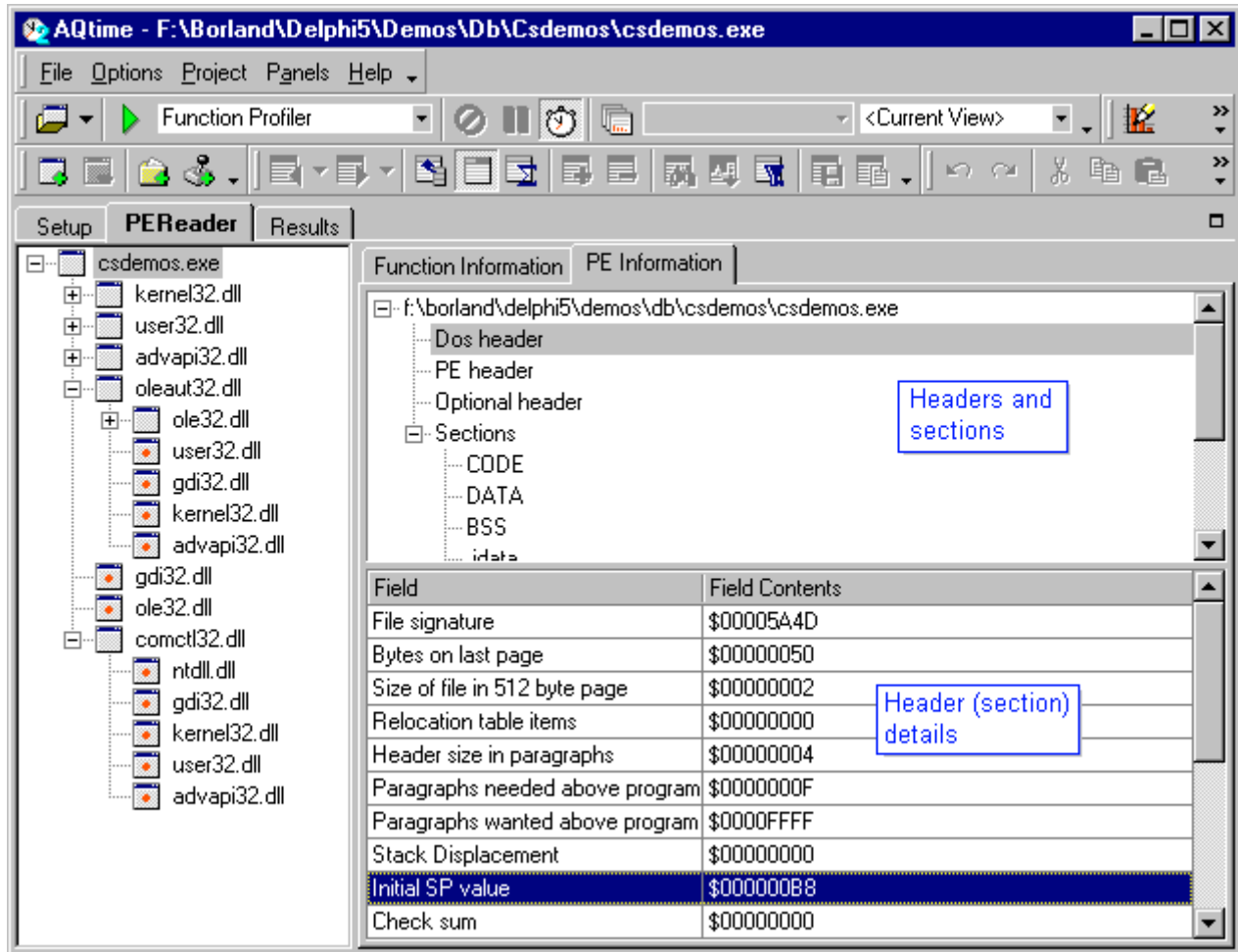
- Ordinal** Holds the function ordinal number if the function is imported by its ordinal number. "N/A" means the function is imported by name.
- Hint** Hint value of a function. This value is a function index in the array of functions exported by the dll. The system uses this value for rapid search for a function in a module.
- Function** Holds the function name if a function is imported by name. "N/A" means that the function is imported by Ordinal number.
- For C++ functions the Function Table displays unmangled names to make the information readable. Review Mangled Names for more information on mangled function names and mangling related problems.
- Entry point** Function address in the module. Usually, this column displays "No bound" for imported functions, i.e. the entry point is unknown until the module is loaded. If the address is specified, the module has been linked via the BIND program.

You can arrange the Function Information pane in the usual way (see Arranging Columns... in the Index).

To find an exported function that corresponds to an imported one, select the imported function in the upper list and then choose **Highlight Matching Function** from the context menu.

PE Information Panel

This panel displays detailed information about headers and sections of the module (a dll or exe file) selected in the **Modules Hierarchy** pane.



Headers

The header of a module consists of three parts (or three headers). They hold detailed information about a module.

- DOS Header* The header that existed in all DOS executable applications plus the field that indicates the offset of *PE Header*.
- PE Header* Holds information about the processor type, the number of application sections, and the date of file creation and file attributes.
- Optional Header* Holds specific information used by the operating system, e.g. the version number of the required operating system, control sum, image base address, etc.

For more information about the structure and contents of PE Headers, see *MSDN Library*. Note that you can access MSDN on Web - <http://www.msdn.microsoft.com>.

Sections

Sections are “segments” of code or data within an executable. In general, a file can include any section with an arbitrary name and purpose. But some sections, e.g. *debug* or *rsrc*, have essential meaning. For detailed information see *MSDN Library*. Note that Windows NT limits the number of sections till 96.



For each section, PEReader displays the following information:

- Virtual address of a section in the process address space.
- Relative size of a section body.
- The offset of the section body in a file.
- Section attributes.

Report Panel

By default, the Report and Explorer panels are located on the same tab page, titled **Results**. The **Report** panel is the basic display for profiling results. Specific elements can be selected in Report to define what will be displayed in turn in other panels: Disassembly, Editor, Graph, Call Graph and Details (see AQttime Panels).

The contents of the Report panel depend on the profiler used to generate the results on display. (These are normally those of the last run, but they can also be retrieved from previous runs through the Explorer panel.) To get help on the profiler which generated these results, press Ctrl-F1 or choose **Help On Selected Profiler** from the Report context menu.

Depending on the selected profiler, each row in the Report panel gives results for one profiled function, line, class, interface or file. As you shift from one line to another and check the ensuing details in other panels, your movements are tracked, so that you can retrace your steps back and forth using the  **Back** and  **Forward** buttons on the Report toolbar.

For numerical columns, the footer (the last grid row) lists the sum of all values in the column. In other words, the footer row displays totals for the items displayed above it.

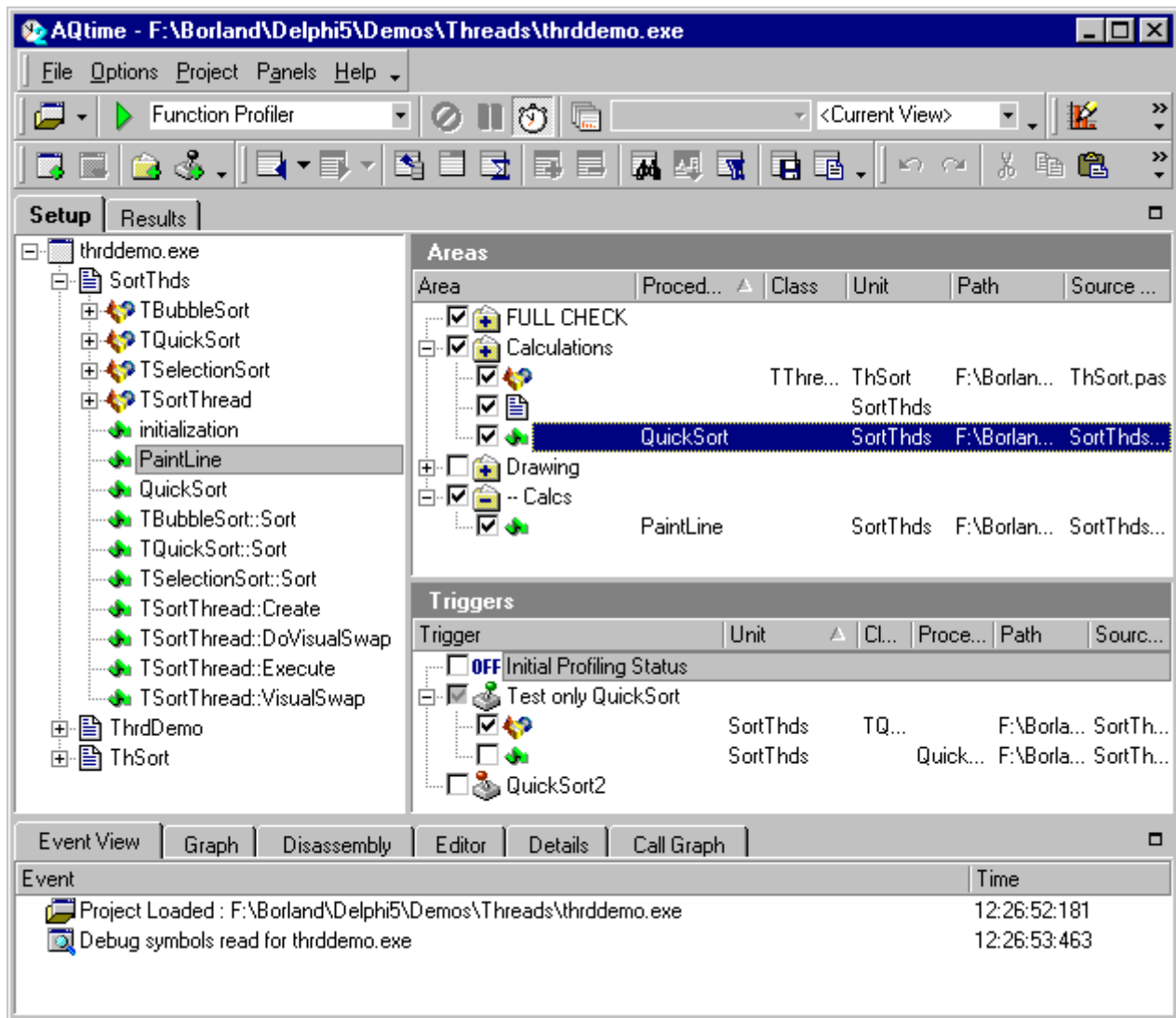
You can arrange the Report panel the same way you can organize other AQttime panels. Besides supporting these standard adjustments, the Report panel lets you:

- Force all columns to display within the visible width, by selecting **Adjust Column Width** from the context menu.
- Change the font color for a column, its text alignment, its format string, etc., by selecting **Format Columns...** from the context menu, which will call the Format Columns dialog.
- Alternatively, change text alignment in a column by selecting **Alignment** from the column header's context menu. See Column Format.
- Change data display format for a column (*Value*, *Graph Bar* and *Percent*) by selecting **Display Format...** from the column header's context menu. See *Displaying Results* in on-line help.
- Find records (lines shown) by some key value, by using the Find dialog or the **Incremental Search** mechanism (see *Searching Results*).
- Select records to display by some key value. See *Filtering Results*.
- Apply pre-defined Views to instantly get a combination of filter and display settings.
- Group results into a subtree when they share one or several common key values by one or more Report columns. The Set Summary Field dialog will let you define how values are calculated for display in the

group's top (summary) line. The usual sort-on-column feature will work on the summary line to sort entire groups.


Setup Panel

The **Setup** panel is your tool for defining what portions of code to profile, and when. See Controlling What To Profile. The panel consists of three panes. Here is a sample:





Modules pane

Modules is the pane to the left of the panel. It displays a list of all executables (exe, dll, ocx, etc) available for profiling, in treelike format. Click on a module to view all procedures, classes and units within it. Select **View by Class**, **View by Unit** and **View by Path/File** from the context (right-click) menu to arrange information within the list.

To add a module to the project, select  **Add Module...** from the Setup toolbar or context menu. The executable on which the project was first opened remains the "main" executable. AQtime launches it when it starts a profile run. Other modules will be loaded by this one as it runs (or possibly not loaded).

Alternatively, you can drag executables into the Modules pane from Windows Explorer. If they are dragged using the left button, they open a new project and become its main executable. If dragged with the right button, a dialog will pop up asking if you want to open a new project or add the executable as a module to the current project.


You can quickly locate a function in the Modules pane without opening each module, by using  **Find Function...** on the context menu to call the Find Function dialog, or simply  **Find Next**.

Areas pane

The **Areas** pane is on the right side, and at the top. It defines and keeps the list of profiling areas. Areas group code elements to be profiled (see *Defining Areas To Profile*). For an element to be profiled in a given run, it must be checked and its area must be checked also. In this way, Areas do not just define code to profile, they keep definitions on hand for later use. There are also Excluding Areas, which subtract elements from larger blocks to profile (e.g. one method from a class).

The pane displays Areas, and each Area can be opened to list its elements. The columns serve to fully specify the element being included or excluded on that line. For Areas themselves, they're empty – Areas only have the name you give them. You can arrange the columns through the usual means.

There is one preset Including Area, FULL CHECK, which you cannot change nor remove, and which includes everything in the application. Otherwise, Areas are like folders holding elements. Including Areas are shown with a + on the folder icon, and Excluding Areas with a -.

You have first to add an Area using  **Add Area...** from the Setup toolbar or context menu. The dialog lets you set the name for the Area, and whether it will be Including (default) or Excluding. You can change both settings later by using **Edit Area...** on the context menu. This also has **Remove**.

Once you have an Area defined you can use the appropriate commands on the context menu to call up the Add Units, Add Procedures and Add Classes dialogs. Or you can drag in elements from the Modules pane. You can also drag elements back out to the Modules pane, but using **Remove** is simpler. A given element may belong to as many Areas as you wish. If it is checked in an Excluding area, however, this will override all checks in Including areas.

If you check for inclusion some functions, but do not check the functions they call, then the execution time spent on those calls will be counted as if it belonged to the caller function (see *Function Profiling Restriction*). When you need to trace out exactly where the time goes, make sure that the child calls get profiled along with their callers. Triggers, described below, are an excellent tool for that.

Triggers pane

The triggers pane is to the lower right of the Setup panel. Triggers are organized in a fashion very similar to that of Areas, but their purpose is different and they apply only to the three function profilers, Function HitCount, Function Trace and Function Profiler. There are On Triggers and Off Triggers. In an On Trigger, whenever execution enters a checked element (function, class, unit), profiling is enabled for that thread. When execution leaves the element, profiling is returned to its former state. Vice-versa for Off Triggers. See *Using Triggers*.


Note that profiling being enabled does not mean it actually operates. It means it is allowed to operate on the Areas checked in the Areas pane. The only exception is that a function that is itself an On Trigger always gets profiled if checked, independent of any Area it may belong to. See *Controlling What To Profile*. However, you may simply check FULL CHECK in the Areas pane, and then leave profiling control to Triggers.

There are two predefined Triggers, Initial Profiling Status For Starting Thread and similarly for New Threads. They are checked by default. When they are checked, profiling is always enabled unless an Off Trigger is under execution. When they are unchecked, profiling is always disabled unless an On Trigger is under execution. Leaving these checked means controlling profiling primarily through Areas (which is always the case of course with all but the three function profilers). To give primary control to Triggers, uncheck the two predefined ones.

To create a new trigger folder, select **Add trigger...** from the popup menu. This will call the Add Trigger dialog, which allows you to specify the trigger name, type (on/off) and some other attributes (see *Setting Up Triggers*). Once you have a folder, you add elements to it in the same way you would add them to an Area.

Panels How-To

Adding and Removing Columns in AQtime Panels

Most AQtime panels do not display all available columns by default. In each panel, the context menu (right-click) offers a  **Field Chooser...** item, which opens the **Column Selection** dialog. To add a column, drag it from the dialog to the panel. To remove a column from the panel, drag it from the panel to the dialog.


You can also remove any column in the Report panel by right-clicking its header and selecting **Remove This Column**.

Column Format

There are many things you can change about the columns in the Report, Details and Disassembly panels.

First, in the header, you can drag the column dividers to change widths, and you can drag the column headers themselves to re-order columns.

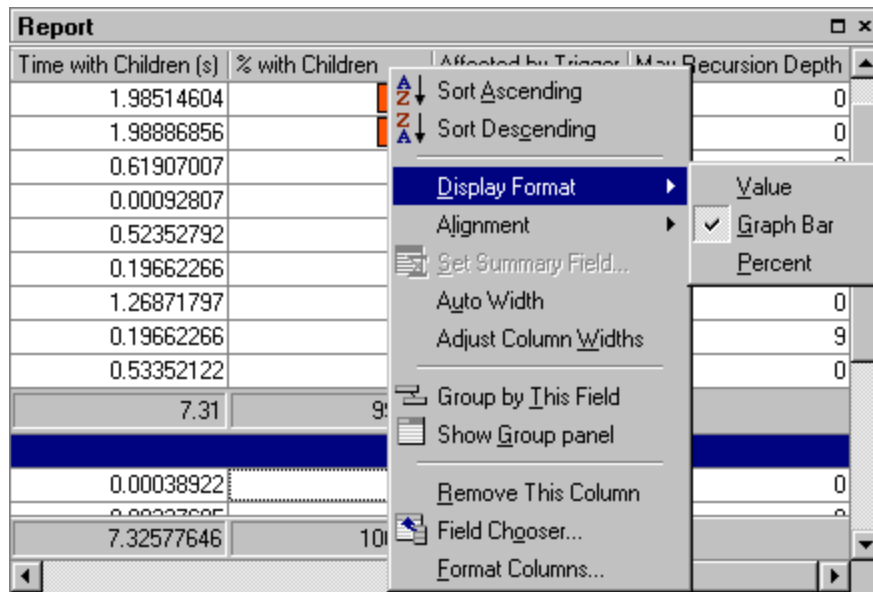
Then, still on the header, you can open the context (right-click menu). See Report Panel Context Menus. There you will find several items unrelated to column format, but also:

- **Display Format:** This isn't "format" in the same sense as Column Format. Value, percent or graph. See *Displaying Results in the Report and Details Panels*.
- **Alignment, Auto Width and Remove This Column:** Obvious meanings.
- **Adjust Column Width** (when Auto Width is off): Forces all available columns into the onscreen table. Note that by default the panels do not start out displaying all available columns.
-  **Field Chooser...** opens a list of fields (columns) from which you can add to the displayed columns by dragging out new fields, or remove displayed columns by dragging their headers into the list. See Adding and Removing Columns to/from AQtime Panels.
- **Format Columns...** opens the Format Columns Dialog, where you can set alignment, select columns to display, change their captions, etc.

Displaying Results in the Report, Details and Disassembly Panels

In the Report, Details and Disassembly panels each column holding results can display them in one of three ways, at your option: by percentage, by value, or as a bar graph. A bar graph is great for visual comparisons.

Display style is set by right-clicking on the header for the column and selecting **Display Format** from the context menu. Inapplicable display styles are disabled.



Graph Panel Series

To **modify** the properties of an existing series on the Graph Panel, select **Series | Modify** from the context menu (right-click) or choose **Properties...**, which will bring up the Chart Properties dialog.

To **add** a series, select **Series | Add Series...**, which will bring up the Add Series dialog.

To **remove** a series, use **Series | Delete**.

You can also drag any column header from the Report panel to the Graph panel. The panel will identify a string column as a **Label field**, and an integer or float column as a **Value field**. If you simply drag over to Graph chart, you will replace the existing Label or Value field (however you cannot replace a Value field with itself).

If you Ctrl-drag a Value field (drag with Ctrl key held down), you will add a new column to the chart. You cannot do this with strings, since the chart cannot have two series of Labels.

Selecting Several Records in a Panel

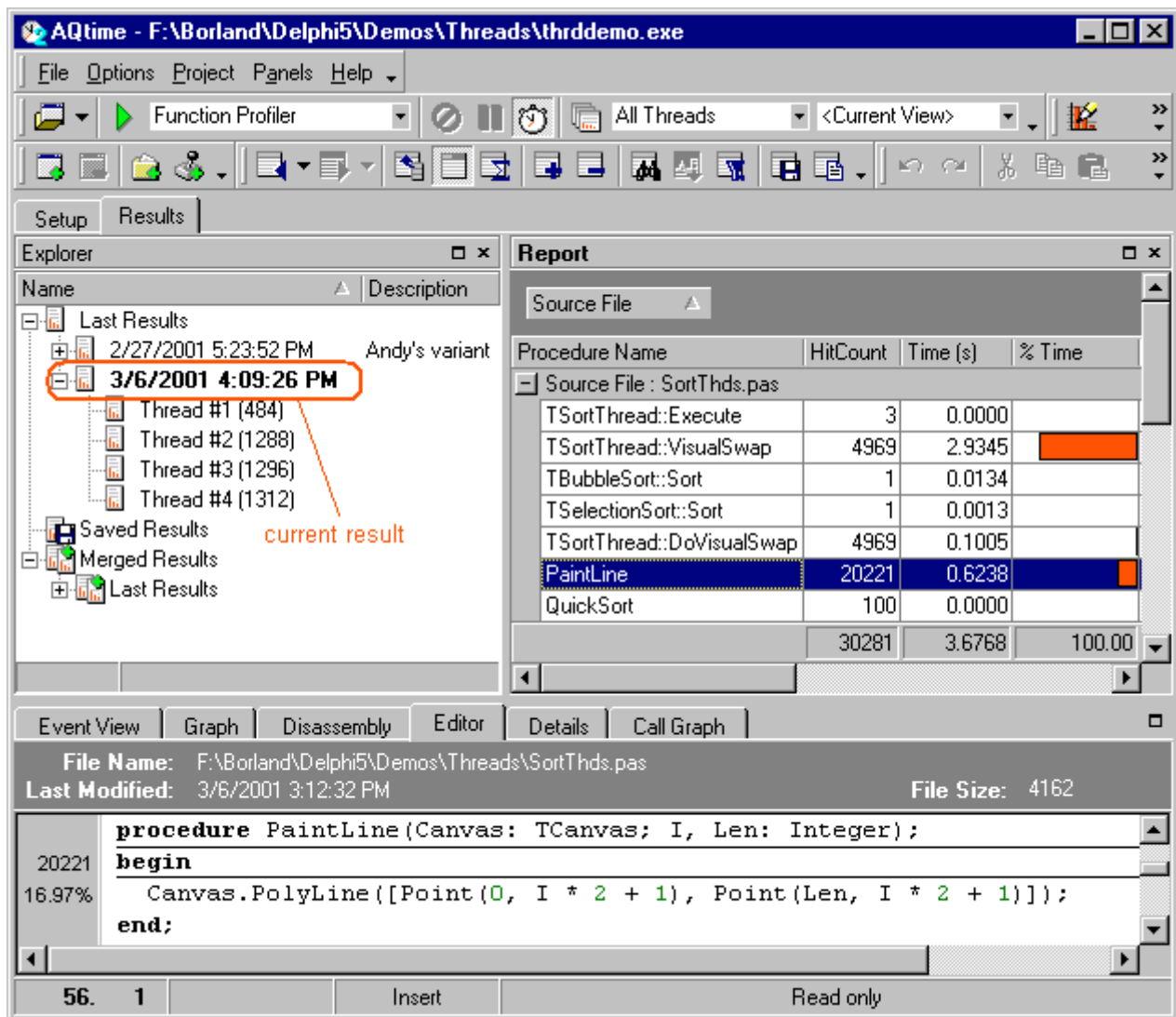
To select several separate records, hold down the Ctrl key and click each record you want to select. To select a range of records, click on the first item, hold down the Shift key and then click the last item. Shift-click always selects down or up from the last clicked item, no matter whether it was clicked with Ctrl, Shift or nothing. It also undoes all other selections, as does a simple click. By contrast, Ctrl-click adds to existing selections, it does not void them.

Working With Results

Comparing and Merging Results

As you can see in the Explorer panel, AQttime keeps a "Last Results" archive of the most recent result sets (five sets by default). These are labeled with date and time, and you can add your description directly onscreen. While a result set is still archived, you can choose to copy it to your own archive, Saved Results, where it will remain until


you delete the set or delete the project from disk. The entire Explorer contents are specific to the current project and current profiler. See Explorer Panel for details on all these points.



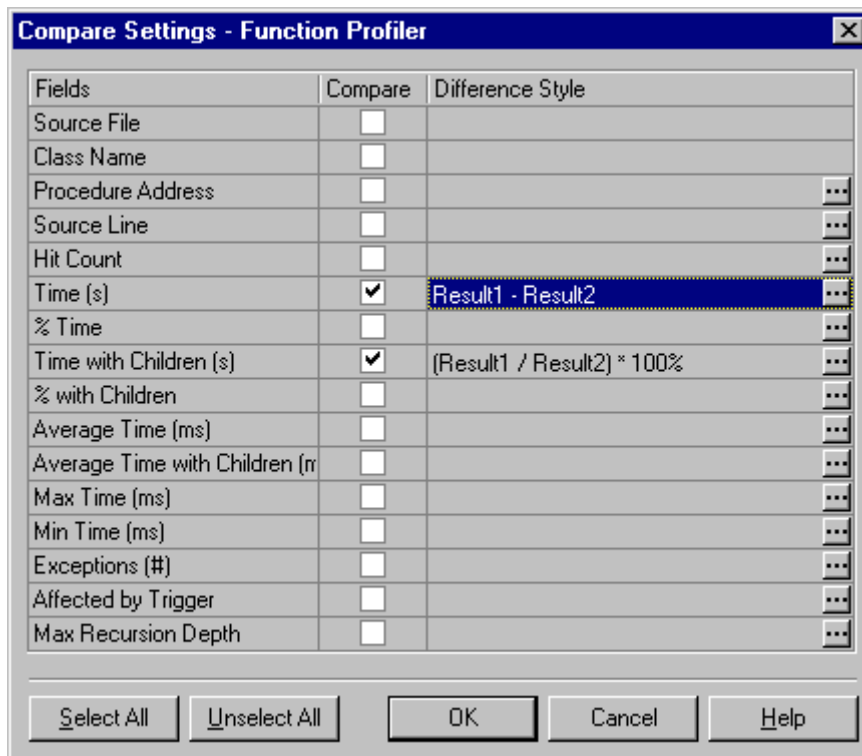
Besides allowing easy reference to past results, this system lets you set up **comparisons** between result sets or **merge** them into a new, combined result set. (With Delphi 5 applications, take care to turn on the *Update procedure names to Delphi 5* option of the Explorer panel. See *Explorer Options*).

Comparing Results

Suppose you have profiled a sorting procedure and discovered that it performs slowly. You may decide to that the algorithm must be optimized. You will try something, then profile the sort again. At this point the Compare facility steps in and lets you focus on the resulting differences in a single comparison report, laid out as a normal result report would be.

We'll call each stored result set (i.e. each dated line in the Explorer panel) a **record**. You can multi-select any number of records in the Explorer panel then choose  **Compare** from the Explorer toolbar or from the context menu (right-click) and, voila!, the comparative report will appear in the Report panel.

The comparison is configurable, of course. Select **Compare Settings...** from the Explorer context menu and you will get the Compare Settings dialog:



The actual dialog depends on the current profiler. The checkboxes in the Compare column select the columns you want to show in your comparison report.

For numeric results, if you have selected only two records to compare, you can select a "Difference Style". If you have more than two records, the Difference Style is "None", which means that columns from each record will simply be shown side by side. Other Difference Styles are simply ways to "compact" the columns from the two records into one by doing a simple arithmetic operation on them to show up the difference. These Styles use "Record 1" for the first record you selected, "Record 2" for the second.


Explorer Options includes an *Always set up Compare params* checkbox. If this is checked, then the Compare Settings dialog will pop up whenever you ask for a Compare.

Merging Results

Merging records means bringing them together into a new record, as if it was another profiler result, except that the numeric fields are replaced by the sum, average, maximum or minimum of the values in the merged records.

The resulting record goes into the Merged Results section of the Explorer panel. Note that the Explorer shows it as if it held its source records also, but this is only a way to identify the source. Only the merged result is kept in Merged Results. Like Current Results, Merged Results must be Saved (from the toolbar or the context menu) to be kept beyond the auto-archive limit (default five records).

The advantage of merges is that they focus on important statistics for the collection of records selected, such as average results over several runs. The limitation is that the application must not have changed in ways important for the profiler results. If function names have changed, for instance, then merging becomes pointless. Likewise, if a profiled function has been optimized.


To merge records, multi-select them in the Explorer panel, then choose  **Merge** from the Explorer toolbar or from the context menu. You may include records from the Merged Results section in a later merge. Merged results are just like other records, but they are flagged as merged results in Explorer (even if stored in Saved Results) by the "inclusion" of the date-times for the source records.

The only options about the merge format are which of the four available operations will be used to summarize merged results on each column. **Merge Setting...** on the context menu brings up the Merge Settings dialog for this.

If the *Auto-merge* option of the Explorer panel is on, there is a result set in the folder specified for that option, into which each new result set is merged.

Exporting Results

Profiling results can be copied to the Clipboard or exported to file from the Report panel in any of the following formats: Microsoft Excel, tab-delimited text, HTML or XML (viewable in IE 5.0 or later). Column headers are always included, in their current order on the panel.


You can multi-select the lines you want to export, and choose **Save Selection...** from the context (right-click) menu. (See *Selecting Several Records in a Panel*.) Or you can simply choose  **Save All**.

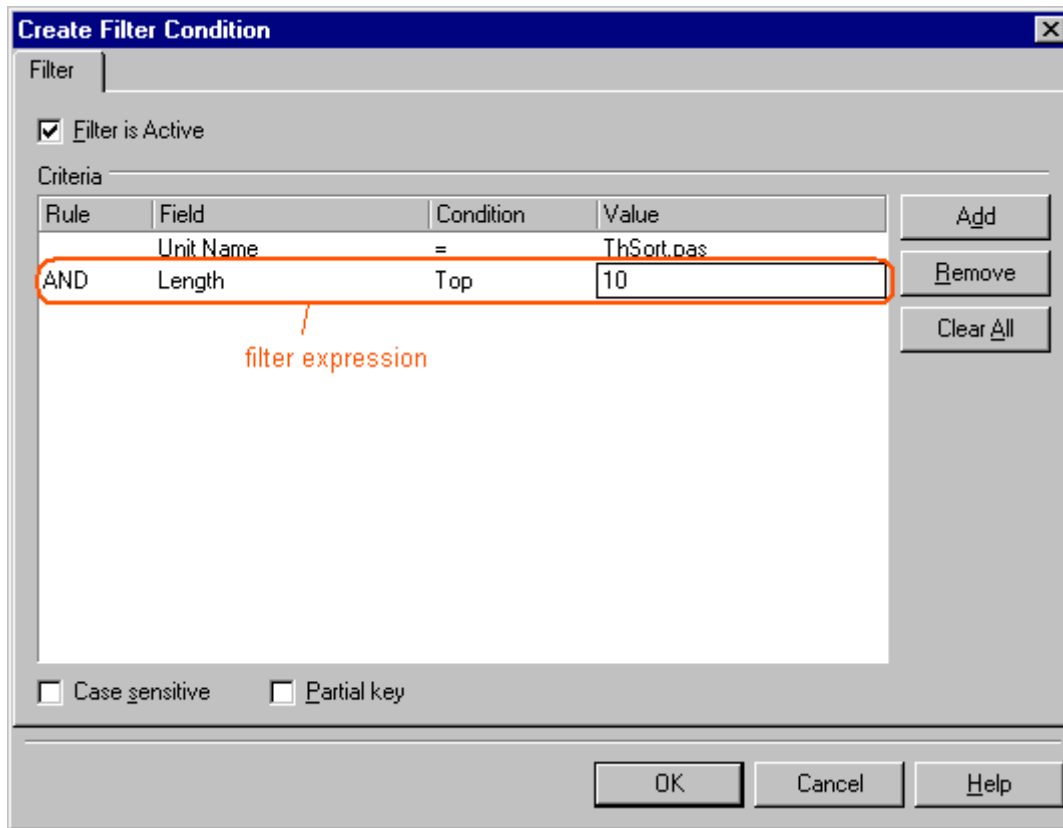
You can also multi-select lines and choose **Add to source file** from the context menu to insert the results as comments, each at the start of the corresponding source code (e.g. function source) in their source files.

Finally, you can save results to a .txt or .bin file. Select the desired result set in the Explorer panel and then choose **Save to File...** from its context (right-click) menu. Note that in this case the text file with results has another format than a file with results exported from the Report panel context menu. **Load From File...** will retrieve results saved to text this way.

Filtering Results

The output of AQtime's many profilers can be displayed in pre-selected form by defining **filters**. A filter defines a *condition* that must that records (report lines) must meet in order to be displayed in the Report panel.

To create or modify filters, choose  **Filter** on the Report toolbar or context menu. Or click the word "*Filtered*" on the left of the Report panel. This brings up the Create Filter Condition dialog:



In the dialog:


- Press the **Add** button to create a new empty filter expression.
- Next, specify the field name, condition and value that will be used by the filter.
- Click the **Filter Active** check box to activate the filter.
- Press **OK** to apply the filter to the result table.

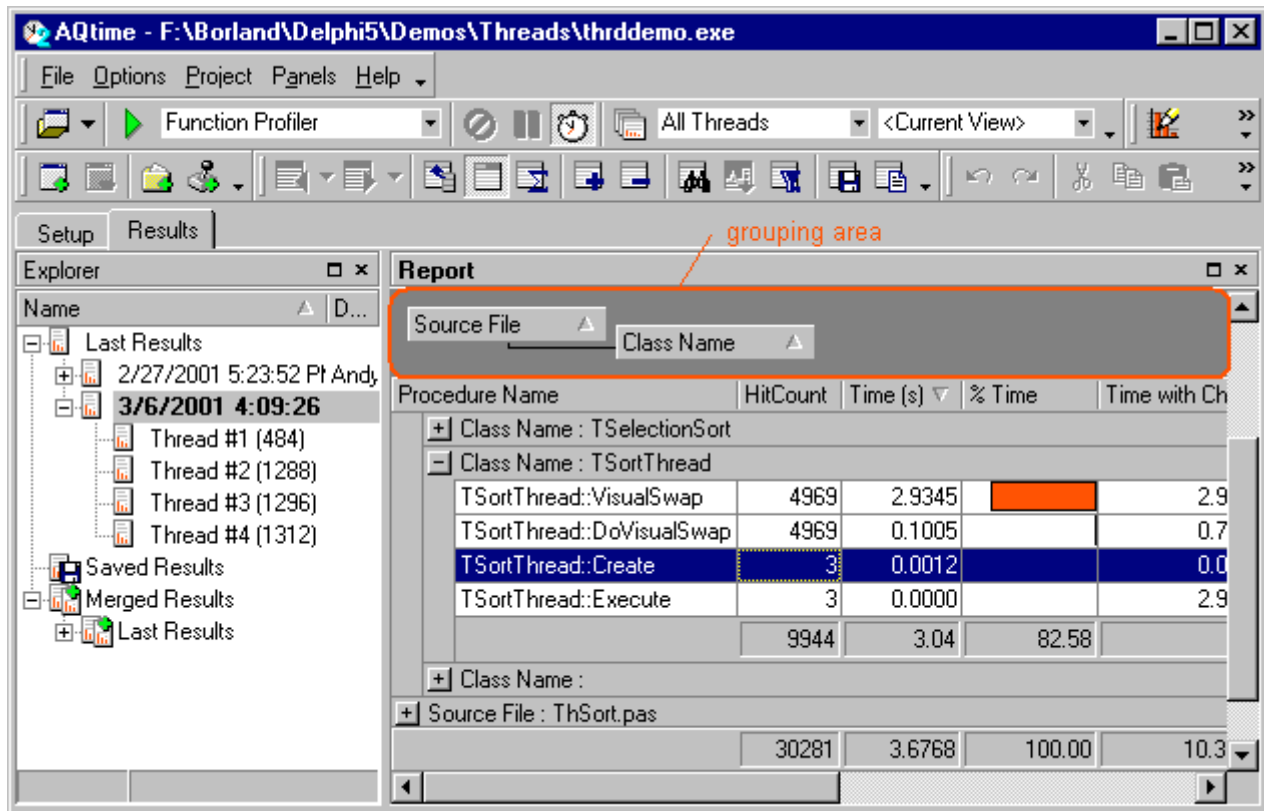
Note that by unchecking Filter Active, you can keep a filter expression available, while not using it currently.

Grouping Results

Grouping results means getting all results (records) that share a single value for one field (e.g. Class), to show on a single line in the Report panel. The column you choose to group results on becomes a synopsis of the entire result set (e.g. results grouped by class), shown in tree fashion, and the individual records are available by opening up the appropriate branch. This vastly simplifies onscreen navigation of the Report panel when there are several score or more records to show.

This is one reason all profilers include fields (e.g. **Source file**, **Class name**, etc.) that help to locate a profiled function in source code. For instance, when you group results on the Source file column, each grouped tree node corresponds to a separate source file. Try it!

You can apply grouping simply by choosing Group by this field from the context (right-click) menu for the column header. But for better control, and especially to undo grouping, you should choose  Show Group Panel from the Report toolbar or from the context menu anywhere in the panel. This opens the Group Panel:



You add columns by dragging their headers to the grouping area (you can group on more than one column), and you remove them by dragging them out of it.

Inserting Profiling Results into Source Code

You can insert profiling results into source code. AQtime inserts profiling results as comments into source files. You can refer to these notes when you are fixing problems in your application. AQtime inserts profiling results for procedures, selected in the **Report** panel. The inserted comments depend on a profiler type. For instance, if you have profiled your application with **Function Coverage**, AQtime will insert the following text:

```
// Comment was generated by AQtime Function Coverage at 9:46:52 AM 6/30/00
// Hit Count : 1
```

To insert profiling results into source code:

1. Select desired procedures in the **Report** panel. You can use SHIFT and CTRL keys to select several procedures.
2. Select **Add Result to Source File** from the **Report** popup menu.

You can switch to the **Editor** panel to verify that AQtime has inserted results into source code.

Printing Test Results from AQtime

AQtime allows you to print profiling results displayed in the **Report** panel:


1. You can export profiling results into an external file (XLS, XML, TXT or HTML) and print them via external applications, e.g. Microsoft Excel or Microsoft Internet Explorer.
2. You can print profiling results directly from AQtime. To start printing, select the **Print** button from the **Report** popup menu. AQtime will print a report using a predefined template.

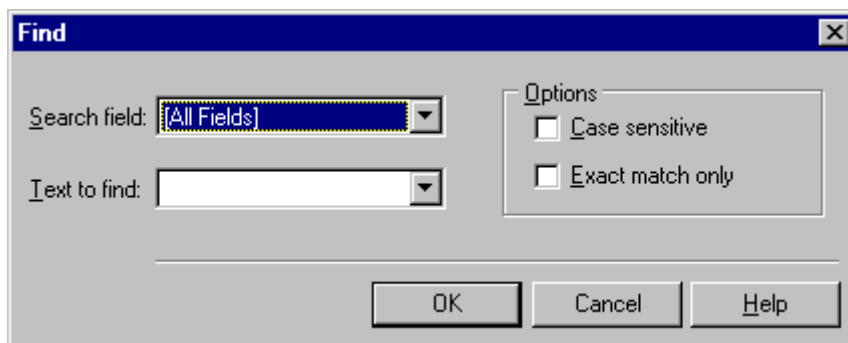
You can customize report properties before printing. AQtime contains the **Print Preview** form specially designed for this purpose. The **Print Preview** form allows you to change color and font settings before printing, change printer properties, paper size, specify background images, etc. To activate this form, select **Print Preview...** from the **Report** popup menu. After you have made all necessary changes, you can print the report from the **Print Preview** form.

Searching Results

AQtime offers two means of searching through the records in a panel.

First, you can click on a column header and begin to type the word to search for. Incrementally, the highlight will move to the corresponding record as you add letters. This is case-insensitive.

Second, in the Report panel, choosing  **Find** from the toolbar or context (right-click) menu will bring up the Find dialog, where you can specify search parameters, and search either any one column or the entire report:





Sorting Results

In any panel, on any column where sorting makes sense, you can sort records on that column by clicking the header, once for sorting in one direction, once more to switch directions. The fact that records are sorted on that column, as well as the sort direction, is shown by an arrowhead next to the column caption.

In the Report panel you can sort on several columns in succession. Hold down the Shift key. The first column you click will be the first sort key, the second will be the second key, etc. Re-click on any column without Shift, and it becomes the single sort key again.

| Procedure Name ▲ | HitCount | % Time ▼ | % w |
|-------------------------|----------|----------|-----|
| TMainForm.Button1 Click | 2 | 50,00 | |
| DoActionA | 1 | 27,64 | |
| DoActionB | 1 | 22,36 | |
| TMainForm.Button2Click | 1 | 0,00 | |
| CoverageTest | 2 | 0,00 | |
| Finalization | 1 | 0,00 | |
| initialization | 1 | 0,00 | |
| Finalization | 1 | 0,00 | |

second sorting column first sorting column

In the Report panel, the context (right-click) menu for the caption bar offers  **Sort Ascending** and  **Sort Descending**. This is another way of doing what a simple click or two would do.

Views

A *View* combines a column layout and filters in the Report panel and a Graph panel layout, for one profiler. To implement a *view* simply select it from the **Views** dropdown list on the Standard toolbar. There are several predefined *views* for each profiler type. For instance, when using the VCL Class Profiler, you can choose the **Leaked Classes Only** view to obtain a list of objects left in memory after program termination. You can also add your own views. See Views Implementation.

Panel Options

To customize internal and installed panels (or plug-ins) -

- Select **Options | Panel Options...** from AQtome's main menu. Then, choose the plug-in you wish to customize in the appeared **Panel Options** dialog.
- Or select **Options...** from the panel's context menu.

All available panel options are listed in the Inspector control. Each panel includes unique customization options. If an option includes multiple criteria, you can expand or collapse it by pressing the plus (minus) symbol to the left of the appropriate item. Some options can contain a list of strings. Use the Add and Remove buttons to edit these options.

Press **OK** to apply any changes you have made within the Options dialog.

Call Graph Panel Options

- **Number of parent levels** – Sets the depth of parent calls that will be displayed for each function. 1 means direct callers only, 2, direct callers and the functions that called them, and so forth.
- **Number of child levels** – Sets the depth of child calls that will be displayed for each function. 1 means direct callees only, 2, direct callees and the functions they call in turn, and so forth.
- **Show pointing-hand cursor** – Sets whether the mouse cursor will change to a pointing hand over areas (lines and rectangles) that can be clicked on to switch to new details.
- **Highlight** – When the mouse points a link between rectangles, these settings define how the linked rectangles are highlighted to make it clearer that they are the ones linked.

- **Active** – Sets whether any highlighting is done.
- **Parent function color** – Sets the highlight color for the parent-function rectangle.
- **Child function color** – Sets the highlight color for the child-function rectangle.
- **Function rectange background**
 - **Header part** – Sets background color for the upper part of function rectangles.
 - **Details part** – Sets background color for lower part of function rectangles.

Details Panel Options

- **Grid settings**
 - **Show summary** – Sets whether a footer row will be added to show totals for **Size** and MicroOps.
 - **Flat grid** – Sets whether the header shows as flat (on) or as 3D (off). The difference is small.
 - **Show grid lines** – Sets whether grid lines are displayed between columns and between rows.
 - **Mark selected lines** – Sets whether a narrow gutter is added on the left margin, in order to display a marker on the currently selected lines.
 - **Background color** – Sets the background color for the cells.
 - **Font color** – Sets the font color for the cell text.
- **Chart settings**
 - **Chart style** – Selects between pie and bar charts.

Disassembly Panel Options

- **Display source code** – If this is enabled, code will be displayed by source line, and clicking on a line will show the asm (disassembly) instructions for it. Alternatively, if *Auto expand* is enabled the disassembly will be always shown below the source. In either case, displaying by source line means that instructions will not always be listed by consecutive address, for instance in the case of loops. If *Display source code* is disabled, then instructions are displayed directly, in the same order as in memory.
- **Interpret addresses** – Sets whether destination names for calls or jumps will be displayed in the Target Procedure column, rather than addresses. The time cost is generally not perceptible.
- **Show instruction note as hint** – Sets whether a note for the instruction (when one is available) will be shown in a hint window. If the option is disabled, there is no hint window. See next option.
- **Show instruction note as lines** – Sets whether one or more lines will be reserved above each instruction to show the note for it when available. If the option is enabled, each line is preceded by a preview row, whether or not a note is present. The height of the preview row is specified by **Number of note lines**. An alternative way to show notes, not for all lines but for the currently selected line, is *Show instruction note as hint*.
- **Number of note lines** – If *Show note as line* is enabled, this sets the number of blank lines reserved for the note, above each instruction. The valid range is between 0 and 10.
- **Auto expand** – This option only applies when *Display source code* is enabled. See that option for details.
- **Upper case** – Sets whether asm instructions will be shown in upper case, rather than lower.
- **Grid settings**
 - **Show summary** – Sets whether a footer row will be added to show totals for the *Size* and *mOps* columns. *Size* is in bytes, *mOps* means micro-operations.

- **Show source line summary** – Sets whether an extra line will be added to the disassembly for each source line (when displayed), in order to show *Size* and *mOps* totals for the source line.
- **Flat grid** – Sets whether the header shows as flat (on) or as 3D (off). The difference is small.
- **Show grid lines** – Sets whether grid lines are displayed between columns and between rows.
- **Mark selected lines** – Sets whether a narrow gutter is added on the left margin, in order to display a marker for the currently selected lines.
- **Background color** – Sets the background color for the table.
- **Font color** – Sets the font color for the table.
- **Font** – Settings for font size (4 to 24 points), for italic and for bold.

Editor Panel Options

- **Gutter size in pixels** – Sets the width of the area to the left of the text in the Editor. A variety of information is displayed there, e.g. the main profiling result for each function. Valid values are 0 - 200.
- **Tab size in spaces** – Number of spaces between the tab columns. When not in ReadOnly mode, pressing Tab will insert spaces up to the next column. 0 to 20.
- **Gutter font** -- Font color and size (4 to 24 points) for data displayed in the gutter.
- **File extensions for highlighting** – The Editor provides different syntax highlighting for different languages, of course. It takes the highlight settings from the Registry keys for the development tool which is assigned to the extension of the current file.

File extensions for highlighting holds one list of file extensions for each development tool logged into the Registry for the machine. The extensions in each list are those for which the Editor will use that tool's syntax highlight settings. You can change them at will.

It is possible for an extension to figure in the list for more than one tool. This is especially the case for C++ extensions. In case of conflict between two lists, the precedence order runs from MSVC (highest), through VB, Delphi, BCB and Java, to .NET (lowest). In other words, if for instance .cpp is in the list for both MS VC++ and Borland C++Builder, the MSVC Registry settings will be used.

For extensions not present in any list, the Delphi settings are used if present in the Registry, else whichever available settings have highest precedence.

Note: The Registry syntax highlight settings set both font appearance and font *language*. If for instance you have set VC++ for Scandinavian fonts then, in the Editor, the files you associate with C++ will use the VC++ highlight *and* Scandinavian high-ANSI characters.

Event View Panel Options

- **Exceptions** -- Settings for the display of exception events. None (except *Active*) has any effect unless *Active* is enabled.
 - **Active** – Enables exception logging. When enabled, exceptions are shown in Event View, as set by the sub-options below, and their time is counted in the function where they occur. Else, exceptions are neither logged as events nor counted as part of execution time.
 - **Max consecutive exceptions** – Number of exception events, uninterrupted by any other event, after which exception logging will be disabled until the next non-exception event. This saves profiling time during exception loops.
 - **Show call stack** – Sets whether, on each exception, the call stack status will be recorded for later display in the Event View panel. This recording will slow down profiling if there are many exceptions.

- **Find call stack from frames** – This is not an "option", but an input field that specifies to AQtme whether the profiled application is compiled with stack frames for all functions. If this is the case, AQtme will use the frames to trace the stack when exceptions occur. Else, it will use its own algorithm for stack tracing. This may produce erroneous entries in the Call Stack pane of the Details panel.
- **Hide IsBadPtr exceptions** – If this is enabled, exceptions raised by IsBadPtr, IsBadWritePtr, IsBadCodePtr or IsBadStringPtr will not be logged to Event View.
- **Include no-debug in stack** – The call stack may well include functions for which there is no debug info, typically from pre-compiled libraries. If this option is enabled, the call stack shown for exception events will include these functions. Else, they will be suppressed from the call stack display.
- **Clear on application start** – Sets whether Event View will begin empty each time the profiled application is launched, that is, on each profile run. Else, it will keep events from previous runs.
- **Time from application start** – If enabled, times shown in the Time column are elapsed times from application start. Else, they are system time.
- **Auto expand** – Sets whether Event View nodes are to be shown expanded by default.

Explorer Panel Options

- **Number of recent results to keep** – Sets the number of entries in the Recent Results list. The default is five.
- **Always set up Compare params** – Sets whether to show the Compare Settings dialog every time you compare results. Disabled by default.
- **Background color** – Sets the background color for the Explorer panel.
- **Font color** – Sets the font color for the Explorer panel.
- **Show results for all profilers** – When this is disabled – the default state - the Explorer panel only includes results for the currently selected profiler. When this is enabled, each profiler is shown as a main branch in the panel, with its results in tree view under that branch.
- **Update procedure names to Delphi 5** – Applies only for compare or merge. Sets whether to update Borland VCL names to the versions used in Delphi 5. The option should be disabled (the default) for all compilers except Delphi 5. The latter uses a different format for function names in its debug info, and in that case the option should be enabled to ensure correct functioning during Compare or Merge.
- **Auto-merge** – Auto-merge is a feature where a special, separate folder accumulates the same results as are generated in the main panel, but with each new result set merged with the previous one. In other words, the first result set in the auto-merge folder is the same one as in the main panel, but the second is the main panel's second result set merged with the first, the third is the main panel's third result set merged with the two preceding ones, etc.
 - **Active** – Enables or disables the Auto-merge feature. Disabled by default.
 - **Folder name** – Sets the name for the special auto-merge folder, when the feature is enabled.

Graph Panel Options

The Graph panel offers a rich choice of options which complement the Chart Properties (available from **Properties** on the Graph context menu).

- **Allow zoom** – If this is checked, it is possible to outline an area of the graph with the mouse (drag top left to bottom right) and get it to fill the panel on release. To revert to the previous view, begin dragging inside the graph, and stop higher and outside it.
- **Animated zoom** – Sets whether the zoom action (if allowed) is animated.

- **Show axes** – Sets whether horizontal and vertical axes will be shown on charts.
- **Background color** – Sets background color for the charts. See *Gradient*.
- **Gradient** – Alternative background color setting, using a gradient.
 - **Active** – Sets whether the gradient settings (below) will be used instead of the *Background color* setting. If enabled, background color will smoothly vary from bottom to top.
 - **Bottom color** – Sets the background color at the bottom of the chart.
 - **Top Color** – Sets the background color at the top of the chart.
- **Monochrome** – Sets whether charts will use only black, white and dithers (as in black and white printing) instead of color.
- **3D view** – Sets whether bars will be shown as 3D solids, rather than flat.

Macro Engine Options

- **Record AQtime events** Sets whether the Macro Engine tracks mouse clicks and keypresses that occur in AQtime. Default: on.
- **Record application events** Sets whether the Macro Engine tracks mouse clicks and keypresses that occur within the profiled application. By default, this option is on.
- **Record all events** Sets whether the Macro Engine tracks mouse clicks and keypresses that occur anywhere onscreen. This feature may be useful when the profiled application interacts with external applications.
- **Auto start application** If this option is checked, the Macro Engine automatically launches the profiled application when you start macro recording, and automatically stops recording when the application is terminated.
- **Delay between events (ms)** Time interval, in milliseconds, inserted between each instruction on playback.
- **Show indicator** Sets whether an indicator will show onscreen when recording or playing a macro. Default: on.
- **Language for new macros** Scripting language in which new macros will be recorded (DelphiScript, VBScript or JScript).
- **Record shortcut** Keyboard shortcut to start recording.
- **Play shortcut** Keyboard shortcut to start playback.
- **Stop shortcut** Keyboard shortcut to stop recording or playback.
- **Editor** Calls the Macro Editor Options dialog.

Note that Macro Engine shortcuts are system-global. They work in all applications, and may override application shortcuts. If, for instance, some application uses Shift-F12 to open a file, and this is also the Record shortcut. Pressing Shift-F12 within that application will not call the Open File dialog, but will start recording an AQtime macro.

Monitor Panel Options

- **Refresh Intervals (ms)** The refresh interval (in milliseconds) used for the Counter, Histogram and Graph views.

- **Counter** Valid range is between 0 and 100,000. If its value is 0, the Counter view is refreshes each time the examined parameters are changed.
- **Graph** Valid range is between 1 and 100,000.
- **Histogram** Valid range is between 1 and 100,000.
- **Histogram** Colors for the Histogram view.
 - **Background Color** Color of the view's background.
 - **Font Color** Text color within the view.
 - **Mark Color** Color used for marks.
 - **Line Width** The width of lines within the view. By default, it is 2.
- **Common Series Color**
 - **Sum of All Series** Color used to draw the sum of all the series.
 - **Sum of All Visible Series** Color used to draw the sum of the series checked in the Series dialog.

Color settings can be changed through the standard **Color** dialog. To call it, press the ellipsis button on the right of the desired option.

PEReader Options

The PEReader plug-in includes three options for optimization. The first one, *Show paths*, sets whether the Modules Hierarchy pane displays imported libraries with paths. The other two options, *Background color* and *Font color*, specify accordingly the background and font colors in the PEReader panel.

Report Panel Options

- **Show summary** – Sets whether a footer row will be added to show column totals.
- **Show group summary** – On the Report context menu, there is a *Show Group Panel* option that will allow grouping results on a column. When this is done, the Set Summary Field option, also on the context menu, allows row summaries to be displayed for each group node. **If** results are grouped, **but** there is no row summary, **then** *Show group summary* enables an alternative way of displaying group summaries – a row is added below each group to show column summaries for the group.
- **Single-click details** – Sets whether a single click on a line, rather than a double-click (the default), will update Details and other panel with data for the element on that line (usually a function). See AQttime Panels) by a single click in the Report panel.
- **Flat grid** – Sets whether the Report grid shows as flat (on) or beveled (off)..
- **Show grid lines** – Sets whether grid lines are displayed between columns and between rows.
- **Mark selected lines** – Sets whether a narrow column is added to the left of the table, in order to display a marker for the currently selected lines.
- **Background color** – Sets the background color for the table.
- **Font color** – Sets the font color for the table.

Setup Panel Options

- **Activate after loading** – Sets whether AQttime will switch to the Setup panel after reading the debug info for the application, that is, after loading a project. Enabled by default.


- **Auto-select new elements** – Sets whether new Areas, new Triggers and new elements added to Areas or Triggers, will be checked on being added (*Auto-select* on) or unchecked (*Auto-select* off).
- **Show methods only under class** – If this option is enabled, the Modules pane (i.e. left-hand treeview) will only show methods as sub-elements of their class. Else it will also show them as elements of their unit. See Setup panel.

Profilers Reference

Static Analysis

The **Static Analysis** profiler does not launch the tested application, but analyses the debug data included in the executable to find such information as:

- the size of routines in bytes,
- their length in source code lines,
- the routine addresses in memory,
- the binary code generated for the routine, etc.

When you press the  Run button for Static Analysis, the application does not execute; the profiler simply checks the entire executable(s). Area and Trigger settings are ignored. Some questions that can be answered by this speedy analysis are:

- What code is used by an application? If the application includes a massive module only to use one or two functions from it, you might choose to extract them from the module, or to re-implement them so as to save on application size and dependencies.
- What routine is located by a certain address? For instance, if the application raises an exception, you can launch Static Analysis and determine from the exception address reported what routine caused it.
- What binary code was produced by the compiler for a routine? This can tell you for instance if array or string parameters are being passed by copying the data to the stack, or only a pointer.

Once Static Analysis is done, each row in the Report panel corresponds to a routine in your code:

The screenshot shows the AQtime Profiling tool interface. The main window displays a report of profiling data. The report table is as follows:

| Source File | Procedure Name | Source Line | Line Count | Length | Procedure Add... |
|-------------|---------------------|-------------|------------|--------|------------------|
| MAIN.PAS | TMainForm::Button2C | 71 | 3 | 24 | \$0044CAA0 |
| MAIN.PAS | TMainForm::Button1C | 64 | 5 | 124 | \$0044CA24 |
| MAIN.PAS | ProfilingTest | 55 | 8 | 124 | \$0044C9A8 |
| MAIN.PAS | DoActionC | 46 | 7 | 40 | \$0044C980 |
| MAIN.PAS | DoActionB | 41 | 2 | 8 | \$0044C978 |
| MAIN.PAS | DoActionA | 36 | 2 | 11 | \$0044C96C |
| Menus.pas | UpdateItem | 1977 | 5 | 55 | \$00446AD4 |
| Menus.pas | UpdateItem overload | 2376 | 5 | 55 | \$004476C8 |
| Menus.pas | ReturnAddr | 113 | 3 | 8 | \$00443070 |
| | | | 15623 | 226081 | |

Below the report, the 'Event View' tab is selected, showing a list of events with columns for Type, Address, Hex, Instruction, Target, Target Procedure, Size, and uOps.

| Type | Address | Hex | Instruction | Target | Target Procedure | Size | uOps |
|------|---|--------------|----------------------------|----------------------|------------------|------|------|
| + | begin | | | | | | |
| + | DoActionA(); {call the procedure that perform action A} | | | | | | |
| - | DoActionB(); {call the procedure that perform action B} | | | | | | |
| ⚡ | \$0044C9B6 | E8BDFFFFFFFF | call -\$00000043 | \$0044C978 DoActionB | | 5 | 4 |
| + | for i:=1 to Param do | | | | | | |
| - | DoActionC(); {call the procedure that perform action C} | | | | | | |
| ⚡ | \$0044C9CC | E8AFFFFFFFFF | call -\$00000051 | \$0044C980 DoActionC | | 5 | 4 |
| | \$0044C9D1 | FF45F8 | inc dword ptr [ebp - \$08] | | | 3 | 4 |
| - | MessageBox(MainForm.Handle, 'Execution finished', 'Profiling Sample', MB_OK); | | | | | | |
| | \$0044C9D9 | 6A00 | push +\$00 | | | 2 | 3 |

The columns hold the following information:

| | |
|--------------------------|---|
| Class Name | If the routine is a method, name of the class it belongs to. |
| Length | Size in bytes of the compiled code for the routine. |
| Line Count | Routine length in source code lines. Note that this is according to debug info and may not fit the actual source file, as the compiler uses its own rule for what is a source line. |
| Module Name | Name of the executable module (exe, dll, etc.) holding the routine. |
| Procedure Address | Routine address in memory. |
| Procedure Name | Full name of the routine, including class name. |
| Source File | Name of the source file for the routine. |
| Source Line | Source file line number where the routine's implementation begins. |
| Unit Address | Address of the compiled linkage unit in memory. |
| Unit Length | Unit size in memory, in bytes. |

Unit Name Name of the compiled linkage unit.

To explore the binary code of a routine, double click its line in the Report panel, and then switch to the Disassembly panel.

To explore the source code of a routine, double click its line in the Report panel and switch to the Editor panel. The source can only be displayed if the source file is on the Search Path. See also AQttime Panels.

Coverage Profilers

There are three Coverage profilers. What parts they profile during execution is determined by the current Area and Trigger settings, as usual. The profilers are:

- **Function Coverage** – Determines whether each profiled function was executed during the run. This profiler is much faster than the two below.
- **Line Coverage (Grouped by function)** – Determines whether each profiled source code line was executed during the run. The line results are grouped by their source function in the Report panel, one report line per function.
- **Line Coverage (Grouped by File)** – This does the same as Line Coverage, but grouping by source file (one file per line) rather than by function. This is more compact and helps get an overall view. Also, this profiler is faster than Line Coverage (by function), since it does not gather function-specific information. If you need to track all lines covered or not covered, begin by using this profiler on the FULL CHECK area. This will let you focus on the the problem files first, and then you can narrow the analysis to these files and use Line Coverage (by function) with them to drill down further.

Coverage profilers allow you to keep track of untested code as testing progresses over time. They also let you find unnecessary code that you may remove, when the function or line remains unexecuted under all possible conditions.

Since Coverage profiling will tend to be applied to large areas of your application, and since, especially, the Line Coverage profiler are inherently CPU-intensive, there is a special *Warning level* option for these profilers that will warn you when you have elected to profile more than a preset number of functions or lines in a single run. See *Coverage Profiler Options*. Note also that Line profilers should not be used with applications that use hooks. See *Profiling Applications That Use Hooks* in on-line help.

Like all profilers, the Coverage profilers display their results in the Report panel, one line per function or per source file. Information is also available separately for each thread in a multithreaded application. To view coverage results for one thread, use either the Explorer panel or the **Threads** dropdown list on the Standard toolbar. (See *Multithreaded Application Profiling*).

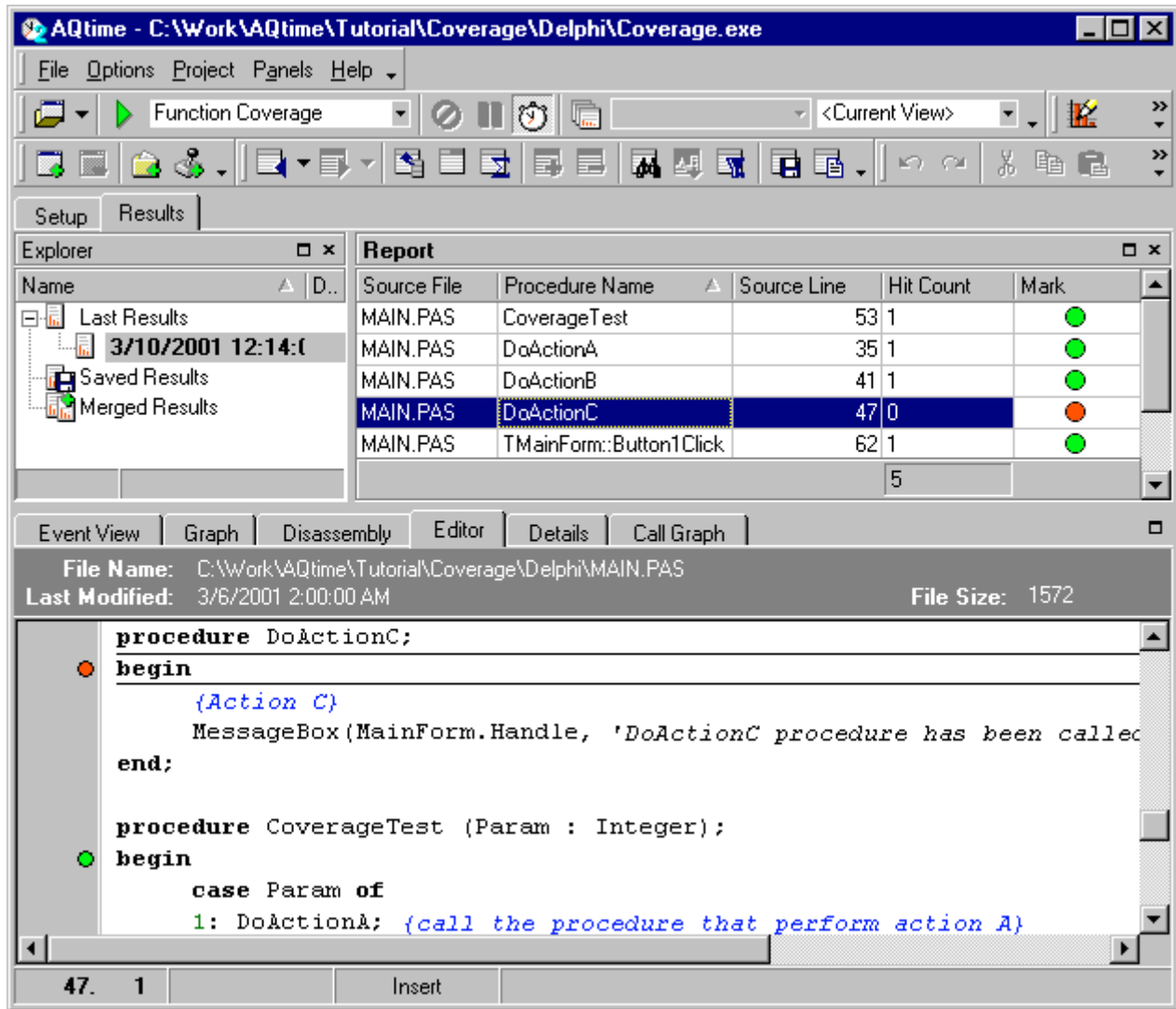
As usual, double-clicking on a line in the Report panel will move the contents of the Editor and Disassembly panels to that line. Switching to the Editor, you will find that the functions or lines executed are marked by green dots in the gutter, while the non-executed ones have red dots. (See *AQttime Panels*).

Note that Editor can display incorrect profiling-related information for some C++ applications that use several functions based on the same template (see *Template Functions Restriction* in on-line help). In this case, refer to the Report panel to get correct results.

The Report format for each of the three Coverage profilers is explained separately in the subsequent topics.

Function Coverage Results

Here is an example of the results from a **Function Coverage** profile run:



Notice that in the Editor the red dot in the gutter to the left of the MessageBox source line shows that this function was not executed.

The Report panel displays the following columns:

| | |
|--------------------------|--|
| Class Name | If the function is a method, name of the class it belongs to. |
| Hit Count | 1 if the function was executed, else 0. This field is hidden by default. |
| Mark | Green dot if the function was executed, red dot if not. |
| Procedure Address | Function address in memory. |
| Procedure Name | Full name of the function, including class name. |
| Source File | Name of the source file for the function. |
| Source Line | Source file line number where the function's implementation begins. |
| Unit Name | Name of the linkage unit holding the function. |
| Module Name | Name of the executable module (exe, dll, etc.) holding the function. |

Affected by Trigger Indicates whether Triggers turned off the profiling, so that some calls may not have been counted.

The **Mark** column helps you quickly see which procedures were executed. The **Class Name**, **Unit Name**, **Source File** and **Source Line** columns help to locate the function in source code. You can sort the results by clicking on any column header.

Line Coverage (Grouped by Function) Results

Here is an example of the results from a Line Coverage (by function) profile run:

The screenshot shows the AQtime application window. The 'Report' panel contains the following data:

| Source File | Procedure Name | Lines Covered | Lines UnCovered | Total Lines | % Cov... |
|-------------|-------------------------|---------------|-----------------|-------------|----------|
| MAIN.PAS | DoActionC | 0 | 3 | 3 | 0.00 |
| MAIN.PAS | CoverageTest | 4 | 2 | 6 | 66.67 |
| MAIN.PAS | TMainForm::Button2Click | 3 | 0 | 3 | 100.00 |
| MAIN.PAS | TMainForm::Button1Click | 3 | 0 | 3 | 100.00 |
| MAIN.PAS | DoActionB | 3 | 0 | 3 | 100.00 |
| MAIN.PAS | DoActionA | 3 | 0 | 3 | 100.00 |
| | | 16 | 5 | 21 | |

The 'Editor' panel shows the source code for 'CoverageTest' (MAIN.PAS):

```

procedure CoverageTest (Param : Integer);
begin
    case Param of
        1: DoActionA; {call the procedure that perform action A}
        2: DoActionB; {call the procedure that perform action B}
        3: DoActionC; {call the procedure that perform action C}
    end
end;
  
```

The Report panel displays one function per line, using the following columns:

Source File Name of the source file for the function.
Class Name If the function is a method, name of the class it belongs to.

| | |
|--------------------------|--|
| Procedure Name | Full name of the function, including class name. |
| Lines Covered | Number lines in the function that were executed at least once. |
| Lines Uncovered | Number lines in the function that were never executed during the run. |
| Total Lines | Number of source lines for the function, according to debug information. |
| % Covered | Percentage of total lines that were executed at least once. |
| Procedure Address | Function address in memory. |
| Unit Name | Name of the linkage unit holding the function. |
| Source Line | Source file line number where the function's implementation begins. |

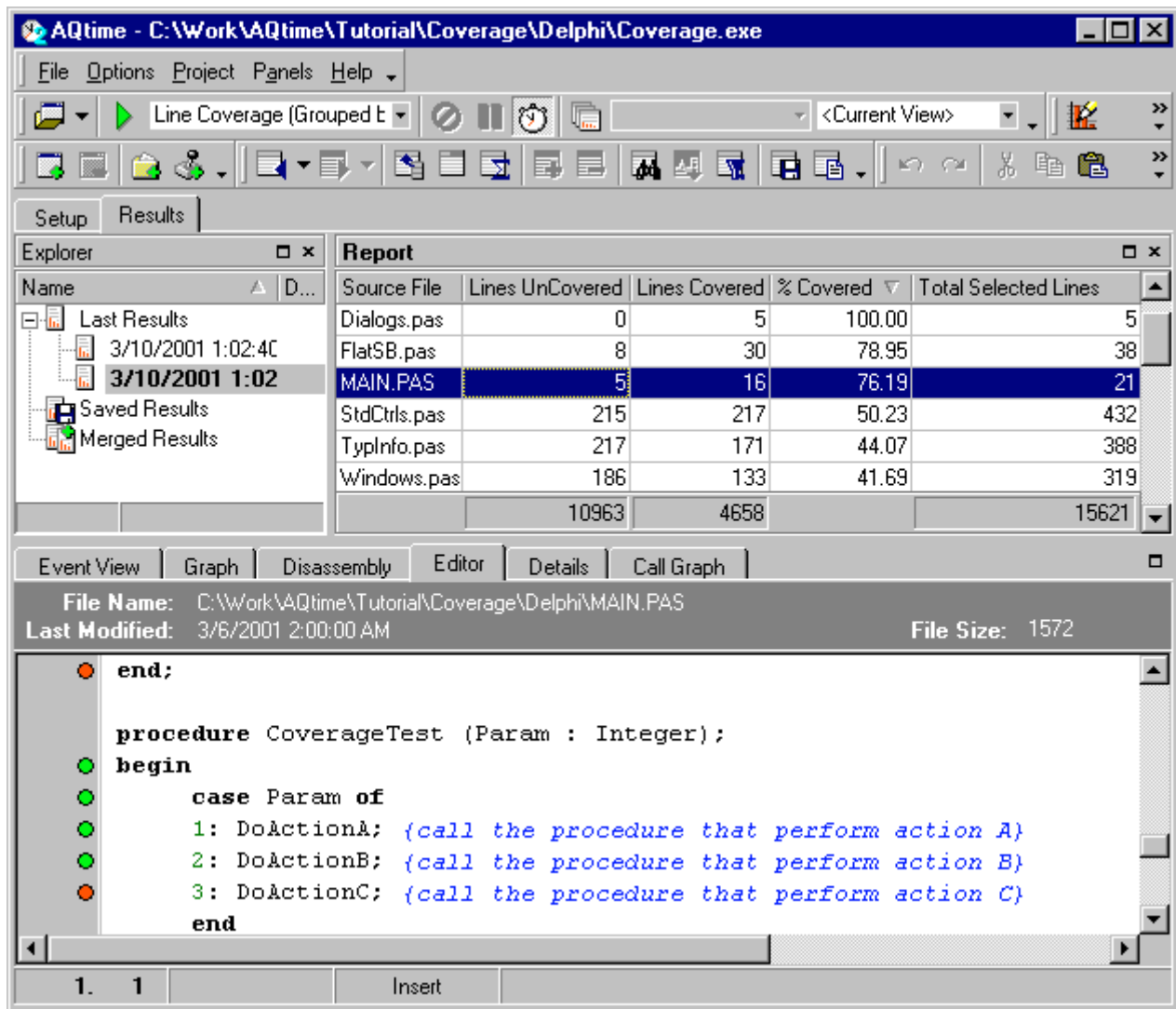
Use the **Lines Covered**, **Lines Uncovered**, **Total Lines** and **%Covered** columns to identify untested code. For instance, if the function includes a large number of lines, of which only a small percentage were executed during a seemingly "complete" test for the function, you might examine the function's algorithm.

To see what functions had the most unexecuted lines, select Procedures covered less than 50% from the **View** dropdown list on the Standard toolbar. The Covered and Uncovered views will display only functions not entirely skipped, or only those entirely skipped – a result you normally get directly from the Function Coverage profiler, which is faster than Line Coverage.

Note that AQtime gets line information as it is specified in the debugger information. Sometimes, this information differs from what appears in your source code. For instance, the number of source lines reported may not coincide with the one you expect.

Line Coverage (Grouped by File) Results

Here is an example of the results from a Line Coverage (grouped by File) profile run:



The **Report** panel displays one line for each source file, using the following columns:

| | |
|-----------------------------|---|
| Source File | File name. |
| Lines UnCovered | Number of unexecuted lines in the file. |
| Lines Covered | Number of executed lines in the file. |
| % Covered | Percentage of lines executed in the file. |
| Total Selected Lines | Number of lines selected for profiling. Should be Lines Uncovered plus Lines Covered. |

As always, this information is dependent on debug info attached to the executable. With this profiler especially, you should be on the lookout for unexpected discrepancies. Some compilers, for instance, such as Borland Delphi, will skip functions that are never called (this is named Smart Linking). Thus, the debug information will log fewer functions and fewer lines than there are in the source file.

Hit Count Profilers

The **HitCount** profilers sort out the most frequently used portions of your code, or the least frequently used, by counting how many times each function or line was executed during the profile run (that is, its hit count). There are many uses for this information. For instance, the most frequently used functions are the ones where a change in time or resource efficiency will have the most effect on the application. Even simpler, unexpectedly frequent or infrequent hits indicate that the code is not doing quite what you designed it to do.

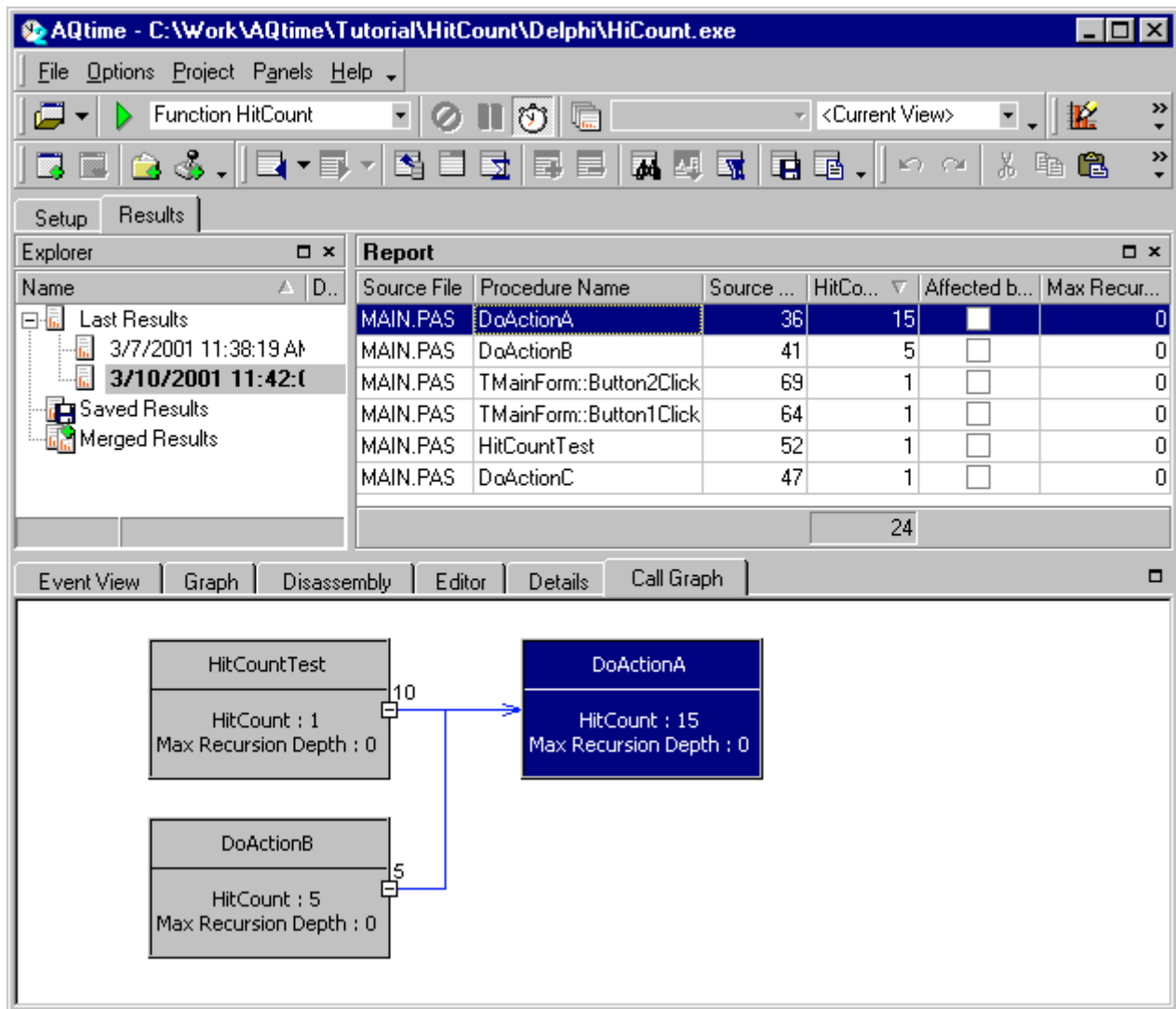
There are two HitCount profilers, **Function HitCount** and **Line HitCount**. Aside from the obvious difference (the first counts hits on functions, the second on single source code lines), there are some similarities between the two –

- Both profile only selected Areas (see *Defining Areas to Profile*).
- Both collect separate results for each thread in a multithreaded application (see *Multithreaded Applications Profiling*).

However, the two profilers are implemented rather differently --

- Function HitCount profiler is linked to functions, therefore it is controlled by Triggers (see *Using Triggers*).
- Function HitCount is light and rapid. Hit count is one of the statistics collected by the Function Profiler, but you will find that Function HitCount works faster and provides results that are simpler to analyze. Normally, before digging in, you need a general map of your application's functioning, and this is what Function HitCount will provide. Once you've narrowed down the questionable areas with these results, you can use the Function Profiler to find more details about them.
- Line HitCount inserts a debug breakpoint for every source line in its profiling Areas, and these breakpoints remain for the duration of the run, so it will tend to slow down the application being profiled. It is in your interest to reduce the Areas selected for profiling, in terms of total line count, and to use the *Max Hit Count* option, which is something like a "safety valve". If it is set, the profiler will stop counting hits on any line once the Max Hit Count has been reached for it. In the case of the Line HitCount profiler, this can remove bottlenecks in the profiling process itself. See *Line HitCount Options*. Note also that Line profilers should not be used with applications that use hooks. See *Profiling Applications That Use Hooks* in on-line help.

Despite their differences, the two profilers provide their **results** in similar formats. Here is an example of the output for Function HitCount --

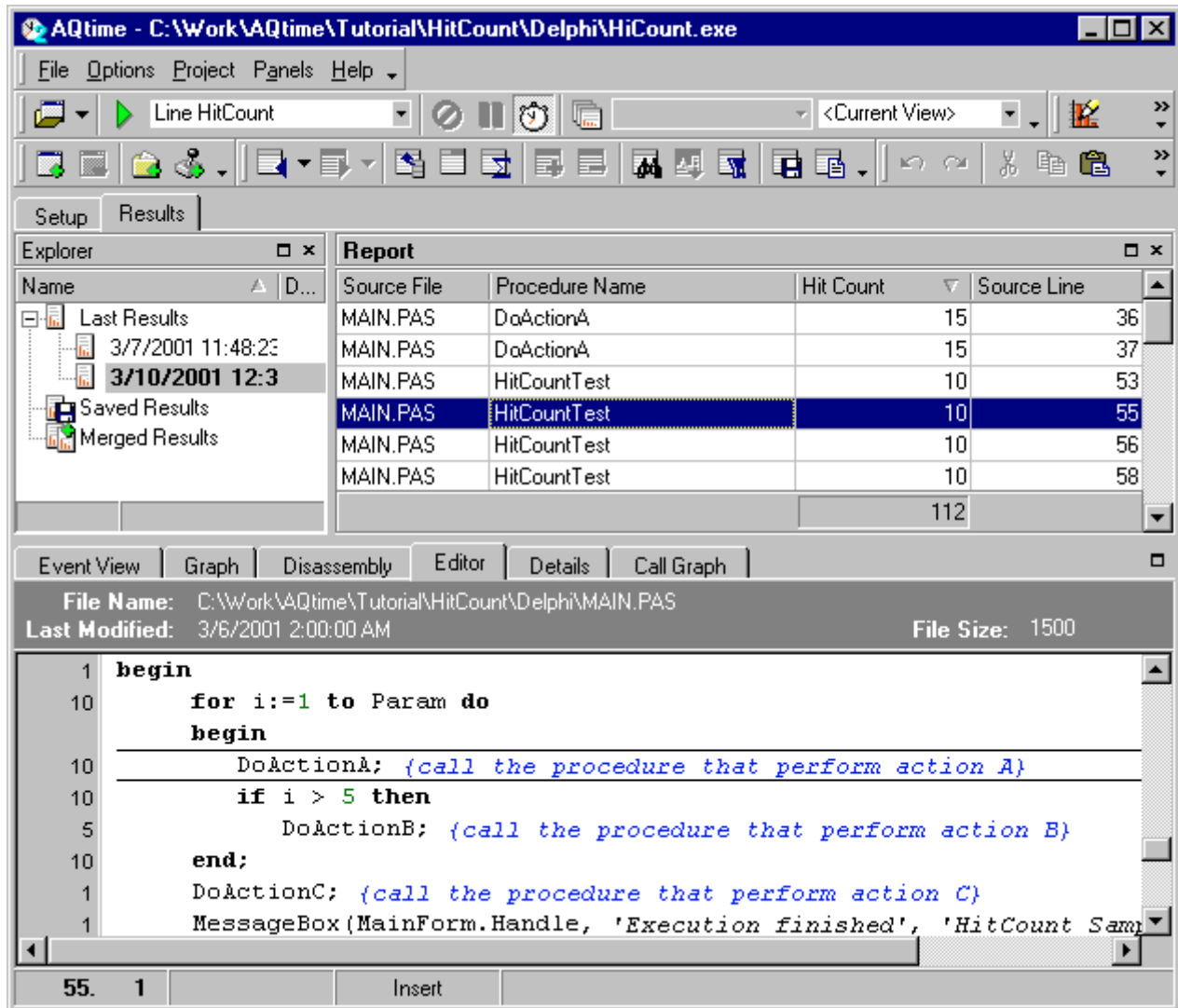


The Report panel displays one function per line, using the following columns:

| | |
|----------------------------|---|
| Class Name | If the function is a method, name of the class it belongs to. |
| HitCount | Number of times the function was called. |
| Procedure Address | Function address in memory. |
| Procedure Name | Full name of the function, including class name. |
| Source File | Name of the source file for the function. |
| Source Line | Source file line number where the function's implementation begins. |
| Unit Name | Name of the linkage unit holding the function. |
| Module Name | Name of the executable module (exe, dll, etc.) holding the function. |
| Affected By Trigger | Indicates whether Triggers turned off the profiling, so that some calls may not have been counted. |
| Max Recursion Depth | Maximum recursion depth (max number of coexisting calls within one thread) reached during the run for this function. This value is calculated only if the |

option *Track recursion depth* is set.

And here is an example of the output for Line HitCount --



The Report panel displays one source line per line, using the same columns as for Function HitCount, above, except that Module Name and Affected by Trigger are absent, and **Source Line**, the line number within Source File, is added. "Procedure" is the function to which the line belongs.

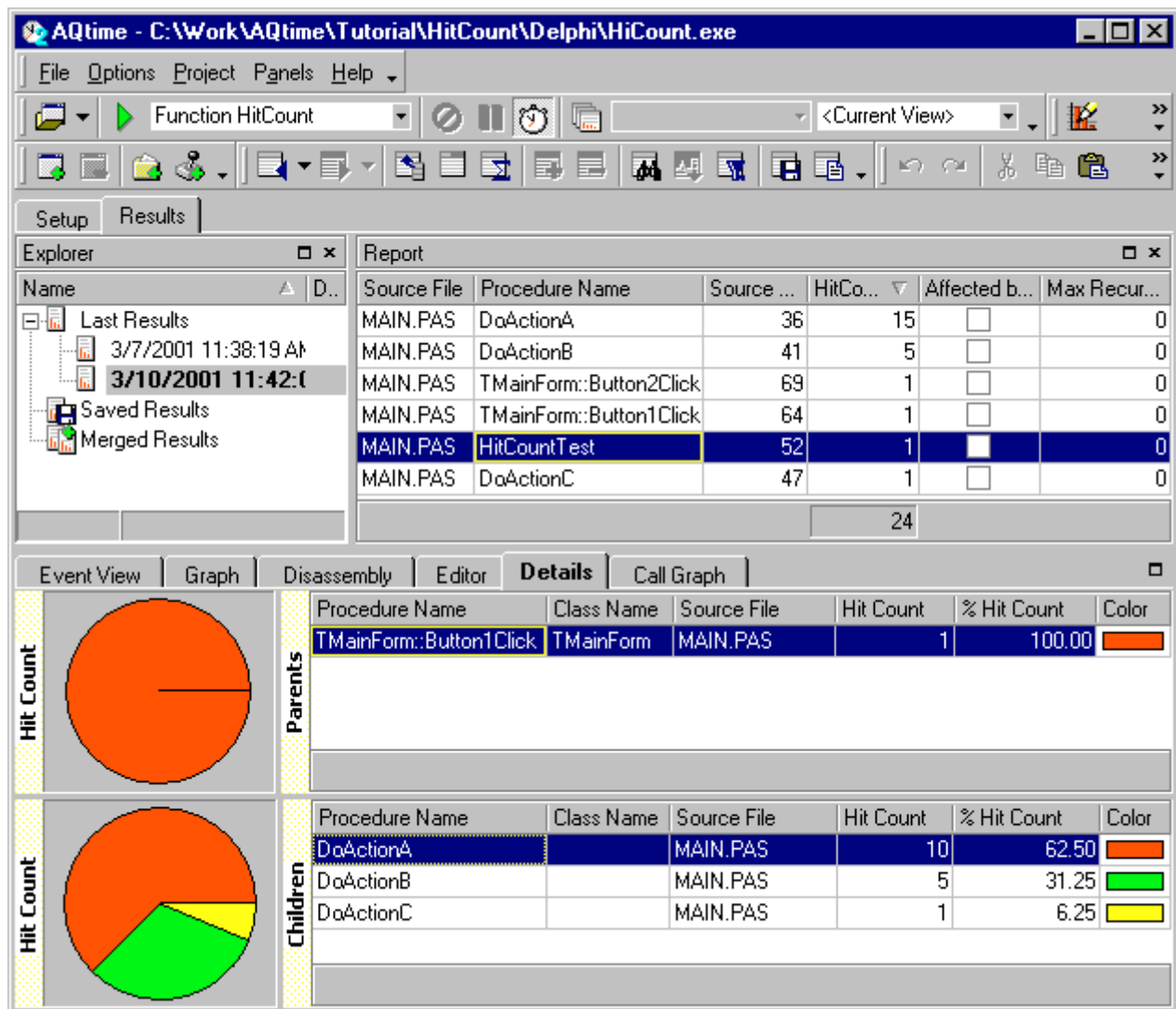
You can sort on the HitCount column, or filter the results according to that value, in order to point out the most frequently used functions or lines. In fact, Function HitCount provides a predefined view, **Top 10 Procedures**, specially for this. Line HitCount provides a similar view called **Top 20**. You can select these from the **Views** dropdown list on the Standard toolbar. (See *Views Implementation*.)

Double-clicking on any line will update other panels to the selected function or source line – Editor, Disassembly and, in the case of Function HitCount, the quite useful Details panel (about which, more below). You can then shift to these panels for more detail (see *AQtime Panels*). In the case of the Editor panel you will find the hit count displayed in the gutter, at the start of each profiled function in the case of Function HitCount, and at the

left of each profiled line in the case of Line HitCount. For the latter profiler, this is the most convenient way of check hit counts.

Note that the gutter can display incorrect information for some C++ programs that use several functions based on the same template (see *Template Functions Restriction* in on-line help). The correct profiling results are nonetheless displayed in the Report panel.

If the *Call relationship tracking* option is on, Function HitCount uses the Details and Call Graph panels to display the call relationships between the current function and those that call it ("parents"), and between it and those it calls ("children"). Details holds the same columns as the Report panel (see *Function HitCount Profiler - Details*). Here is a sample:



Double-clicking on a line in Details (a rectangle in Call Graph) will update the other panels to the function displayed on that line (rectangle). Switching from panel to panel in this way, in order to get the marrow out of the Function HitCount Profiler results, is made much easier by the "browser" buttons, **Back** and **Forward** on the Report toolbar.

Some routines listed in Details may have zeros in the HitCount column. It means that the routine was called recursively during profiling. See *Profiling Recursive Functions*.

Function Profiler

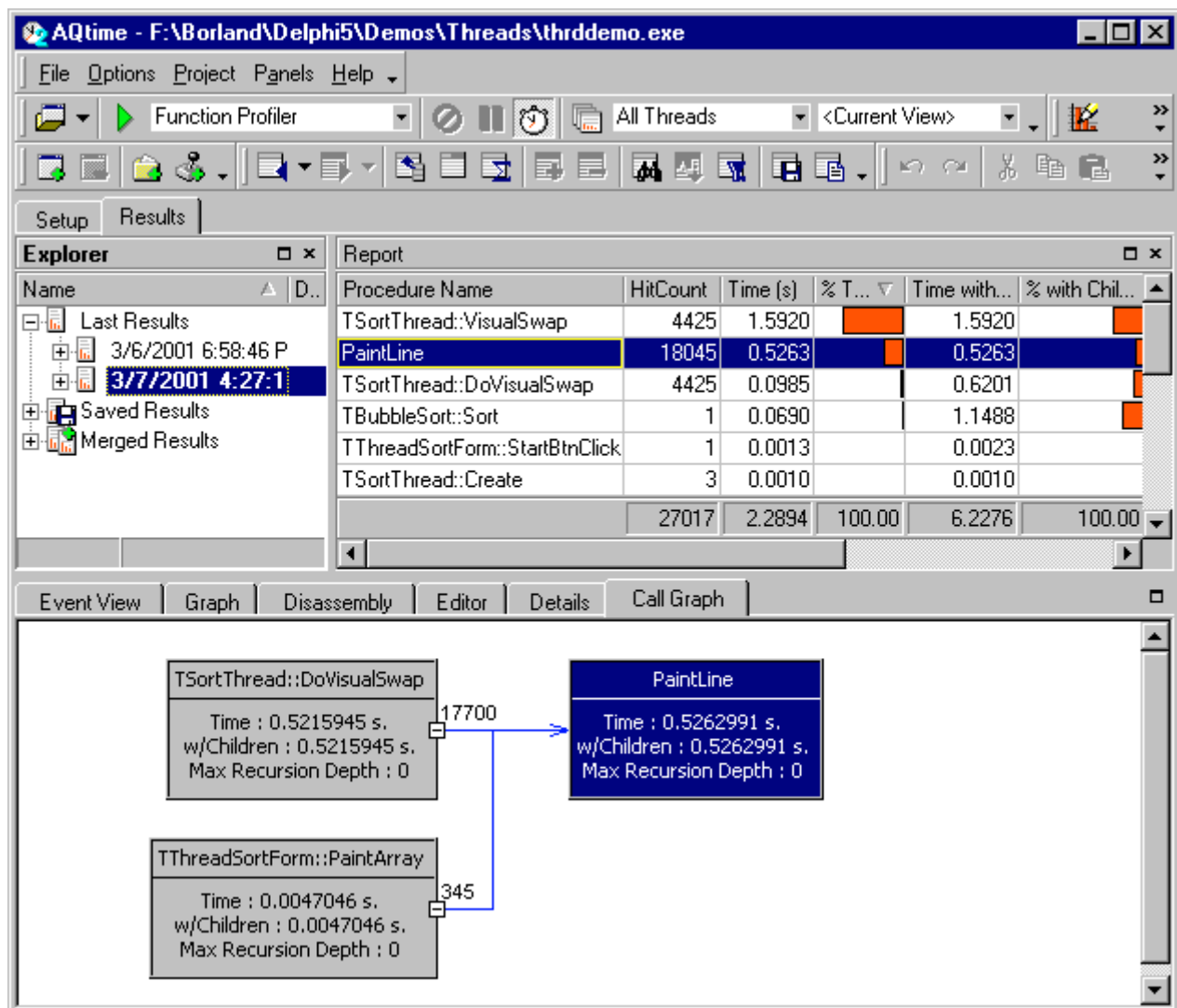
The **Function Profiler** provides much information about **calls** to the profiled functions during the profile run (see the list below). In particular:

- It **times** the execution of each call (which is what you expect of classic profiling).
- It tracks the **hierarchy** of function calls, that is, which functions ("parents") call which ("children"), i.e. which functions (parents) called other functions (children). To display this information in a convenient, graphical, browsable format, it uses the Call Graph panel, which is used by no other profiler.

As usual, the Function Profiler keeps separate track of each application thread (see *Multithreaded Applications Profiling*), and it supports Triggers (see *Using Triggers*) as well as Areas.

However, the profiler cannot track functions under five bytes in size (see *Small Function Profiling* in on-line help). And, because of the amount of information it gathers on each call, it can be slower than other profilers. For instance, if hit counts are your immediate interest, the Function HitCount Profiler will provide results quicker and in a simpler format.

Here is an example of the output of the Function Profiler:



The Report panel displays one function per line, using the following columns:

| | |
|--|--|
| Procedure Name | Full name of the function, including class name. |
| Class Name | If the function is a method, name of the class it belongs to. |
| Source File | Name of the source file for the function. |
| Unit Name | Name of the linkage unit holding the function. |
| Length | Size in bytes of the compiled code for the function. |
| Line Count | Function length in source code lines. Note that this is according to debug info and may not fit the actual source file, as the compiler uses its own rule for what is a source line. |
| Procedure Address | Function address in memory. |
| Source Line | Source file line number where the function's implementation begins. |
| HitCount | Number of times the function was called. |
| Exception (#) | Number of times the function was entered but not successfully exited. This is usually a count of exception exits, but it will also tally cases where the function was exited through a jump to some other function, instead of through the <i>ret</i> instruction (see <i>Function Instrumenting Restriction</i> in on-line help). |
| Time with Children | Total time spent on calls to this function, in seconds, including its calls to child functions. The sum for all profiled functions appears in the footer of this column. It will normally be an important multiple of actual profile-run duration, as child calls are counted several times. |
| Time (s) | Total time spent executing the function's own code, in seconds, excluding child calls. The sum for all profiled functions appears in the footer of this column. |
| Time % | Total time spent executing the function's own code, as a percentage of the time spent executing all profiled functions. |
| % with Children | Time with Children value as a percentage of the sum of Time with Children for all profiled functions. |
| Average time (ms) | Average time, in milliseconds, spent executing the function's own code on one call. This is simply Time (s) / HitCount. |
| Average with Children (ms) | Average time, in milliseconds, spent on each call to the function, child calls included. This is simply Time with Children / HitCount. |
| Max time (ms) and Min time (ms) | Maximum and minimum time, in milliseconds, spent executing the function's own code on a call. Exceptional values point out perhaps unexpected special conditions. |
| Affected by Trigger | Indicates whether Triggers turned off the profiling, so that some calls may not have been counted. |
| Max Recursion Depth | Maximum recursion depth (max number of coexisting calls within one thread) reached during the run for this function. This value is calculated only if the <i>Track recursion depth</i> option is set. |

Note that calls to child functions are only timed (and deducted from the Time (s) total) if the child functions are part of the profiling Areas. Else, they count in the execution time of the parent function ("own code"). You may mis-identify bottlenecks unless you make sure that child functions are profiled along with their parents (callers). Triggers may help you do this without profiling everything during the run. See also *Function Profiling Restriction* in on-line help.

The Function Profiler has a *Show non-hit functions* option which is off by default (see *Function Profiler Options*). This means profiled functions for which the hit count was zero are not displayed. They will be displayed if the option is on.

The usefulness of the **% with children** column is that it tells which are the expensive **calls**. A function may cost time due to its own code, or to the child calls it makes – but in any case it costs time. Often, an optimization will consist simply in making more efficient child calls – for instance, in moving a child call out of a loop. **% Time** reports the cost of the function's own code. **% Time with Children** reports the actual cost of running the function, no matter whether the cost is incurred in the function's code or in the calls it makes.

The *% with children relative to real time* Function profiler option does not change the relationship between the values in this column; the longest remains the longest and what is half as long remains half as long. With the option enabled, the figures are simply all made larger (and the column total is much above 100%). With *% with children relative to real time* enabled, 25% means that calls to the current function (and child calls) consumed a quarter of the entire profiled time. With the option disabled, the 25% would become much smaller, say 7.9%, and it would mean that calls to the current function (and child calls) consumed nearly 8% of the total time spent on any call during profiling, child calls being counted once for themselves, once more for their caller, once more for their caller's caller, etc. The column total would be 100%.

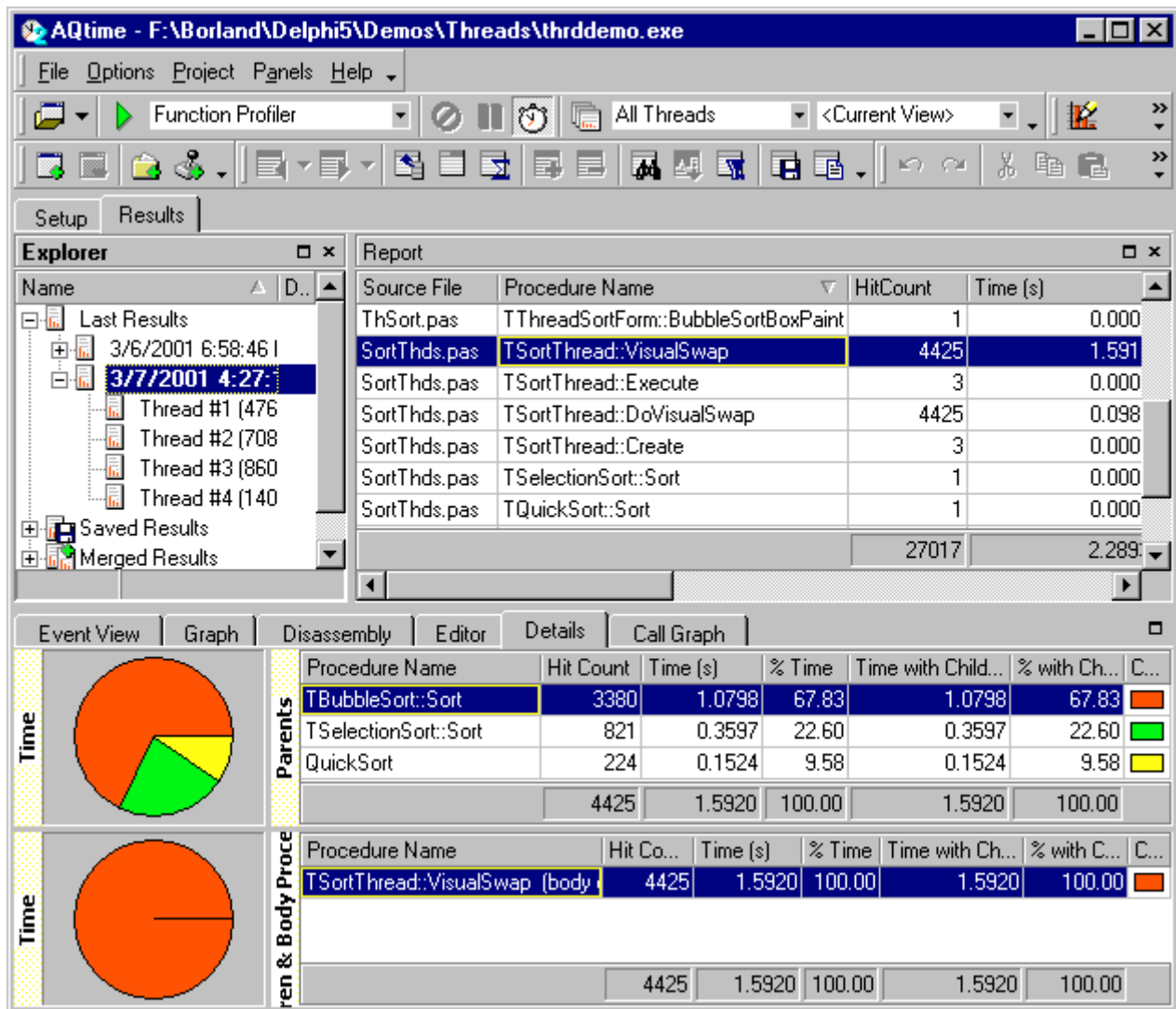
Sorting or filtering on **Time %** and **% with Children** will show up the most time-consuming functions. There is a predefined view for the Function Profiler, **Top 10 Procedures**, which filters the results to only display the ten functions that take the most time to execute their own code. You can select it from the **Views** dropdown list on the Standard toolbar (see *Views Implementation*).

Comparing **Time(s)** and **Time with Children** (or **Time %** and **% with Children**) will tell you whether the time cost of a call is due to the function called, or the functions in calls in turn.

To explore the source code of a function, double-click its line in the Report panel and switch to the Editor panel. The source can only be displayed if the source file is on the Search Path. See also AQtme Panels – the Disassembly, Call Graph and Details panels are also updated by double-clicking a line in the Report panel.

Depending on the Function Profiler Options, *Display in Editor gutter* section, various counts will appear in the Editor gutter next the start of each profiled function. This is one more means of browsing the Function Profiler results. Note that the Editor gutter can display incorrect information for some C++ programs that use several functions based on the same template (see *Template Functions Restriction* in on-line help). However, the correct profiling results are displayed in the Report panel.

The Details panel acts as a "magnifier" for parent-child call relationships related to one line in the Report panel. See *Function Profiler - Details*). Here is a sample:



Double-clicking on a line in the Details panel will update the other panels to the function displayed on that line. Switching from panel to panel in this way, in order to get the marrow out of the Function Profiler results, is made much easier by the "browser" buttons, **Back** and **Forward** on the Report toolbar.

Some routines listed in Details may have zeros in the Time, Time with Children, % Time and % with Children columns. This means that the routine was called recursively during profiling. See *Profiling Recursive Functions* in on-line help.

Function Trace Profiler

Description

The **Function Trace** profiler visually displays the sequence of function calls in real time. It is a good means to find what function is being executed at a given point in time. One good application for it is when you need to know the actual call stack for a function (e.g. a function that raises an exception). Another is when you need to know whether something occurs in the application at the expected point, for instance, whether the application posts data to the

database straight after a user has pressed OK. Another good application of Function Trace, again, is to profile an application with complex recursive calls.

The Function Trace profiler requires an extra tool that is not supplied with your AQtme installation. This can be either:

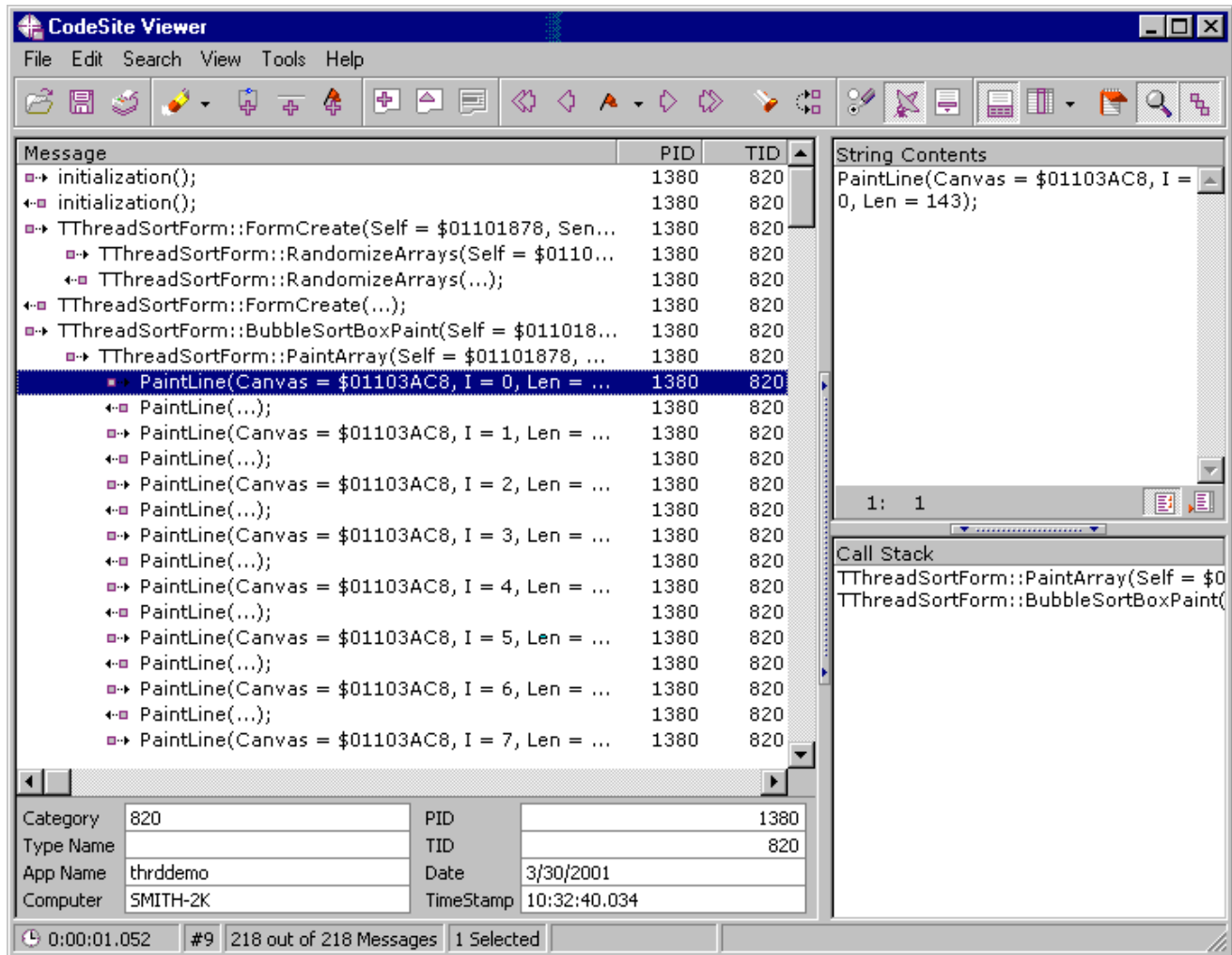
- CodeSite (<http://www.raize.com/CodeSite/>) or
- Overseer (Freeware-Open Source, <http://delphree.clexpert.com/pages/projects/nexus/download.htm>),

If you are using *Overseer*, you must initialize it before running the Function Trace profiler. You do this simply by launching then closing *Overseer* outside of AQtme.

Real-time display can be on either of these tools, or also on the Event View panel. To use the panel for this, make sure that *Event View* is enabled as one of the *Hierarchy display location* options for the profiler. See Function Trace Options.


The Function Trace profiler shows each call, with the amount of detail you specify in Options. It is not a statistical tool, but totally detail-oriented. It is very easy to both slow your application to a crawl and to generate a flood of detail, simply by letting Function Trace profile too much of the application in one run. Use Areas, use Triggers, use the System options. See Controlling What To Profile. When you take care to correctly restrict what gets profiled, you'll find Function Trace indispensable.

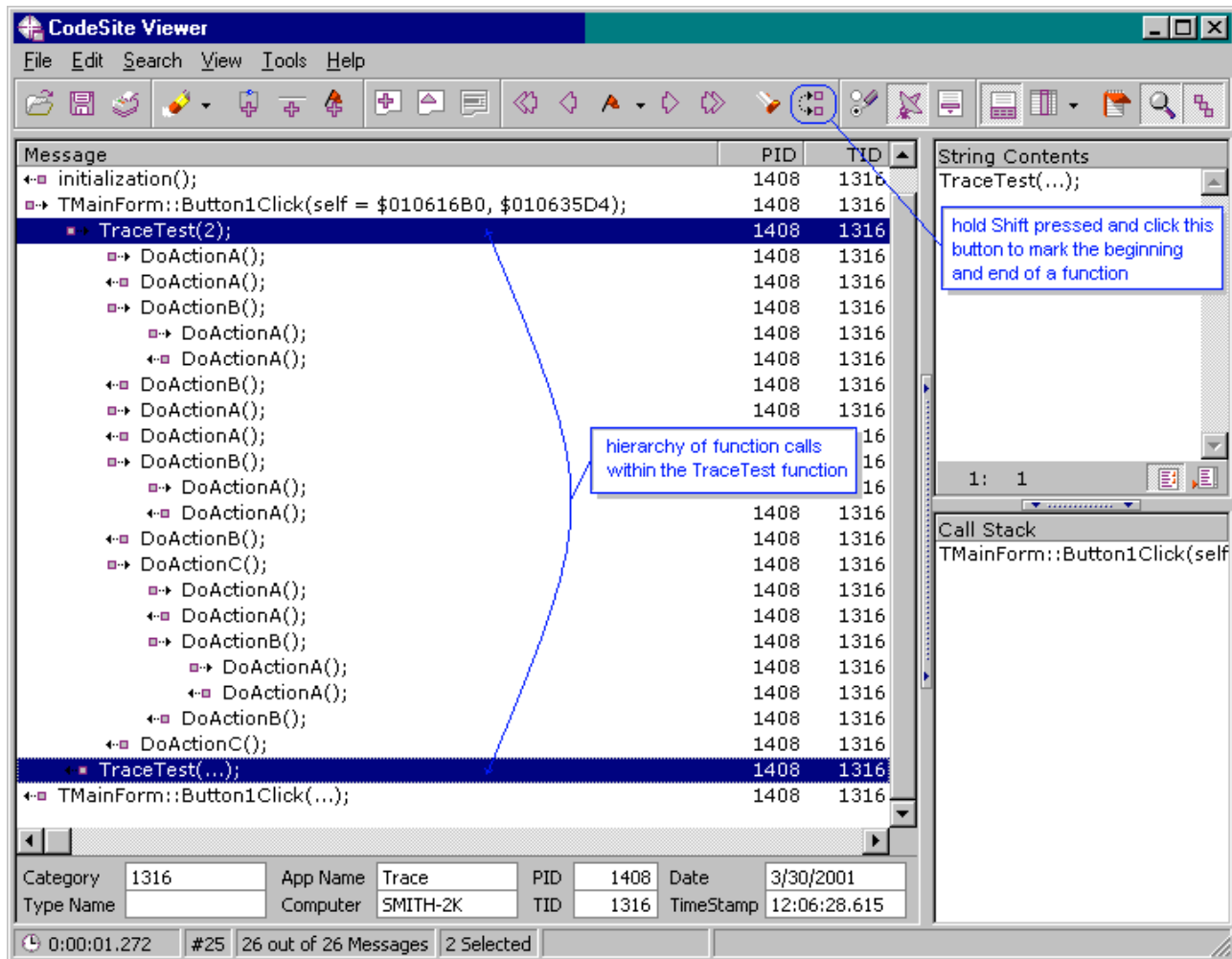
The results remain in whatever display(s) you chose, once the run is done. Using CodeSite or Overseer you will have something similar to:



For each call, two messages are displayed, one on entry and one on exit. If many child calls intervened, the two lines may be quite distant from one another. If the *Trace with parameters* option is set, the first message includes the parameter names and values, as a string. Concerning this parameter information, note the following:

- For object methods, there is always the memory address of the object itself.
- For some Borland VCL constructors and destructors an additional parameter can be displayed. It does not exist in source code but is added by the compiler.
- If the application was compiled with Optimization enabled, parameter names and values may be incorrect. We recommend that you turn Optimization off if you mean to trace parameter values on calls.
- Parameters will not be found by the profiler if the application has been compiled with debug information in PDB format (see *Compiler Settings for Microsoft Visual C++*).
- Parameters will not be shown for Win API calls since of course there is no debug information for these.
- There are other limitations too. See *Function Trace Profiler - Displaying Parameters*.

The  **Find Matching Methods** button on the CodeSite or Overseer toolbar lets you move quickly between the entry and exit messages. This is the way to easily see when any call began and when it ended (the call *bounds*).



Displaying Parameters

The Function Trace profiler can display the parameters passed on function calls, as well as the return value. However, the type of information displayed depends on several conditions.

1. For **methods** the first parameter always holds the instance address in memory.
2. Neither value nor name can be shown for the parameters of **Win API calls**. For this, you may use the Memory and API Resource Check profiler.
3. No parameter information is available for Visual C++ or Visual Basic applications using the **PDB debug info** format.
4. For Visual C++ and Visual Basic using the **DBG format**, the parameter values are shown, but not the names. Points 6 and 7 also apply.
5. For **Borland** Delphi and C++Builder, both value and name are given except where points 6 or 7 apply, or in the following cases, where the parameter values are given, but not the names –
 - VCL routines for which the compiler does not provide parameter names.
 - The last parameters of routines with a variable number of parameters.
 - The application was compiled with the *Local Symbols* option disabled. Or the present unit (.dcu) was compiled with *Local Symbols* disabled.

6. With any compiler, for parameters passed **by reference** the value is not shown, only the address.
7. With any compiler, for the following parameter **types** the value is not shown, only the parameter's address –
 - Pointers
 - Strings (except Borland VCL String and WideString types)
 - Arrays
 - Variants
 - Object types
8. Several conditions can cause a function's **return value** to be displayed incorrectly. The following cover most such cases --
 - The result value is calculated in code after the function returns. This is the case, e.g., with functions that use the Borland VCL *safecall* calling convention.
 - The result type is a 64-bit integer.
 - The result type is a Variant. In this case, the result value displayed is the Variant's address.
 - The result type belongs in the list in point 7. Here also, the value displayed will be the result's address in memory.

VCL Profilers

AQtime includes two VCL profilers, VCL Class and VCL Reference Count. The VCL (Visual Component Library) is Borland's main library for applications built with Delphi or C++Builder. These two specialized profilers analyze the use of VCL classes, report about leaked objects and help find their source (creation point). They profile the entire run, without taking any account of Areas or Triggers.


To get the full use out of the VCL profilers, you will need to recompile the Borland VCL with Stack Frames enabled. See Compiler Settings for Borland Delphi and Compiler Settings for Borland C++Builder for details. The profilers record the state of the call stack when resources are added. To trace the call stack, they assume that every call (outside System) uses stack frames. When the assumption is wrong, the errant call that caused the leak may be missing from the stack displayed in Details (depending on whether the call occurred within an event handler or within a routine called by another one in your code).

Next, to enable VCL profiling, you must first make sure the VCL libraries are in the exe itself. When compiling, either turn off Delphi's or C++Builder's **Build with runtime packages** option, or add VCL libraries to the open project. See Compiler Settings for Borland Delphi and Compiler Settings for Borland C++Builder.

The **VCL Class Profiler** (or instance profiler) logs calls to the *NewInstance* and *FreeInstance* methods of *TObject*. It reports how many instances of each class have been created in total and the peak count attained during the run. It also shows whether any object remained in memory after program termination.

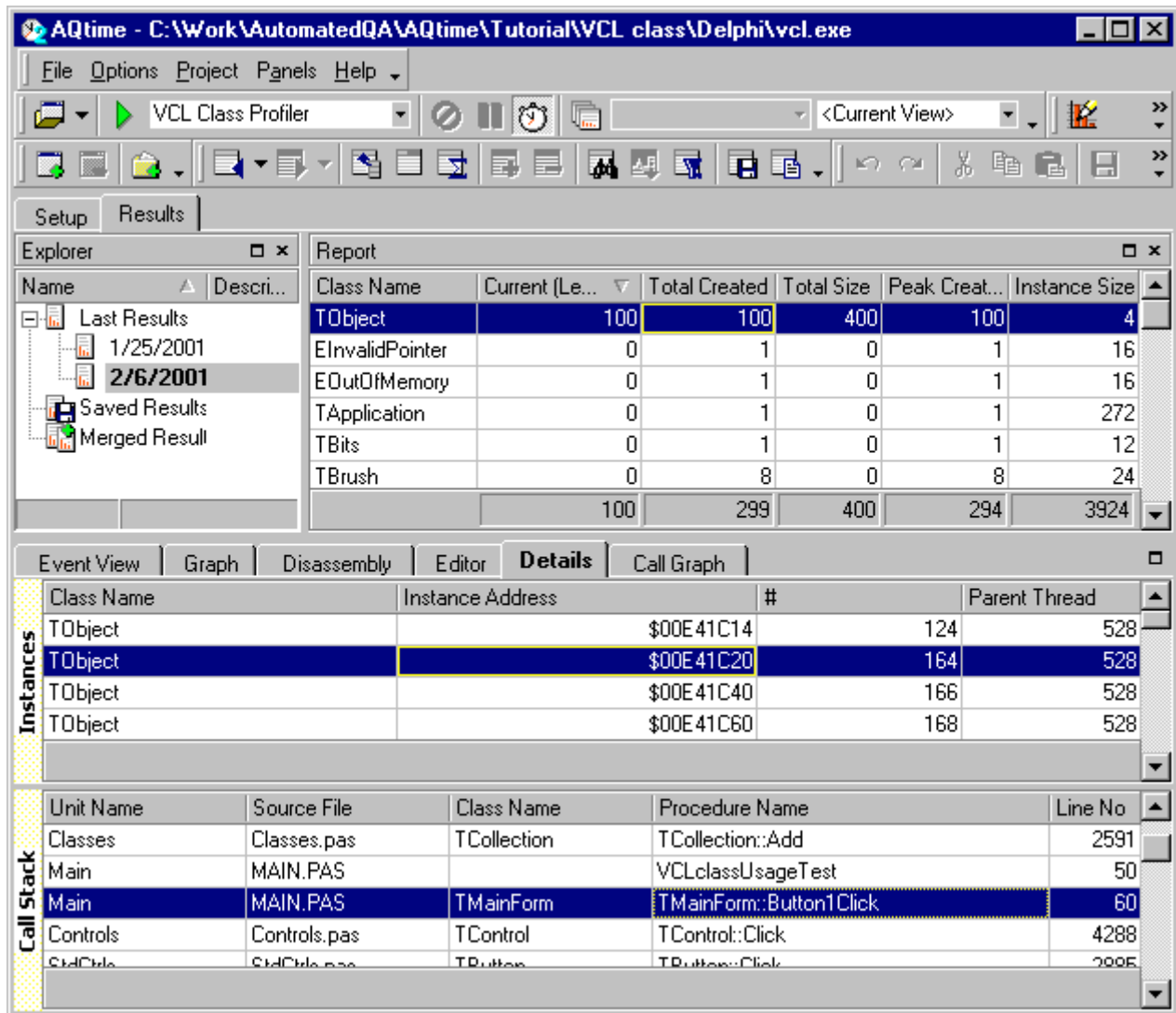
The **VCL Reference Count Profiler** (or interface profiler) logs calls to *AddRef* and *Release* and likewise reports how many references were used for each interface, in total and as a peak count for the run, and shows whether any interface had references left after program termination.

One technical note. Since by default the Reference Count profiler logs calls to the *AddRef* and *Release* methods of *TInterfacedObject*, it will miss the cases where a child class of *TInterfacedObject* overrides these methods, and the new implementations do not call the inherited methods. However, you can set the profiler to log another ancestor implementation of *AddRef* and *Release* than that of *TInterfacedObject*. This is the purpose of the VCL Reference Count Profiler's *Classes* option. (See *Profilers Options - VCL Profilers*).

While the VCL profilers ignore Areas and Triggers, they do not ignore the  **Enabled/Disabled Profiling** button. Using it, however, may very well put their create and release counts out of step, so that at the end they will report spurious "leaks". Beware! See *Leaked Objects Restriction* in on-line help.

When a VCL profiler displays its results in the Report panel, there is a line for each class that got at least one instance created, or for each interface that received at least one reference. For unreleased objects or references, the Details panel will display the call stack at that the instance was created or that the first reference was added.

Here are sample results from the VCL Class Profiler:



The screenshot shows the AQtime VCL Class Profiler interface. The main window title is "AQtime - C:\Work\AutomatedQA\AQtime\Tutorial\VCL class\Delphi\vc.exe". The interface includes a menu bar (File, Options, Project, Panels, Help), a toolbar, and a status bar. The "Results" tab is active, showing an "Explorer" panel on the left with a tree view of results (Last Results, 1/25/2001, 2/6/2001, Saved Results, Merged Result). The "Report" panel on the right displays a table of class statistics:

| Class Name | Current (Le... | Total Created | Total Size | Peak Creat... | Instance Size |
|-----------------|----------------|---------------|------------|---------------|---------------|
| TObject | 100 | 100 | 400 | 100 | 4 |
| EInvalidPointer | 0 | 1 | 0 | 1 | 16 |
| EOutOfMemory | 0 | 1 | 0 | 1 | 16 |
| TApplication | 0 | 1 | 0 | 1 | 272 |
| TBits | 0 | 1 | 0 | 1 | 12 |
| TBrush | 0 | 8 | 0 | 8 | 24 |
| | 100 | 299 | 400 | 294 | 3924 |

Below the Report panel is the "Details" panel, which is currently showing the "Instances" tab. It displays a table of instance addresses and parent threads:

| Class Name | Instance Address | # | Parent Thread |
|------------|------------------|-----|---------------|
| TObject | \$00E41C14 | 124 | 528 |
| TObject | \$00E41C20 | 164 | 528 |
| TObject | \$00E41C40 | 166 | 528 |
| TObject | \$00E41C60 | 168 | 528 |

Below the Instances tab is the "Call Stack" tab, which displays a table of call stack entries:

| Unit Name | Source File | Class Name | Procedure Name | Line No |
|-----------|--------------|-------------|-------------------------|---------|
| Classes | Classes.pas | TCollection | TCollection::Add | 2591 |
| Main | MAIN.PAS | | VCLclassUsageTest | 50 |
| Main | MAIN.PAS | TMainForm | TMainForm::Button1Click | 60 |
| Controls | Controls.pas | TControl | TControl::Click | 4288 |
| StdCtrls | StdCtrls.pas | TButton | TButton::Click | 2005 |

The Report panel uses the following columns:

| | |
|------------------------|---|
| Class Name | Name of the class. |
| Current (Leaks) | Number of instances still in memory after termination. |
| Instance Size | Size of each instance in bytes. |
| Current Size | Total size of leaks for the class, in bytes: Current (Leaks) * Instance Size. |

| | |
|----------------------|--|
| Total Created | Number of instances created during the run. |
| Total Size | Memory needed for all the instances created during the run: Total Created * Instance Size. This is not the maximum allocated for the class; see Peak Size. |
| Peak Created | Maximum number of concurrent instances reached during the run. |
| Peak Size | Maximum amount of memory allocated at once for instances of this class: Peak Created * Instance Size. |

For the VCL Class profiler there is a predefined view **Leaked classes only**, which shows only classes with leaked instances. You can select this view from the **View** box on the Standard toolbar.

To find what method created a leaked object, double-click the line for its class in the Report panel. This will scroll the Details panel to the display instances of this class.

For the VCL profilers, Details holds two horizontal sections. The topmost (*Instances* or *Interfaces*) lists leaked objects or unreleased references with their creation details. Clicking on one updates the bottom one (*Call Stack*) to show the call stack at the moment of creation – the topmost line will be the routine where the object was created or the reference added, the next line will be the routine that called it, etc. Double-clicking on a line in the Call Stack section will show the source code for this routine in the Editor panel (if it is available).). Note that the call stack is traced only if the *Stack / Show call stack* option is enabled.

The picture somewhat above this point shows Details for the VCL Class profiler. The only column needing some explanation is #. This is the instance number in the order of creation for instances of that class. The picture below shows Details for the VCL Reference Count profiler. Here, # is the reference number, in the order in which references were added for that interface. For complete information on columns in the Details panel, see *VCL Class Profiler - Details* and see *VCL Reference Count Profiler - Details*.

The screenshot shows the AQtime Reference Count Profiler interface. The top menu bar includes File, Options, Project, Panels, and Help. Below the menu is a toolbar with various icons. The main window is divided into several panels:

- Explorer:** Shows a tree view of results, including Last Results (1/25/2001, 2/1/2001, 2/6/2001), Saved Results, and Merged Result.
- Report:** A table showing reference count data for two classes: TOneInterfaceObject and TTwoInterfaceObject. The columns are Class Name, Unreleased..., Total A..., Peak Ref..., and Class
- Event View:** A table showing events for TOneInterfaceObject, including _AddRef and _Release operations. The columns are Class Name, Kind, Instance Address, #, RefCount, and Called By Thread.
- Call Stack:** A table showing the call stack for the selected event. The columns are Unit Name, Source File, Procedure Name, and Line No.

The Report panel data is as follows:

| Class Name | Unreleased... | Total A... | Peak Ref... | Class ... |
|---------------------|---------------|------------|-------------|-----------|
| TOneInterfaceObject | 1 | 3 | 3 | 16 |
| TTwoInterfaceObject | 0 | 4 | 4 | 20 |
| | 1 | 7 | 7 | 36 |

The Event View panel data is as follows:

| Class Name | Kind | Instance Address | # | RefCount | Called By Thread |
|---------------------|----------|------------------|----|----------|------------------|
| TOneInterfaceObject | _AddRef | \$00E32FBC | 1 | 1 | 1168 |
| TOneInterfaceObject | _AddRef | \$00E32FBC | 3 | 2 | 1168 |
| TOneInterfaceObject | _AddRef | \$00E32FBC | 4 | 3 | 1168 |
| TOneInterfaceObject | _Release | \$00E32FBC | 5 | 2 | 1168 |
| TOneInterfaceObject | _Release | \$00E32FBC | 12 | 1 | 1168 |

The Call Stack panel data is as follows:

| Unit Name | Source File | Procedure Name | Line No |
|-----------|--------------|--------------------------|---------|
| Main | main.pas | TForm1::FormCreate | 108 |
| Forms | Forms.pas | TCustomForm::Create | 2550 |
| Forms | Forms.pas | TApplication::CreateForm | 6692 |
| RefCount | RefCount.dpr | initialization | 19 |

Here, for the VCL Reference Count profiler the Report panel holds the following columns:

| | |
|------------------------------|---|
| Class Name | Name of the class implementing the interface. |
| Unreleased References | Number of references for this interface not released by the end of the run. |
| Total AddRef | Total number of references added for this interface during the run. |
| Peak References | Maximum number of simultaneous references to this interface reached during the run. |
| Class size | Size of the class in bytes. |

Sampling Profilers

The **Sampling** profilers poll the process for the application being profiled at regular micro-intervals to know what part of the code is executing at that instant. They provide excellent statistical information on time use within the

application, at practically zero cost on application speed. This is a very quick way to find the time hogs within the application. But very quick functions will mostly slip through the mesh unobserved.

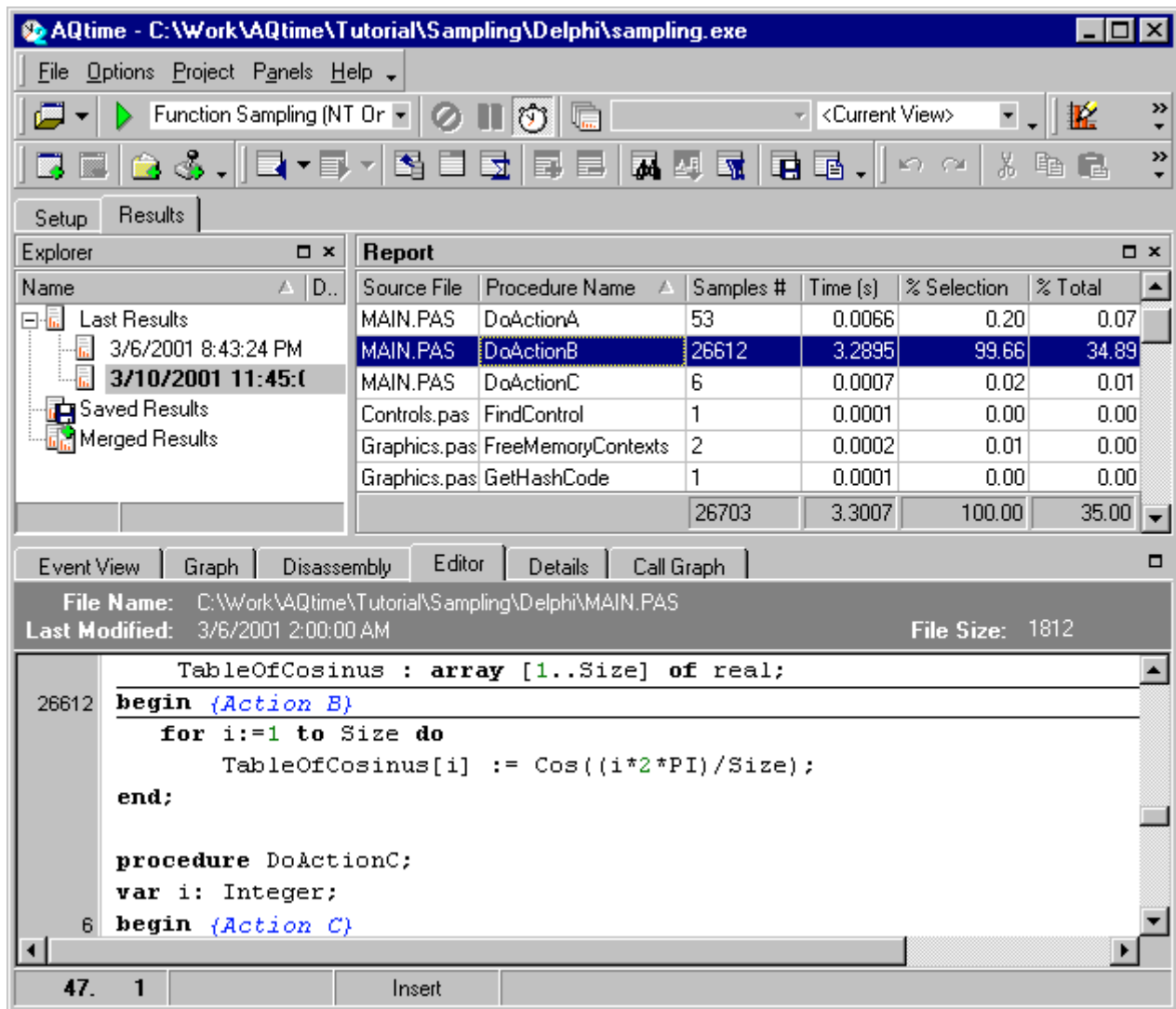
The Sampling profilers require Windows NT or 2000 to do their work. Their polling micro-interval can be changed in the Sampling Profiler Options dialog. Extending it is a way to concentrate the observations on the slowest functions. Note that when the process is not running, no observation is taken. This is the case for instance when the *sleep* WinAPI command has been given.

As sampling has almost no effect on performance, The **Function Sampling** profiler is an excellent tool to use on the FULL CHECK Area (see Setup panel). Its results will not be as accurate as those of the Function Profiler, but they will be achieved much quicker. Once poorly performing areas have been identified narrowly enough, the Function Profiler can be brought to bear on them.

The Function Sampling profiler records how often each function (above mesh size) was found running at a polling instant (the sample count for the function). The **Line Sampling** profiler does likewise for each line, but of course only the slowest lines will give statistically valid poll information.

At the end of the run, the sample count is found not only in the Report panel, but also in the gutter of the Editor, at the start of each profiled function for the Function Sampling profiler, and at the start of each line in a profiled function for the Line Sampling profiler.

Here is an example of the Report panel after a Function Sampling profile run:



The panel uses the following columns:

| | |
|--------------------------|--|
| Class Name | If the function is a method, name of the class it belongs to. |
| Procedure Address | Function address in memory. |
| Procedure Name | Full name of the function, including class name. |
| Samples # | Sample count: the number of times the function was found executing at a poll interval. The total sample count for all profiled functions appears in the footer of this column. |
| Source File | Name of the source file for the function. |
| Source Line | Source file line number where the function's implementation begins. |
| % Selection | The sample count as a percentage of the total sample count for all functions shown in the Report panel according to the current filter. |
| % Total | The sample count as a percentage of the total sample count for all profiled functions. |
| Time (s) | The sample count translated into approximate seconds. See below for the accuracy of |

the translation.

Unit Name Name of the linkage unit holding the function.

In order to translate a sample count to seconds, a calibration loop is run at the start of profiling to find how many polling intervals there are per second. The longer this loop runs, the more accurate will be the estimate (but it is still an estimate). The loop count is set as *Calibration loop count* inside Sampling Profiler Options.

Source File, **Unit Name** and **Source Line** will let you identify the source for the functions with high sample counts. You can also first isolate the ten worst simply by selecting the **Top 10 procedures** view from the **View** dropdown list on the Standard toolbar.

You can directly check in the Editor the source for any function by double-clicking on its line in the Report panel. This also refreshes the Disassembly panel to display the binary code for the function. See *AQtime Panels*.

The Line Sampling profiler lets you pursue with the same quick, easy tests *inside* the most suspect functions revealed by Function Sampling. The Report panel will display all lines in the profiled functions, for which there was at least one sample (that were running at least once at a polling moment). The following is an example:

The screenshot shows the AQtime application window. The **Report** panel displays a table of profiling data for the file `MAIN.PAS`. The table has columns: **Source File**, **Procedure Name**, **Source Line**, **Samples #**, **Time (s)**, **% Selecti...**, and **% Total**.

| Source File | Procedure Name | Source Line | Samples # | Time (s) | % Selecti... | % Total |
|-------------|----------------|-------------|-----------|----------|--------------|---------|
| MAIN.PAS | DoActionB | 48 | 579 | 0.070916 | 2.16 | 0.76 |
| MAIN.PAS | DoActionA | 40 | 63 | 0.007716 | 0.23 | 0.08 |
| MAIN.PAS | DoActionB | 47 | 7 | 0.000857 | 0.03 | 0.01 |
| MAIN.PAS | DoActionA | 39 | 4 | 0.000490 | 0.01 | 0.01 |
| | | | 26811 | 3.283801 | 100.00 | 35.16 |

The **Editor** panel shows the source code for `MAIN.PAS`. The code is as follows:

```

end;

procedure DoActionB;
const Size = 2500;
var i : Integer;
    TableOfCosinus : array [1..Size] of real;
7  begin {Action B}
579   for i:=1 to Size do
26153   TableOfCosinus[i] := Cos((i*2*PI)/Size);
3  end;
  
```

The status bar at the bottom shows the current line is 48, column 23, and the file is in Read only mode.

The panel uses the following columns:

| | |
|--------------------------|--|
| Source File | Name of the source file to which the line belongs. |
| Procedure Name | Full name of the function to which the line belongs, including class name. |
| Procedure Address | Function address in memory. |
| Class Name | If the function is a method, name of the class it belongs to. |
| Source Line | The number of the line in the source file. |
| Samples # | Sample count for the line. The total sample count for all lines in profiled functions appears in the footer of this column. |
| Sample Address | The approximate memory address for the line. See the explanation below concerning the <i>Shift size</i> option. |
| % Selected | The sample count for this line as a percentage of the total sample count for all lines shown in the Report panel according to the current filter. |
| % Total | The sample count for this line as a percentage of the total sample count for all lines in profiled functions. |
| Time (s) | Approximate total time spent executing the line, outside of any call, in seconds. The <i>Calibration loop count</i> option sets the precision of the estimate. |
| Unit Name | Name of the linkage unit holding the function to which the line belongs. |

Selecting the **Top 20** view is a quick way to focus on the twenty most time-consuming lines in the profiled functions.

There is a restriction, though: Line profilers should not be used with applications that use hooks. See *Profiling Applications That Use Hooks*.

An important parameter for the Line Sampling profiler is the *Shift size* option (see *Sampling Profiler Options* in on-line help). Line Sampling at each poll interval finds the code address currently being executed. To work faster, it can skip resolving this address down to the last bit, and assign all addresses within a binary block (4, 8, 16, ... bytes) as belonging to one line. The Shift Size is the binary exponent used, or the number of trailing bits dropped. More important, the fewest memory blocks the profiler has to take into account, the less the memory requirements.

For instance, Shift Size 4 means that addresses are resolved in blocks of 16 bytes. Lines that fit easily within these 16 bytes will be wrongly accounted – all polling that finds that block executing will count it as an execution of the first possible line. Lines that span many times 16 bytes will be correctly accounted for within statistical limits. In between is a gray area. But the point of Line Sampling is to get statistics for the slowest lines, not 16-byte or even 64-byte lines. At the cost of miscalculating lines for which, in any case, the statistics would be totally unreliable, the work of the Line Sampling profiler can be done using reasonable amounts of memory, and at good speed, even when it has many lines to poll for (the usual case).

The Sample Address column actually holds the starting addresses of the first block attributed to each line, rather than the exact starting address of line code in memory.

Shift Size also applies to Function Sampling, but its effect there is less important.

Platform Compliance Analysis

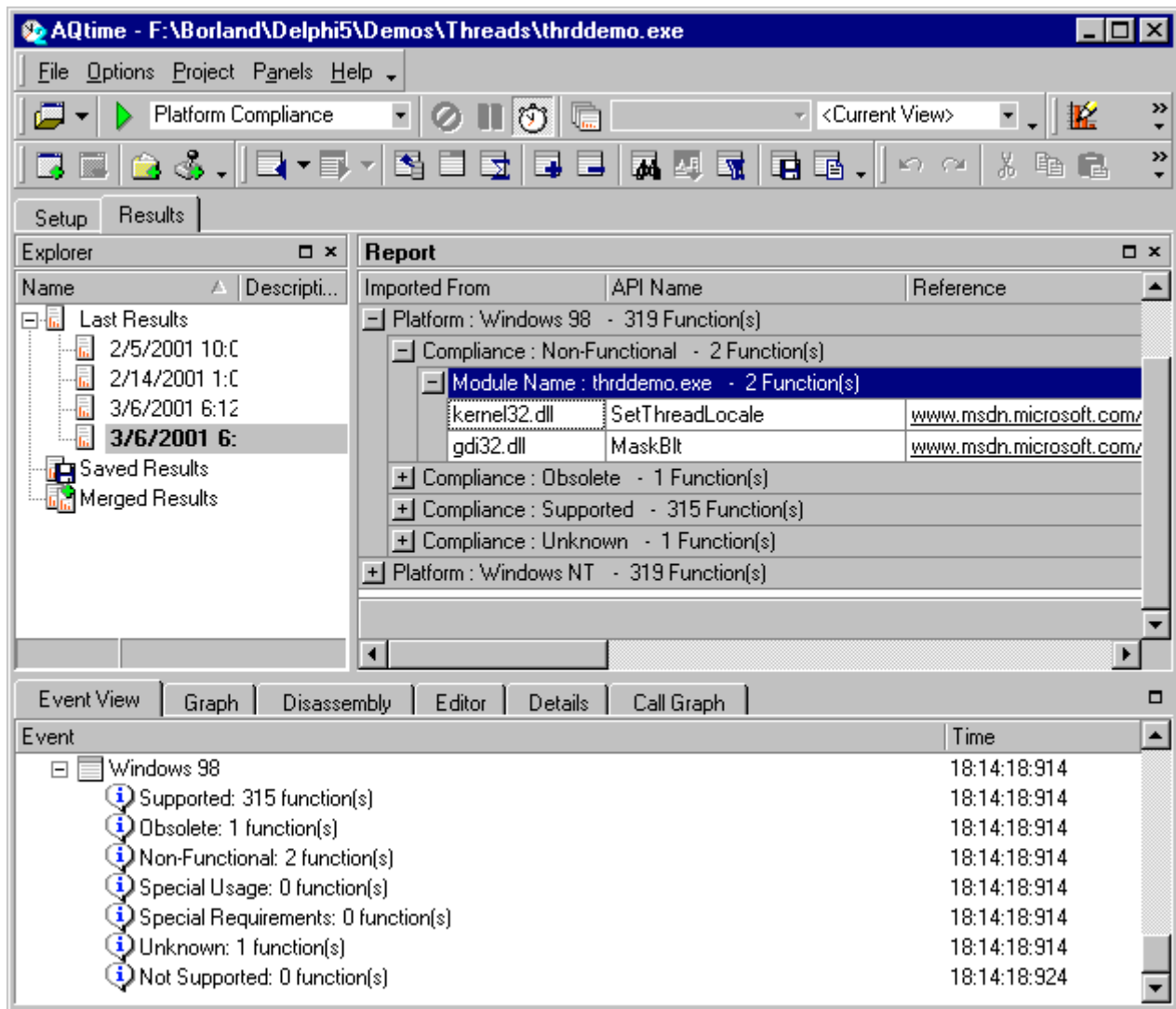
The **Platform Compliance** profiler helps determine whether the profiled application can work on a specific platform (operating system). It is a static profiler. Running it means running the profiler on the application, but not the application itself. It can be run on selected Areas if needed.

The Platform Compliance profiler is so remarkably informative, complete and easy to run (a matter of seconds) that you will likely run it on all your applications. Its one limitation is that it depends on static information. All platform calls under Windows are DLL calls. DLLs can be statically linked, i.e., be called using addresses defined at compile time, or dynamically linked, that is, be called using addresses found at runtime only, and especially from DLLs that are likely to change. Platform Compliance cannot check dynamically defined calls.

This limitation especially affects profiling **Visual Basic** applications. In VB, statically linked calls are those defined through the DECLARE statement. If your VB application uses only the MSVBVM50 (Visual Basic 5.0) or MSVBVM60 (Visual Basic 6.0) libraries and uses no DECLARE of its own, Platform Compliance Analysis will yield no information at all in the Report pane.

Statically linked calls are said to be *exported*, and their target functions to be *imported* at runtime from system libraries which *export* them (make them available to external calls). The **Platform Compliance** profiler analyzes all exported calls that address system libraries (i.e. DLLs), and checks which platforms' libraries will support the call, and how -- Windows 95, Windows 98, Windows NT, Windows 2000, Windows CE or WINE. This analysis is done against a database of compliance information that is part of the AQttime installation. You can update the database using the Win API Database Editor – an additional tool available at our web site (www.automatedqa.com).

Here is an example of Platform Compliance Analysis output:



The Report panel holds the following columns:

| | |
|----------------------|---|
| API Name | Function name according to the system API (Application Programming Interface). |
| Module Name | Name of the application module (.exe, .dll, etc.) which calls the function. |
| Imported From | Name of the system DLL (may or may not be *.dll) which exports the function. |
| Platform | Platform (OS) name. |
| Compliance | Compliance of the call for this platform. This column can display the following values: <ul style="list-style-type: none"> <i>Supported</i> The call is correctly supported. <i>Unsupported</i> The function is absent from the DLL for this OS. When loading the DLL, the application will display an error message. <i>Obsolete</i> The function is present and active, but obsolete. Using it is not recommended by the platform maker. |

Special Requirements The call will be supported if some additional software is present on the machine, as listed in Notes. If the software is absent, the compliance status is *Unsupported*. For instance, some calls to Windows NT 4.0 require Service Pack 4.0 or later.

Non-Functional Though MSDN reports this function as "unsupported", it is present and will prevent an error message from being posted. However, calls to it will do exactly nothing.

Special Usage The function is present and active, but it is not fully functional. E.g. it may ignore some parameters.

Unknown AQttime does not have any information about the function.

| | |
|------------------|--|
| Notes | Additional information about compliance conditions. |
| Ordinal | Ordinal number of the function taken from debug information. If the debug information does not provide it, the value shown is 0. |
| Reference | Hyperlink to the on-line function documentation. To open it, simply double-click on the cell. |

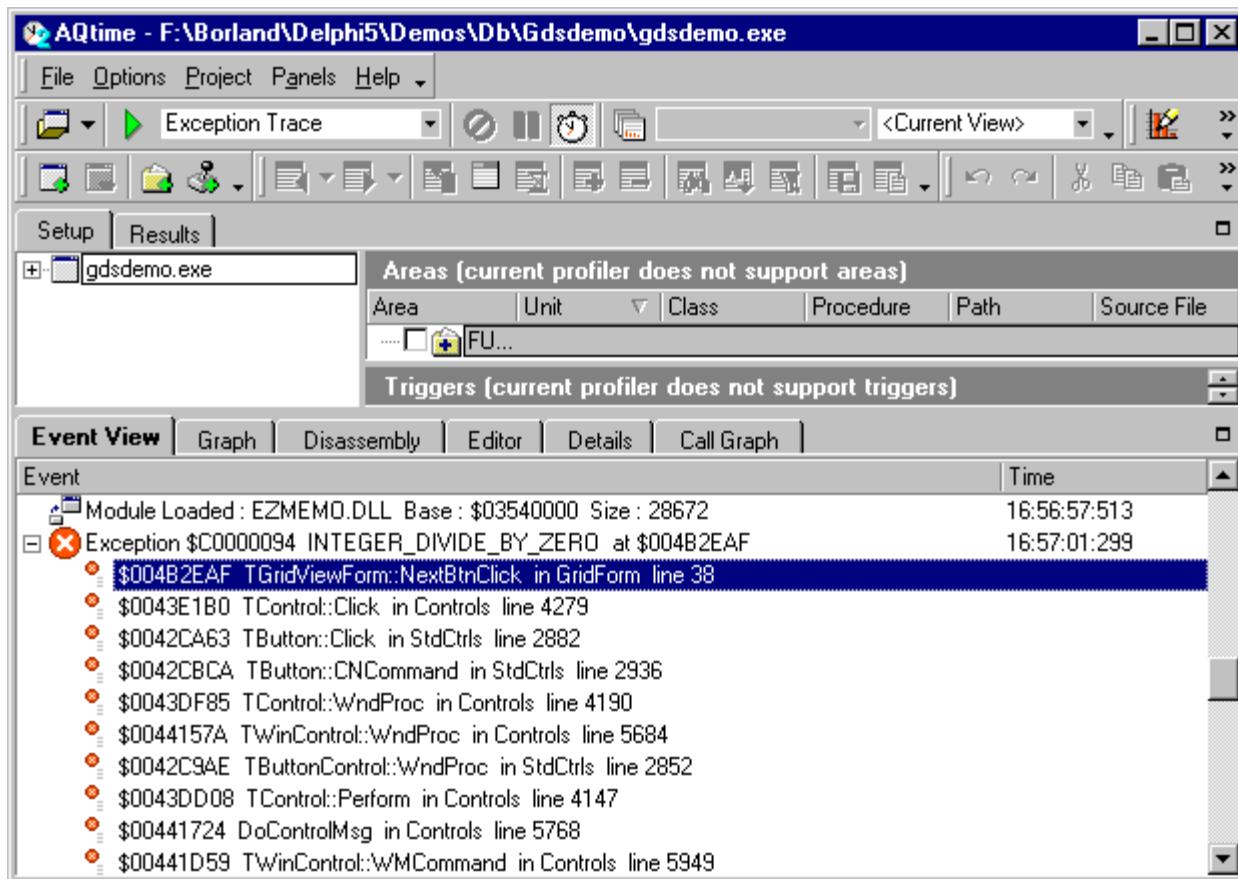
The Platform Compliance profiler uses a special type of filtering that is set before data collection. Calls with a certain compliance status can be eliminated from the results for clarity. This pre-filtering is set by the Platform Compliance Settings dialog. If the *Customize settings before profiling* option is enabled (in Profiler Options) the dialog will be displayed before starting each run of the Platform Compliance profiler.

Warning-type compliance values are always included in the results, to wit, *Non-Functional*, *Special Usage*, *Special Requirements* and *Unknown*. Other compliance categories can be filtered out through the first two settings in the dialog:

| | |
|---------------------------------|--|
| <i>Obsolete and Unsupported</i> | Supported functions are removed from results. Functions of other categories remain. |
| <i>Only Unsupported</i> | Supported and Obsolete functions are removed from results. Functions of other categories remain. |
| <i>Full Analysis</i> | All compliance categories are included in results. |

Exception Tracer

The **Exception Trace** profiler monitors the application execution and if an exception occurs it outputs exception information (exception type, address, call stack, etc.) on the Event View panel. This profiler works faster in comparison with other profilers and does not slow down the tested application greatly. Use it if you need only to explore exceptions that occur during the application execution.



Note that you can view the source code of a routine in the call stack: Simply double-click the routine in the Event View panel and then switch to the Editor one.

Since this profiler uses the Event View panel, the *Exceptions / Active* option of this panel must be turned on. Else, the Exception Trace profiler displays an error message and does not start profiling. The other Event View options specify what information will be displayed when the exceptions occur. See *Event View Options*.

Unused VCL Units Profiler

Unused VCL Units Profiler – Description

When you add a unit to the *uses* clause, the Delphi linker will always include this unit into the executable file, even if a program does not use any procedures from this unit. This can occur when you drop a component onto a form to have a look at it and then you delete the component from the form. The Delphi IDE does not remove the component's unit from the *uses* clause.

Delphi includes an optimizing linker that will not link any functions from this unit if they are not referred to from other sections of your code. However, each unit has *initialization* and *finalization* procedures generated by the Delphi compiler (they are generated behind the scenes even if you do not create those sections in source code). These procedures are used to execute the unit's initialization and finalization code and to clear global unit reference-count variables when exiting the program.

AutomatedQA's **Unused VCL Units** profiler scans all units that are included into a project and checks whether they are used in an application or not. Upon finishing these operations, the profiler displays a list of units that are

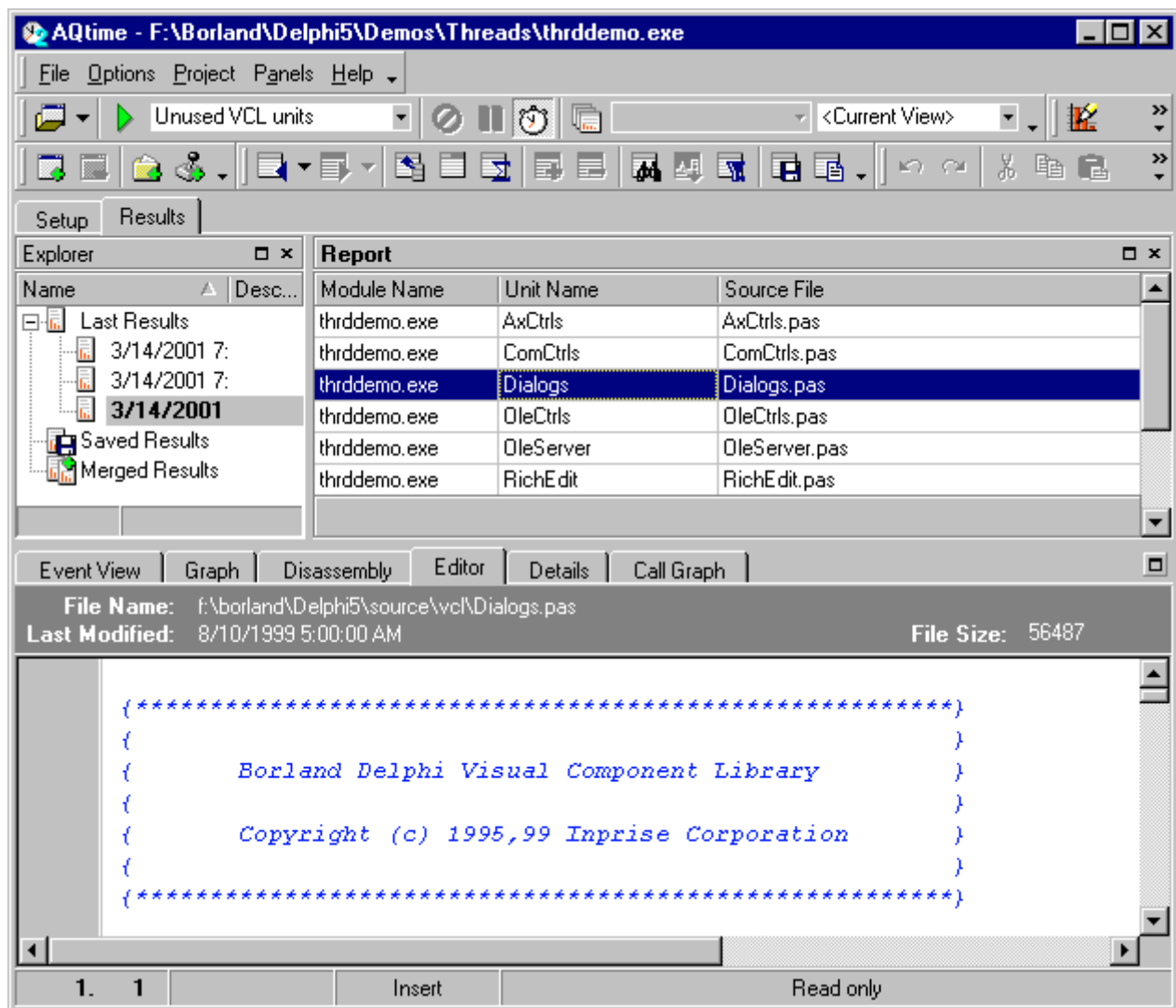
not used in the project. For each unused unit it also provides an additional list that tells you which application units refer to the unused one. On the basis of these lists you can decide if you want to remove a unit or not. For more information on how to work with the profiler, see *Unused VCL Unit Profiler – Principles of Operation*.

Since units use one another, you should run the Unused VCL Units profiler several times to remove all unused modules. Please review *Unused VCL Units Profiler – Principles of Operation*.

This profiler ships with full source code and a sample application to allow AQtime users to have a look at how to create a custom profiler that iterates through the project's units and procedures. The sample application is located in the directory wherein the Unused VCL Units plug-in is installed.

Note that the Unused VCL Units profiler does not analyze the application type before running. That is, profiling of non-VCL application will produce no results.

Here is an example output from the Unused VCL Units profiler:



The Report panel displays a list of "doubtful" units (the profiler "thinks" they are not used in the application). The Details panel holds a list of application units that refer to the one selected in Report. These are units from which you can remove the reference to the selected one. Both panels include the following columns -

| | |
|--------------------|---|
| Unit Name | The name of an unused unit. |
| Module Name | The name of the module (EXE, DLL, etc.) that uses the unit specified in the Unit Name column. |
| Source File | The source file name for a unit. |

To explore the referring files or source code of a unit, simply double-click its name in the Report panel and then switch to Details or to the Editor panel.

Unused VCL Units Profiler – Principles of Operation

The Unused VCL Units profiler is a static analysis profiler. It helps you find units that are included in your application, but are not used by your application. If a unit is not actually used by the application, its *initialization* and *finalization* code is called only.

The Unused VCL Units profiler includes database that holds information about the number of procedures used by *initialization* and *finalization* sections of each standard VCL unit. This profiler compares the number of procedures exported by a unit with the number specified for this unit in the database. If the unit exports more functions than the database indicates, the profiler regards it as a used unit. If the number of exported procedures is equal to the number specified in the database, the unit is included into the list of the unused ones.

However, it is not possible to determine with 100% accuracy what units should be removed from your application. For instance, if a unit holds only constants and variables used by the application, and it does not export any procedures, you do not have to remove it from the application. Some more examples: unit A uses only class types declared in unit B; Or unit A uses a class declared in unit B and this class contains only inherited methods and does not define its own ones. In both cases, the profiler will report that unit B is not used in unit A while it is used indeed. The Unused VCL Units profiler includes a specific option, *Unused units with names containing*, that allows you to exclude such units from analysis. See *Unused VCL Units Options*.

You can use an alternate method of excluding a unit from analysis: Add the unit name to the IgnoredUnits.txt file that ships with the Unused VCL Units profiler. This file holds three lists of units - for Borland Delphi ver. 3, 4 and 5. All units, listed in the file, are considered used.

To find units that refer to the unused one, the profiler scans the source code (namely, the *uses* section) of each application unit. If a unit refers to the unused one, it is added to Details. That is why the Unused VCL Units profiler must "know" paths to all units in the application. To specify the search path, use the Search Directories or Project Search Directories dialogs.

Units import one another, so you must execute the Unused VCL Units profiler several times to remove unused modules from your application. For example, your project contains two unused units - *Buttons* and *Graphics* (standard VCL units). The *Buttons* unit uses some procedures from the *Graphics* one. So, the number of exported procedures for *Graphics* is greater than the number specified for this unit in the database. Therefore, the Unused VCL Units profiler considers the *Graphics* unit to be used. After the first launch, the profiler will report that only the *Buttons* unit is unused. Remove this unit from your project, recompile the application and return to AQtime. After the second launch, the profiler will report that the *Graphics* unit is not used.

The number of procedures in the *initialization* section varies from one Delphi version to another. The Unused VCL Units profiler works with Borland Delphi ver. 3, 4 and 5. To specify which Delphi version was used to build an application, set an appropriate value for the *Delphi version* option before profiling.

In addition to standard VCL code, the Unused VCL Units profiler can analyze user-defined units. It includes two options, *Initialization lines discarded* and *Initialization bytes discarded*, used to profile custom units. See *Unused VCL Units Options*.

ATL Reference Count Profiler

ATL Reference Count Profiler – Description

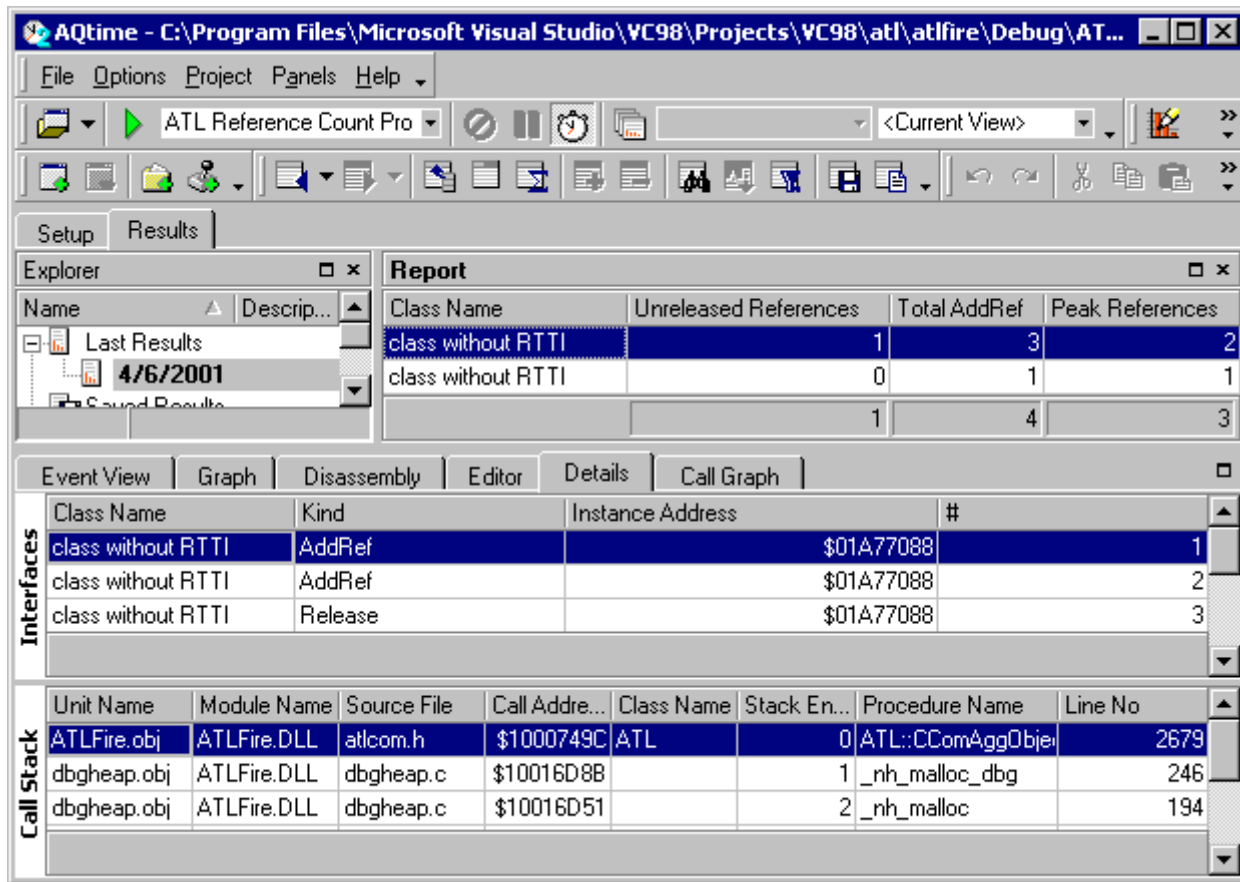
The **ATL Reference Count** profiler analyzes the usage of ATL classes and COM objects in applications using the Active Template Library, built with Microsoft Visual C++ or Borland C++Builder. Before launching this profiler, make sure the executable is compiled with Run-TimeType Information. See ATL RefCount Profiler- Compiler Settings.

The ATL Reference Count profiler is similar to the VCL Reference Count profiler. It determines how many references of each interface have been used (total and peak at any one time) and tracks whether certain objects stay in memory after application termination because of unreleased references from the application.

The ATL Reference Count profiler tracks calls to the *AddRef* and *Release* methods of classes inherited from the following ATL classes. (For more information on these classes, refer to the *ATL Library Help*.)

- CComObject
- CComAggObject
- CComPolyObject
- CComTearOffCachedObject
- CComTearOffObject
- CComObjectNoLock (for more information on these classes, refer to the *ATL Library Help*).

The profiler always analyzes the entire run, independent of any Areas or Triggers. Here is an output sample:



The Report panel displays all classes created during the run. It holds the following columns:

| | |
|------------------------------|--|
| Class Name | Name of a class using an ATL interface. |
| Unreleased References | Difference between calls to <i>AddRef</i> and to <i>Release</i> , for the class. |
| Total AddRef | Total number of calls made to <i>AddRef</i> during the run. |
| Peak References | The maximum number of simultaneous references reached during the run. |

Unlike Report, the Details panel only displays information about unreleased interfaces. It includes two panes: The upper one, *Interfaces*, holds a list of **calls** to the *AddRef* and *Release* methods of a class. It has the following columns:

| | |
|-------------------------|---|
| Class Name | Name of a class with an interface leak. |
| Kind | Call: <i>AddRef</i> or <i>Release</i> . |
| Instance Address | The address in memory of the unreleased object. |
| # | The creation number of this instance for the class. The first instance constructed is 1, the second 2, etc. |

If you have enabled *Show call stack* in the Profiler Options, you can view the **call stack** at the moment of any call listed in the Interfaces pane. Double-click on it and switch to the lower pane, *Call Stack*. The topmost row lists the method that made the call to *AddRef* or *Release*, the next row - the caller for the top-row method, and so forth. The available columns are:

| | |
|-----------------------|--|
| Procedure Name | Method, function or procedure. Called <i>Procedure</i> below. |
| Class Name | Class holding <i>Procedure</i> if it is a method. |
| Module Name | Executable module holding the binary code for <i>Procedure</i> . |
| Unit Name | Unit where <i>Procedure</i> is defined. |
| Source File | Source file holding the implementation for <i>Procedure</i> . |
| Line No | Line on which the implementation for <i>Procedure</i> begins in <i>Source File</i> . |
| Stack Entry No | <i>Procedure</i> 's index in the call stack. The topmost routine has index 1. |

Double-clicking on a row in the Call Stack pane will update the appropriate panels to show this routine – Editor, Disassembly, Graph, etc. Switching to the Editor will give you the source for the routine, and so forth.

You can arrange the Report and Details panels in the usual manner.

ATL RefCount Profiler – Compiler Settings

To be profiled with the ATL Reference Count profiler, an application must meet two requirements. First, the application must be compiled with debug information (see *Preparing a Project For AQtune*). Second, it must be compiled with Run-Time Type Information (RTTI). If a tested application does not include RTTI, the ATL Reference Count Profiler cannot determine the names of classes used. Profiling results will hold “Class without RTTI” in place of the class name.

This topic explains how to include RTTI in the executable:

Microsoft Visual C++

1. Open your application in Visual C++.
2. Call the **Project Settings** dialog (press ALT-F7 or select **Project | Settings...** the main menu).
3. Then, switch to the **C++** tab and choose the **C++ Language** category from the dropdown list.
4. Turn on the Enable Run-Time Type Information (RTTI) option.
5. Rebuild your application.

Borland C++Builder

1. Open your application in C++Builder.
2. Choose **Project | Options...** from C++Builder’s IDE. This will call the Project Options dialog.
3. Select the **C++** tab.
4. Check **Enable RTTI** in the **Exception handling** section.
5. Rebuild your application.

BDE SQL Profiler

The execution speed of a database application depends directly on the speed of SQL queries. The **BDE SQL Profiler** times the execution of SQL queries and SQL stored procedures when commanded through the Borland Database Engine.

The BDE SQL Profiler works with applications compiled with Borland Delphi v. 2 - 5 and C++Builder v. 3 - 5. It tracks calls to the *CreateCursor* and *Prepare* methods of the *TQuery* object and calls to the *ExecProc* and *Prepare* methods of the *TStoredProc* object (*CreateCursor* is called from the *Open* and *ExecSQL* methods).

Here is an example of the BDE SQL Profiler output:

The screenshot shows the AQtime - C:\Work\Tests\BDESQTest\IntraDB.exe window. The BDE SQL Profiler is active, displaying a report of database operations. The report table lists the following data:

| Class Name | Operation Type | Name | Time(s) | SQL expression |
|-------------|----------------------|-----------------|----------|--------------------------------|
| TQuery | TQuery.CreateCursor | AddCntryQuery | 0.00000 | Insert Into Country Values('Ne |
| TQuery | TQuery.CreateCursor | CalcBudgetQuery | 0.00444 | SELECT P1.Proj_Name, SUM |
| TQuery | TQuery.Prepare | GeneralQuery | 3.87167 | Select Emp.First_Name, Emp |
| TQuery | TQuery.CreateCursor | GeneralQuery | 0.00082 | Select Emp.First_Name, Emp |
| TStoredProc | TStoredProc.Prepare | StoredProc1 | 0.00000 | |
| TStoredProc | TStoredProc.ExecProc | StoredProc1 | 9.60653 | |
| | | | 13.48347 | |

Below the report, the SQL expression for the selected query is shown:

```
SELECT P1.Proj_Name, SUM(P2.Projected_Budget) AS Budget
FROM Proj_dept_Budget P2, Project P1
WHERE P1.Proj_ID = P2.Proj_ID
GROUP BY P1.Proj_Name
```

The Call Stack panel at the bottom shows the sequence of calls:

| Source File | Class Name | Procedure Name | Line No |
|--------------|-------------|-------------------------------|---------|
| DBTables.pas | TBDEDataSet | TBDEDataSet::OpenCursor | 4022 |
| DBTables.pas | TDBDataSet | TDBDataSet::OpenCursor | 6041 |
| Db.pas | TDataSet | TDataSet::SetActive | 8242 |
| Db.pas | TDataSet | TDataSet::Open | 8203 |
| MainUnit.pas | TForm1 | TForm1::RunCalcBudgetBtnClick | 60 |
| Controls.pas | TControl | TControl::Click | 4288 |

The Report panel displays information about the called methods using the following columns:

- Operation Type** Type of the DB operation. One of the following:
- TQuery.CreateCursor (this operation is performed within TQuery Open or ExecSQL methods)

| | |
|-----------------------|---|
| | <ul style="list-style-type: none"> – TQuery.Prepare – TStoredProc.ExecProc – TStoredProc.Prepare |
| Name | Name of the query or stored procedure. |
| Class Name | Class name for the query or stored procedure. Normally this is <i>TQuery</i> or <i>TStoredProc</i> . |
| Time(s) | Execution time in seconds the query or stored procedure. This does not include fetch time. |
| SQL expression | SQL code executed. This field is empty for stored procedures. |

In addition to the Report panel column, the executed SQL expression is displayed in the upper part of the Details panel called *SQL*, with color syntax highlighting.

If the *Show call stack* option is on, the BDE SQL profiler tracks the call stack for the analyzed methods and displays it in the lower pane of Details, named *Call Stack*. The topmost line is the top of the stack, that is, it shows the method that called the BDE operation currently selected in the Report panel. Each line below shows the caller of the line above it. The *Call Stack* grid uses the following columns:

| | |
|-----------------------|---|
| Procedure Name | Method, procedure or function name. |
| Class Name | Class name if <i>Procedure</i> is a method. |
| Module Name | Name of the executable module (exe, dll, etc.) where the binary code for <i>Procedure</i> ran from. |
| Unit Name | Name of the source unit for <i>Procedure</i> . |
| Source File | Name of the source file for <i>Procedure</i> . |
| Line No | Line number, in <i>Source File</i> , where the implementation of <i>Procedure</i> begins. |

To view the call stack for a method from the Report panel, simply double-click it there and switch to Details. To view source code for any routine in Call Stack, double-click it and open the Editor Panel.

Memory and API Resource Check Profiler

General Overview

This profiler tracks the current project, monitoring each call to API functions and memory allocation. It performs the following tasks:

- Checks whether the profiled application creates resources correctly and releases all allocated resources. Resource leaks are traced only for those functions that work with pens, brushes, pictures, icons, bitmaps, fonts, etc. For the full list of the checked GDI functions, see *List of Checked Functions*.
- Analyzes the use of the memory management functions, i.e.
 - *GetMem*, *FreeMem*, *ReallocMem*, *New* and *Dispose* (Delphi and C++Builder functions);
 - *malloc*, *calloc*, *realloc*, *expand* and *free* (C++ functions);


- *new* and *delete* (C++ operators).

These functions are traced in all situations (for instance in constructors and destructors). The full list of functions checked for memory leaks is in *List of Checked Functions*. If a resource or a memory block was not disposed of, AQtime inserts a record into the Report panel. This record holds information about the function that allocated the block.

Traces whether your application uses allocated memory blocks correctly, that is, whether all memory writes are within bounds of currently allocated blocks, and whether no call to *free* applies to a currently unallocated block.

- Checks parameter values and return value for all resource-related API calls. Further calls can be checked for by modifying the database used by the profiler. The AutomatedQA web site (www.automatedqa.com) offers an extra tool to allow you to do this, the **Win API Database Editor**.

The Memory and API Resource Check profiler is supplied in your AQtime installation as an optional plug-in. It adds a Leak Filters page to the Create Filter Condition dialog. It also supplies a number of predefined Views for its results (see *Memory and API Resource Check – Description of Results*). Used with the **Real-Time Monitor** plug-in, the profiler can display its results during the run, as they occur.

Memory and API Resource Check can only operate over the entire run of the application. It takes no account of Areas or Triggers, and disables the  **Enable\Disable Profiling** button.

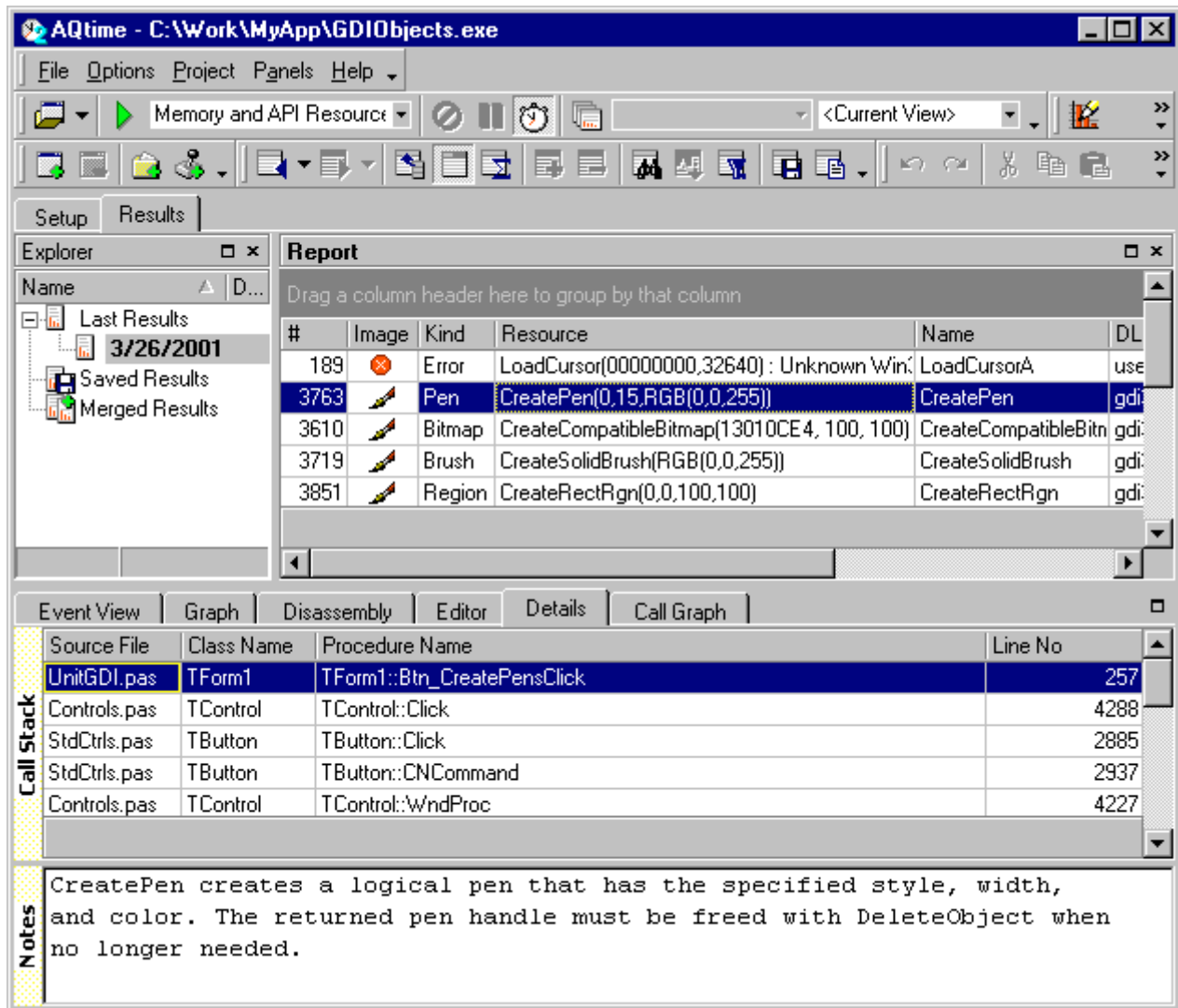
Profiler results and columns of the **Report** and Details panels are explained in Memory and API Resource Check – Description of Results. Some additional notes about VC++ application profiling are in *Profiling VC++ Applications With Memory and API Resource Check*. Resources or memory blocks may sometimes be reported as unreleased when they are released. This depends on the order in which the DLLs are loaded and unloaded from memory. See *Leak Resources Restriction*.

Description of Results

The Memory and API Resource Check profiler is installed in the **WinAPI Analysis** group of AQtime profilers. On launching it displays the Settings dialog, where you can specify modules and function types to be profiled. (This can be turned off by disabling *Customize settings before profiling* option).

Memory and API Resource Check will analyze only modules selected in the **Modules** page of the dialog. When you run the profiler for the first time, this page holds only modules specified in the Setup panel. In the course of the profile run AQtime detects modules (EXEs, DLLs or OCXs) being loaded into the memory and adds them to the page. Another page of the Settings Dialog - **Hook Extensions** - specifies what function types will be profiled.

Below is an example of the Memory and API Resource Check profiler output:



The Report panel includes the following columns:

| | |
|------------------|--|
| # | Resource ID in creation order (earlier allocations get lower ID numbers). Errors are counted as resources. |
| Name | API function called. |
| DLL Name | Library the function is exported from. |
| Kind | Function category. The profiled function categories are selected in the Settings dialog. |
| Image | Icon for the category. E.g. indicates errors of all types. |
| Reference | Hyperlink to the MSDN topic concerning the function. To open the topic, double-click the hyperlink. |
| Resource | Function name with actual parameters of the function call. |
| Size | Size of the memory block. This column holds 0 for the other types of resources. |
| Thread Id | Identifier for the thread where the function was called. |

Value A value that is associated with the resource: For GDI resources (pen, brush, bitmap, etc.) this is the handle to the GDI resource, for memory blocks this is the address of the block in memory, etc. This column holds 0 for errors and warnings.

The Details panel holds two panes: **Call Stack** and **Notes**. These panes display additional information about the function selected in the Report panel. The Notes pane contains a short description of the function.

If the *Stack / Show call stack* option is on, the Call Stack pane of Details displays the sequence of function calls that lead to calling the function. Note that Memory and API Resource Check can trace the call stack using two different methods depending on the *Stack / Rely on stack frames* option. The Call Stack pane holds the following columns:

| | |
|-----------------------|---|
| Procedure Name | Function or procedure called. |
| Class Name | Class the function belongs to if it is a method. |
| Unit Name | Unit the function is declared in. |
| Module Name | Module (EXE, DLL or OCX) where the function is located. |
| Source File | Source file holding the function code. |
| Line No | Line at which the function code begins in the source file. |
| Call Address | Function address in memory. |
| Stack Entry No | Function position in the stack. The function with index 1 is the immediate caller of the function selected in the Report panel. |

Click a function in the Report panel to select it and view its call stack and description in Details. Switch to the Editor panel to view the source for the selected function, and to the Disassembly panel to view its binary code. When you click a function in the Call Stack pane of Details, AQtime will automatically select this function in the Report panel and change the contents of the Disassembly and Editor panels.

The Memory and API Resource Check plug-in adds the Leak Filters page to AQtime's Create Filter Condition dialog. This lets you select results for viewing depending on criteria which only apply to this profiler. A number of predefined filters are included to help you separate errors in the profiled application from those that occur in VCL or MFC code.

Along with its custom filters, the profiler includes a selection of predefined Views, which include both a filter and a Report panel column layout:

- Filter known leaks
- Filter errors
- Filter warnings
- Filter errors and warnings
- Filter memory leaks
- Default

To apply a View, select it from the **Views** drop-down list box on the Standard toolbar.

Resources or memory blocks may sometimes be reported as unreleased when they are released. This depends on the order in which the DLLs are loaded and unloaded from memory. See *Leak Resources Restriction*.

Profiling VC++ Applications

The Memory and API Resource Check profiler tracks functions that allocate or deallocate memory. There are two ways functions can do this. They can use **system** memory management calls, or they can call on the **runtime** memory manager that is part of the runtime library of VC++ and other tools. A runtime memory manager requests large blocks from the system, and eventually releases them. It then deals on its own with the many memory-allocation calls from the application. This improves speed and, more important, avoids thrashing system memory (frequent allocation and de-allocation of small blocks).

If your application is compiled in the **Release** version without special settings, the Memory and API Resource Check profiler will still be able to monitor it, but calls to the runtime memory manager will not be profiled.

To track calls to the runtime memory manager in your VC++ application, you must compile it in the **Debug** version. You do this either simply by specifying the Debug compilation variant, or by putting the `#define Debug` directive in your source code and then setting compiler options as indicated below. A Debug version (commanded either way) is compiled to use MSVCRTD.DLL (MSVC runtime, debugging version), and this is what AQtune uses to monitor the runtime memory manager. See *Profiling Memory Management Routines*.

Using a Debug version is what we recommend. However, you may prepare a **Release** version so that the Memory and API Resource Check profiler can monitor calls to the runtime memory manager:

- Select **Project | Settings** from the main menu in VC++. This will open the **Project Settings** dialog.
- Move to the **C/C++** page.
- Select **General** from the **Category** list and then add `_Debug` to the preprocessor definitions.
- Select the **Code Generation** category. Choose *Debug Single-Threaded*, *Debug Multithreaded* or *Debug Multithreaded DLL* from the **Use run-time library** dropdown list.
- Press **OK** to close the Project Settings dialog.

Note that if you wish to profile an ActiveX control, it must be compiled in the Debug version and this "debug version" control must be registered in the system.

Typically, in VC++ applications many functions are imported from additional DLLs, e.g. MFC42.DLL. For example, when you call methods of the CPen object, the functions of the MFC42.DLL are used. To make this library available for the Memory and API Resource Check profiler, you should add it to the **Modules** page of the Settings dialog and check it there. Note that you should add the debug version of this library, i.e. MFC42D.DLL (the letter D suffix indicates that this is a debug version library).

Profiling VCL Applications

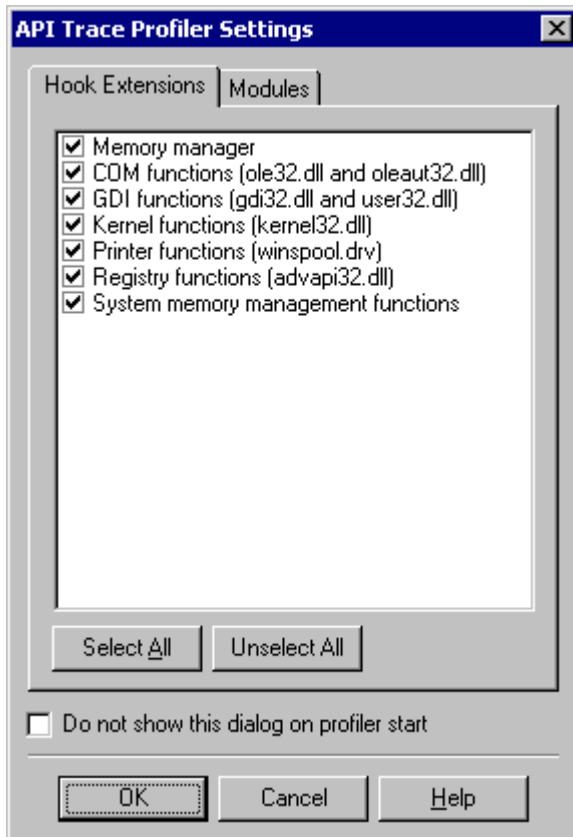
Two compiler options are important when VCL applications are to be profiled with the Memory and API Resource Check profiler:

1. Use debug libraries. This option must be enabled for C++Builder. It is located on the Linker page of the Project Options dialog.

2. Build with run-time packages. The Memory and API Resource Check profiler analyzes two kinds of memory management functions: System (Windows) and Memory Manager (runtime library sub-allocator). If run-time packages are used, the Memory Manager will be located in a VCL package. To profile calls to Mmory Mnager you have to select this package on the Modules page, or add it to the Setup panel. See *Profiling Memory Management Routines*.

Settings Dialog





This dialog lets you specify which function categories will be traced while profiling and which modules (EXEs, DLLs and OCXs) will be profiled. If *Customize settings before profiling* is enabled in Memory and API Resource Check Profiler Options, this dialog appears when you start profiling.



The dialog holds two pages: **Hook Extensions** and **Modules**.

In the Hook Extensions page, you can select groups of function categories to be tracked when profiling:

Memory manager


-  VCL Memory
-  MSVC++ Memory
-  VB Memory
-  BCB RTL Memory


COM functions (ole32.dll and oleaut32.dll)

 CoInitialize, OLEInitialize, IMalloc, etc.

For the full list of functions checked, see *List of Checked Functions - COM functions*.

GDI functions (gdi32.dll and user32.dll)

 Pen, Brush, Font, etc.

 Window, Dialog, Cursor, etc.

 Icon and File Operation

For the full list of functions checked, see *List of Checked Functions - GDI functions*.

Kernel functions (kernel32.dll)

 File, Handle, Process, Thread, etc.

For the full list of functions checked, see *List of Checked Functions - Kernel functions*.

Printer functions (printspool.drv)

 Printer

 Print Dialogs

For the full list of functions checked, see *List of Checked Functions - Print Spooler functions*.

Registry functions (advapi32.dll)

 Registry

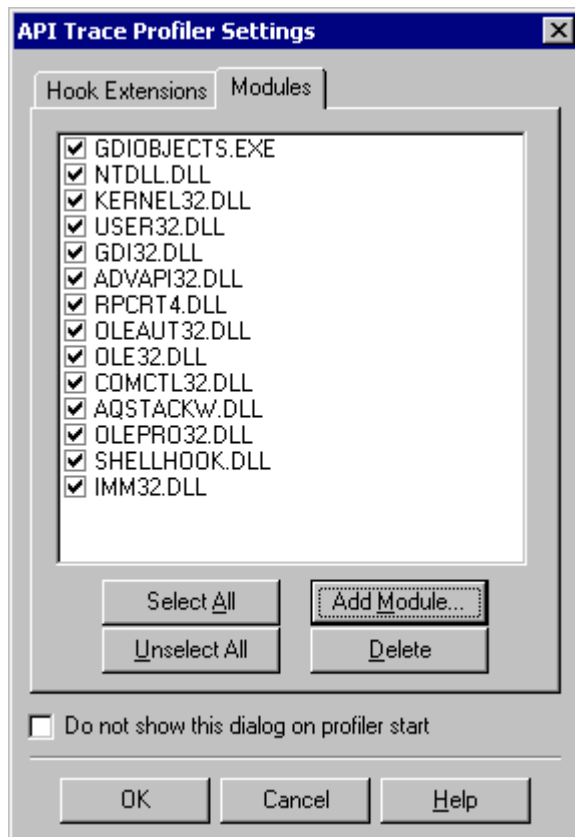
For the full list of functions checked, see *List of Checked Functions - Registry functions*.

System memory management functions

 Virtual Memory, Heap, Local Heap, etc.

For the full list of functions checked, see *List of Checked Functions - System memory management functions*.

In the **Modules** page you can select modules to be profiled. During the profiling, AQtime determines what modules (EXEs, DLLs or OCXs) are loaded and adds these modules to the page. You can add or remove some modules by pressing the **Add Module...** or **Delete** buttons.



To avoid excessive profiling, Memory and API Resource Check includes an Ignore system modules option. If it is on (default state), system dynamic link libraries are excluded from profiling and they are not displayed on this page.

If a Visual C++ application was built using the MFC library you may need to select MFC4D.DLL for profiling. See *Memory and API Resource Check - Profiling VC++*.

For a Borland application compiled with VCL packages, you may need to select VCLxx.BPL. See *Memory and API Resource Check - Profiling VCL Applications*. Otherwise, there is no need to add packages or borlandmm.dll to the current project – Memory and API Resource Check automatically includes them in profiling.

In the case of a Borland application compiled with VCL packages, you may need to select VCLxx.BPL. See *Memory and API Resource Check - Profiling VCL Application*. Otherwise, there is no need to add packages or borlandmm.dll to the current project – Memory and API Resource Check automatically includes them in profiling.

Important! The modules selected in this dialog (Settings) and the modules in the Setup panel are profiled in a different way. This mostly concerns memory allocation profiling. If you need to profile resource allocation, just check the necessary module on the Modules page:

- In the modules selected in the Setup panel, the profiler will track all functions that use the memory manager as well as the system memory management functions (VirtualAlloc, HeapAlloc, etc).
- In the modules selected on here, AQtime the profiler will only track memory allocations done through the system memory management functions.

A small example: Suppose, your project includes dll and exe files. If you check the dll file in the Modules page and do not add it to the Setup panel, AQtime will profile only memory allocation that is done using the external functions (e.g. *HeapAlloc*). It will not analyze memory allocation performed within the dll using the memory

manager (e.g. the *GetMem* function or the *new* operator). For more information about memory managers, see *Profiling Memory Management Routines*.

Note that, depending on compiler options, the memory manager can be located either in the profiled module or in a dll (VC++) or package (VCL) used by this module. In this case, you may have to select the module in the Modules page or add it to the Setup panel to profile calls to memory manager. See *Profiling Memory Management Routines*.

Profiling Memory Management Routines

The Memory and API Resource Check profiler tracks functions that allocate or de-allocate memory. There are two ways functions can do this. They can use **system** memory management calls, or they can call on the **runtime** memory manager that is part of the runtime library of VC++ and other tools. A runtime memory manager requests large blocks from the system, and eventually releases them. It then deals on its own with the many memory-allocation calls from the application. This improves speed and, more important, avoids thrashing system memory (frequent allocation and de-allocation of small blocks).

There are two ways in which you can set what memory allocations the Memory and API profiler will check. First, in **Options | Profiler Options | Memory and API** you must enable *Profile memory management routines* if you wish calls to the runtime manager to be profiled (runtime-library routines are the "routines" in question here).

Then, of course, the appropriate calling module must have been chosen for profiling in the Setup panel. In the case of system memory allocations, the calling module will normally be the runtime library module for the compiler used. If you enable *Customize settings before profiling* in the profiler options, the Memory and API Settings dialog will appear before each profile run, giving you a chance to modify the selections from the Setup panel.

For instance several common Borland VCL routines (*GetMem*, *FreeMem*, *ReallocMem*, etc) are calls on the VCL memory manager. So, to profile these calls, you should enable *Profile memory management routines*. The same applies to the VC++ equivalents, but with this compiler you must also compile your application in the Debug version. (See *Profiling VC++ Applications with Memory and API Resource Check*).

Note that memory manager can be located in the profiled module or in one of dynamic link libraries (VC++) or packages (VCL). This depends on compiler options. If you want to profile calls to memory manager, you may have to add the module to the Setup panel or select it in the **Modules** page of the Settings dialog. The following tables explain this:

VCL

| The "Build with run-time packages" option is... | Setup panel | Modules page | The memory manager is... |
|---|-------------|--------------|--------------------------|
| Enabled | X | - | - Not profiled |
| | X | X | - Profiled |
| | - | X | - Profiled |
| Disabled | X | - | - Profiled |
| | X | X | - Profiled |
| | - | X | - Not profiled |

Visual C++

| The module uses... | Setup panel | Modules page | The memory manger is... |
|--------------------|-------------|--------------|-------------------------|
|--------------------|-------------|--------------|-------------------------|

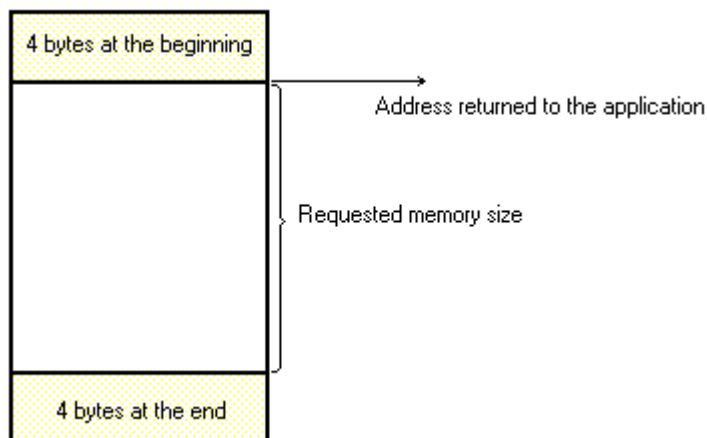
| | | | |
|-------------------------|---|---|-------------------|
| Dynamically linked DLLs | X | - | - Profiled |
| | X | X | - Profiled |
| | - | X | - Not profiled |
| Statically linked DLLs | X | - | - Profiled |
| | X | X | - Profiled |
| | - | X | - Not profiled |

One thing to note: One block can be allocated from system memory to the runtime manager, then re-allocated from the runtime manager to the application. The Memory and API profiler will recognize that the second allocation did not add to the total memory allocated for the application. But some other AQtime tools may record addition. e.g. the Real-Time Monitor.

Checking Bounds of Memory Blocks

The Memory and API Resource Check Profiler includes an option, *Check Memory Bounds*, that specifies whether the profiler reports an error when the profiled application writes to addresses above the upper or below the lower bound of an allocated memory block.

To implement this check, the hooks the functions that allocate memory blocks. It returns a block allocated for 8 bytes more than requested, but the application is informed only of owning the size it requested. 4 bytes are reserved before the requested block, and 4 are reserved after –



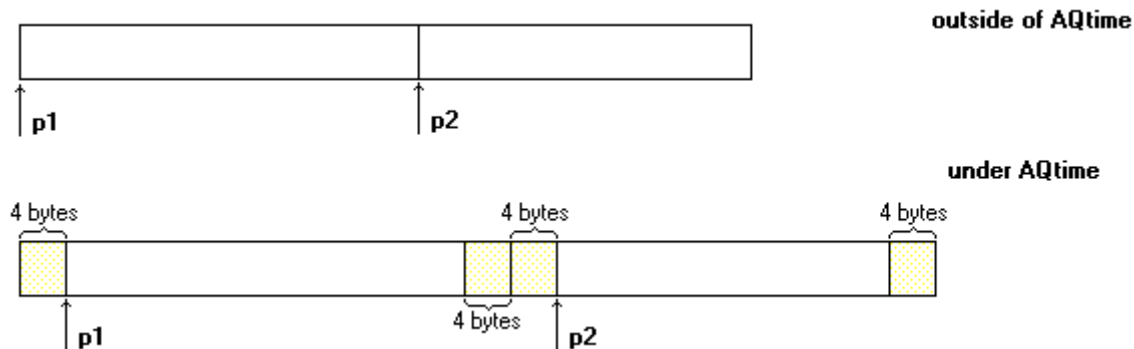
Important notes.

One. The profiler leaves its own signatures in each 4-byte buffer. It knows bounds were exceeded when it finds a signature to have been overwritten, and it checks for this when the application releases the block. In other words, if, on top of exceeding the bounds of the block, the application does not release it, the profiler will only report the unreleased memory. However, one common effect of writing out of bounds is to destabilize the application, leading to a crash – no memory is released at all. See Process Termination Restriction.

A good workaround is to check the error counter in the Monitor panel during profiling and to press **Project | Get Results** in case of an emergency. See Getting Results During Testing.

Two. The application may be coded on assumptions concerning memory, which should not be made, but which "generally work". It thus may usually work well outside AQtime and misbehave under AQtime with *Check Memory*

Bounds enabled. For instance, it might allocate two consecutive blocks of memory, then attempt to fill them in one call to `ZeroMemory()` --

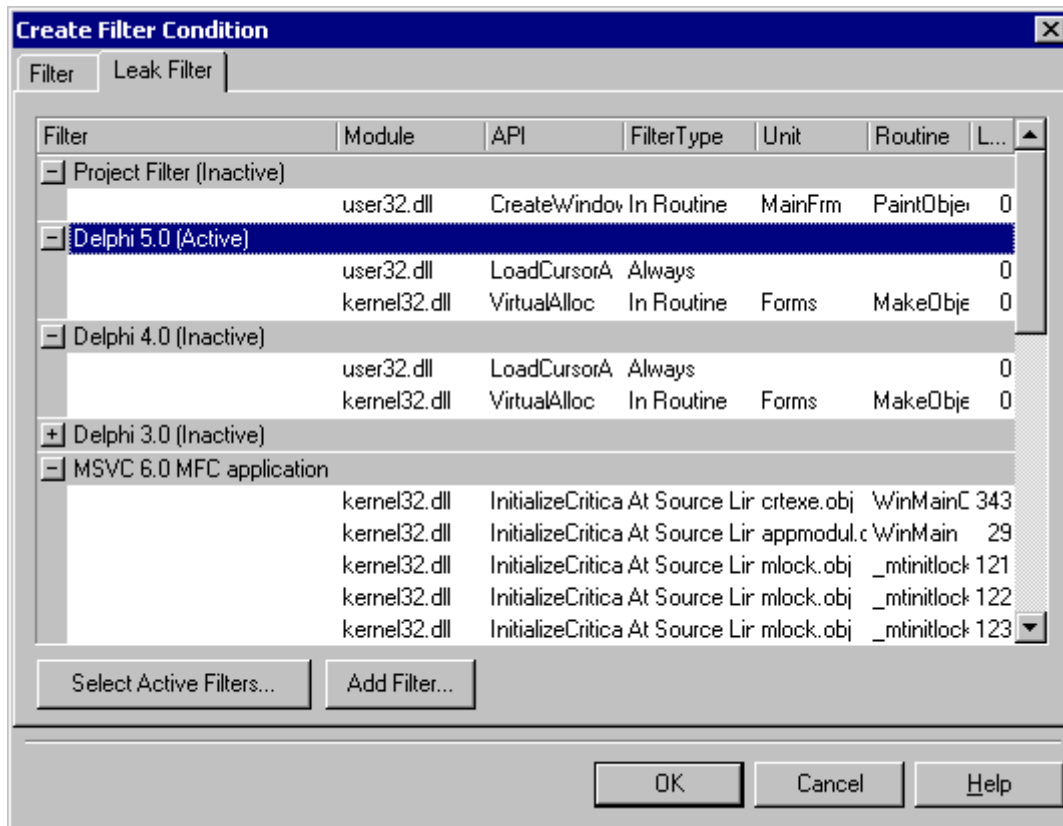


This ruins the memory-bounds checking, of course, but it also constitutes unexpected conditions for the application itself. The short answer is that applications should not be coded on the assumption that they control the order of allocations from the memory manager. The workaround is to disable *Check Memory Bounds*. If the problem goes away, then this was the probable cause.

Three. Since the bounds-checking control causes additional memory space to be allocated, some tools that trace the memory usage (e.g. Task Manager) will slightly exaggerate the memory used by your application.

Leak Filters Dialog

The Memory and API Resource Check plug-in includes special filters for better analysis of profiling results. These filters are available on the **Leak Filter** page of AQttime's Create Filter Condition dialog. This page is added to the dialog when you install the plug-in using the Install Extensions dialog.



The Leak Filters page holds a list of custom filters that allow you to exclude certain functions from the Report panel. These filters can be used along with ordinary filters. But they work in a different way: If a record matches an ordinary filter, it is kept in the Report panel. However, if a record matches one of the enabled Leak Filters, it is **excluded** from the Report panel. This page allows you to create filters that will hide some leaks, for instance leaks that occur in some external libraries and cannot be fixed by you. Detailed information about known VCL leaks is available at www.automatedqa.com/support/leaks.asp.

Each filter is a group of simple filter expressions (*items*) that set the actual filtering. Items are displayed as child nodes of filters, which are parent nodes. You can add, modify and remove filter items:

- To create a new filter item, choose **Add new...** from the context menu or press **Add Filter...**. This will open the Create Filter Item Dialog, where you can tune the filter item properties. Use the **Filter Name** combo box in this dialog to specify the filter the item will belong to. If you type a new filter name there, AQtme will create a new filter with this name.

New filter items are available for all projects unless they are added to a filter called **Project Filter**. This filter can be modified but not deleted. Items specified there are available for the current project only.

You can create new filters directly from the Report panel. Just select the desired leak and then choose **Filter Item** from the context (right-click) menu. This will call the Create Filter Item Dialog and fill its edit boxes with data from the Report panel.

- To edit an existing filter item, select it in the Leak Filters list and then choose **Modify...** from the context menu or simply double-click the filter item.
- To remove the existing filter item, select **Delete...** from the context menu.
- To rename the existing filter, right-click it, select **Rename...** from the context menu and specify the new filter name in the subsequent Rename Filter dialog.

The Leak Filters page holds a number of predefined filters. They help you separate leaks that occur in the profiled application from those that occur in VCL or MFC code. The predefined filters cannot be modified, renamed or deleted.

To specify active filters, press the **Select Active Filters...** button and then select filters to enable in the subsequent Project Filters dialog. The changes made in this dialog are saved for the current project only.

To save all modifications made in the Leak Filters page and apply filters to the Report panel, press **OK**. **Cancel** will close the dialog and discard any changes.

Predefined Filters

If your application includes code written using MFC, VCL or other libraries, there will be inevitable memory leaks due to errors in the imported library code. We call them *imported* leaks. The Memory and API Resource Check profiler of course tracks these leaks along with those that occur in your own code.

The imported leaks can distract your attention from the faults in your code. The Memory and API Resource Check profiler provides **predefined filters** to remove imported leaks from the results display. These filters also hide "pseudo-leaks" that occur due to the order in which DLLs are loaded and unloaded (see Leak Resources Restriction).

Currently there are such filters for **MSVC 6.0 MFC** and **Delphi (3.0 to 5.0)** applications. To view the structure of these filters, open the Leak Filters page of the **Create Filter Condition** dialog (See Memory and API Resource Check Profiler - Create Filter Condition Dialog). The predefined filters cannot be modified, renamed or deleted, but you can enable or disable them.

Predefined filters do not cover all possible cases. Remember that you can easily create your own filters – even by using leaks already displayed in the Report panel. Just select the desired leak and choose **Filter Item** from the context menu. This will call the Create Filter Item Dialog with fields filled-in for the selected leak. Press **OK** to create the new filter.

Detailed information about known VCL leaks is available at www.automatedqa.com/support/leaks.asp.

Leak Resources Restriction

When you profile a Delphi, C++Builder or VC++ application, you may note that sometimes AQtime reports that certain resources have not been released by the application, when they actually have been released.

It is possible of course that AQtime should simply be misbehaving. But in our experience, this is always instead due to the order DLLs in which were loaded and unloaded. The Memory and API Resource Check profiler uses a special DLL (APITraceHook) to record allocations and de-allocations, and this DLL must be loaded by the process being profiled. It is not impossible for a DLL to load after APITraceHook, allocate something, then only de-allocate and unload once APITraceHook itself has unloaded. The reverse can also occur, giving rise to a spurious report that the application attempted to de-allocate a resource it had not allocated.

For instance, in VC++ applications WinMain loads and unloads all DLLs, including APITraceHook. This gives rise to a minimum of three "unreleased resources".

You can solve the problem as it occurs in your own environment by creating a Leak Filter for the pseudo-leaks showing in the Report panel. Just select the desired leak and choose **Filter Item** from the context menu. This will call the Create Filter Item Dialog with fields filled-in for the selected leak. Press **OK** to create the new filter.

Leak filters can be used along with ordinary filters, but they work the opposite way. If a record matches an ordinary filter, it is kept in the Report panel. If a record matches one of the enabled Leak Filters, it is **excluded** from the Report panel.

Non-Existent Resources in the Report panel

The Memory and API Resource Check profiler reports whether the profiled application tried to release a non-existent resource. This can occur, for instance, if the application calls a memory-release function on an invalid pointer.

However, sometimes AQtime may report that there was an attempt to free a non-allocated resource while this resource actually is allocated. Typically, this happens because you do not profile all DLLs used by the application. For instance, an unprofiled DLL allocates a resource and passes it to the application, which is responsible for releasing it – the release call will appear to apply to a non-existent resource.

To solve the problem, either add all necessary DLLs to the Setup panel or add them to the **Modules** page of the Settings dialog. Note that there is no need to add packages for VCL applications – Memory and API Resource Check does not use them in analysis.

An alternative explanation is found in Leaked Resources Restriction, but it is less likely.

List of Checked Functions

All the checked functions are divided into six groups in accordance with the DLL a function is exported from:

- COM functions
- GDI functions
- Kernel functions
- Print Spooler functions
- Registry functions
- System memory management functions

COM functions (ole32.dll and oleaut32.dll)

| | |
|-------------------|-----------------------|
| CoInitialize | CoInitializeEx |
| CoTaskMemAlloc | CoTaskMemFree |
| CoTaskMemRealloc | CoUninitialize |
| OleInitialize | OleUninitialize |
| SysAllocString | SysAllocStringByteLen |
| SysAllocStringLen | SysFreeString |
| SysReAllocString | SysReAllocStringLen |
| VariantClear | |

GDI functions (gdi32.dll and user32.dll)

| | |
|--------------------|------------------|
| CallWindowProcA | CallWindowProcW |
| CloseEnhMetaFile | CloseMetaFile |
| CloseWindowStation | CopyCursor |
| CopyEnhMetaFileA | CopyEnhMetaFileW |
| CopyIcon | CopyImage |
| CopyMetaFileA | CopyMetaFileW |

| | |
|----------------------------|-----------------------------|
| CreateAcceleratorTableA | CreateAcceleratorTableW |
| CreateBitmap | CreateBitmapIndirect |
| CreateBrushIndirect | CreateColorSpaceA |
| CreateColorSpaceW | CreateCompatibleBitmap |
| CreateCompatibleDC | CreateCursor |
| CreateDCA | CreateDCW |
| CreateDialogIndirectParamA | CreateDialogIndirectParamW |
| CreateDialogParamA | CreateDialogParamW |
| CreateDIBitmap | CreateDIBPatternBrush |
| CreateDIBPatternBrushPt | CreateDIBSection |
| CreateDiscardableBitmap | CreateEllipticRgn |
| CreateEllipticRgnIndirect | CreateEnhMetaFileA |
| CreateEnhMetaFileW | CreateFontA |
| CreateFontIndirectA | CreateFontIndirectW |
| CreateFontW | CreateHalftonePalette |
| CreateHatchBrush | CreateICA |
| CreateIcon | CreateIconFromResource |
| CreateIconFromResourceEx | CreateIconIndirect |
| CreateICW | CreateMDIWindowA |
| CreateMDIWindowW | CreateMenu |
| CreateMetaFileA | CreateMetaFileW |
| CreatePalette | CreatePatternBrush |
| CreatePen | CreatePenIndirect |
| CreatePolygonRgn | CreatePolyPolygonRgnstdcall |
| CreatePopupMenu | CreateRectRgn |
| CreateRectRgnIndirect | CreateRoundRectRgn |
| CreateSolidBrush | CreateWindowA |
| CreateWindowExA | CreateWindowExW |
| CreateWindowStationA | CreateWindowStationW |
| CreateWindowW | DefFrameProcA |
| DefFrameProcW | DefMDIChildProcA |
| DefMDIChildProcW | DefWindowProcA |
| DefWindowProcW | DeleteColorSpace |
| DeleteDC | DeleteEnhMetaFile |
| DeleteMetaFile | DeleteObject |
| DestroyAcceleratorTable | DestroyCursor |
| DestroyIcon | DestroyMenu |

| | |
|------------------------|------------------------|
| DestroyWindow | DuplicateIcon |
| ExtCreatePen | ExtCreateRegion |
| ExtractAssociatedIconA | ExtractAssociatedIconW |
| ExtractIconA | ExtractIconW |
| GetClassInfoA | GetClassInfoExA |
| GetClassInfoExW | GetClassInfoW |
| GetDC | GetDCEx |
| GetEnhMetaFileA | GetEnhMetaFileW |
| GetIconInfo | GetMetaFileA |
| GetMetaFileW | GetStockObject |
| GetWindowDC | InsertMenuA |
| InsertMenuItemA | InsertMenuItemW |
| InsertMenuW | KillTimer |
| LoadBitmapA | LoadBitmapW |
| LoadCursorA | LoadCursorFromFileA |
| LoadCursorFromFileW | LoadCursorW |
| LoadIconA | LoadIconW |
| LoadImageA | LoadImageW |
| LoadKeyboardLayoutA | LoadKeyboardLayoutW |
| LoadMenuA | LoadMenuIndirectA |
| LoadMenuIndirectW | LoadMenuW |
| OpenWindowStationA | OpenWindowStationW |
| RegisterClassA | RegisterClassExA |
| RegisterClassExW | RegisterClassW |
| ReleaseDC | ReleaseStgMedium |
| SetClipboardData | SetEnhMetaFileBits |
| SetMetaFileBitsEx | SetTimer |
| SetWindowRgn | SetWinMetaFileBits |
| SHFileOperationA | SHFileOperationW |
| SHFreeNameMappings | UnloadKeyboardLayout |

Kernel functions (kernel32.dll)

| | |
|----------------------|---------------------------|
| _lclose | _lcreat |
| _lopen | BeginUpdateResourceA |
| BeginUpdateResourceW | CloseEventLog |
| CloseHandle | CreateConsoleScreenBuffer |
| CreateEventA | CreateEventW |

| | |
|------------------------------|------------------------------|
| CreateFiber | CreateFileA |
| CreateFileMappingA | CreateFileMappingW |
| CreateFileW | CreateMailslotA |
| CreateMailslotW | CreateMutexA |
| CreateMutexW | CreateNamedPipeA |
| CreateNamedPipeW | CreatePipe |
| CreateProcessA | CreateProcessW |
| CreateRemoteThread | CreateSemaphoreA |
| CreateSemaphoreW | CreateThread |
| DeleteCriticalSection | DeleteFiber |
| DeregisterEventSource | DuplicateHandle |
| EndUpdateResourceA | EndUpdateResourceW |
| EnterCriticalSection | FindClose |
| FindCloseChangeNotification | FindFirstChangeNotificationA |
| FindFirstChangeNotificationW | FindFirstFileA |
| FindFirstFileW | GetStdHandle |
| InitializeCriticalSection | LeaveCriticalSection |
| MapViewOfFile | MapViewOfFileEx |
| OpenBackupEventLogA | OpenBackupEventLogW |
| OpenEventA | OpenEventLogA |
| OpenEventLogW | OpenEventW |
| OpenFile | OpenFileMappingA |
| OpenFileMappingW | OpenMutexA |
| OpenMutexW | OpenProcess |
| OpenSemaphoreA | OpenSemaphoreW |
| RegisterEventSourceA | RegisterEventSourceW |
| ReleaseMutex | ReleaseSemaphore |
| TerminateThread | TlsAlloc |
| TlsFree | TryEnterCriticalSection |
| UnmapViewOfFile | |

Print Spooler functions (printspool.drv)

| | | |
|--------------|--------------|--------------|
| ClosePrinter | OpenPrinterA | OpenPrinterW |
|--------------|--------------|--------------|

Registry functions (advapi32.dll)

| | |
|---------------------|---------------------|
| RegCloseKey | RegConnectRegistryA |
| RegConnectRegistryW | RegCreateKeyA |

| | |
|-----------------|-----------------|
| RegCreateKeyExA | RegCreateKeyExW |
| RegCreateKeyW | RegOpenKeyA |
| RegOpenKeyExA | RegOpenKeyExW |
| RegOpenKeyW | |

System memory management functions

| | |
|------------------|------------------|
| GlobalAlloc | GlobalFree |
| GlobalReAlloc | HeapAlloc |
| HeapCreate | HeapDestroy |
| HeapFree | HeapReAlloc |
| LocalAlloc | LocalFree |
| LocalReAlloc | PrintDlgA |
| PrintDlgW | ReleaseStgMedium |
| SetClipboardData | VirtualAlloc |
| VirtualFree | |

Profilers How-To

Enabling and Disabling Profiling From Application Code

AQtime is an OLE server which can be controlled from other applications. The entire interface is provided in AQtime.idl, which is part of the standard installation. AutomatedQA's AQtest makes extensive use of it to integrate AQtime tests and results. But the simplest use of AQtime as an OLE engine is simply to turn profiling on or off from application code. This has the same effect as using the Enable Profiling toolbar button. Note however that the OLE commands only work if there is only one instance of AQtime running.

This is useful wherever Areas plus Triggers do not give you the control you seek, or where they would, but at the cost of some complications, or simply where you want to set Triggers from source code rather than from the AQtime user interface.

For C++ or Delphi applications, all the functions you need to do this are provided in a few files that you add to your source files. Once you have them linked in, turning profiling on and off is a matter of a few simple calls.

For Borland Delphi or C++Builder, the files to add are: AQtimeCOMAPIProvider.pas and AQtime_TLB.pas from the <AQtime>\PlugIns folder.

For Microsoft Visual C++ applications, the files to add are: AQtimeCOMAPIProvider.h and AQtimeCOMAPIProvider.cpp, from the <AQtime>\PlugIns\MSVC folder, and AQtime.tlb from the <AQtime>\PlugIns folder.

The TLB files hold all the interfaces to the OLE server. AQtime_TLB.pas translates the IDL into Object Pascal declarations, AQtime.tlb translates it into a type library format which is not human-readable. AQtimeCOMAPIProvider contains the few higher-level declarations that you will use to turn profiling on or off from application code:

| Declaration | Description |
|---------------|---|
| TAQtimeDriver | Class implementing methods to connect to AQtime and operate |


| | |
|-----------------------------------|--|
| | with it. |
| AQtimeDriver | Variable of the TAQtimeDriver type. Once created, provides access to the AQtime interface. |
| InitializeCommunicationWithAQtime | Procedure. Establishes a connection between the AQtime OLE server and your application. Uses one integer parameter, which it passes to CoInitializeEx. |
| FinalizeCommunicationWithAQtime | Procedure. Closes the connection between AQtime and your application. |
| AQtimeEnableProfiling | Function. Enables or disables profiling according to its boolean parameter. |

Call InitializeCommunicationWithAQtime procedure to establish the OLE connection. Once this is done, you could use the AQtimeDriver variable to control AQtime. But, if all you need is to enable or disable profiling, calling AQtimeEnableProfiling is simpler. Once your application has no more need for controlling AQtime, call FinalizeCommunicationWithAQtime to close the connection.

The important thing is to make sure that the connection is established before you call AQtimeEnableProfiling, and closed only when this will not be called anymore. It is not impossible, though, to open the connection, use it, close it, then re-open it later and use it again. Using the connection in any way when it is not established, or is closed leads to access violation.

You can try this first with the sample application, OnOffProfiling, that is part of your AQtime installation (in source). Or you can try the following sample code in an application of your own.

In either case, before running the test, set up the application in AQtime so that the application has full control over profiling:

- Select *FULL CHECK* in the AQtime's Setup. For more convenient result analysis, you should uncheck all profiling areas, except *FULL CHECK*.
- Be sure the  **Enable/Disable Profiling** button is not pressed so AQtime will not profile the application by itself.

Sample code:

```
...
// Initializes a connection with AQtime
InitializeCommunicationWithAQtime(0);
// Enables profiling
AQtimeEnableProfiling(True);

// Call the Large_Function
Large_Procedure(Param1, Param2);


// Disables profiling
AQtimeEnableProfiling(False);
// Closes a connection with AQtime
```

```
FinalizeCommunicationWithAQtime();
...
```

There is an entire tutorial devoted to this topic: [Enable/Disable Profiling From External Application](#). Be sure to check it.

Getting Results During Testing

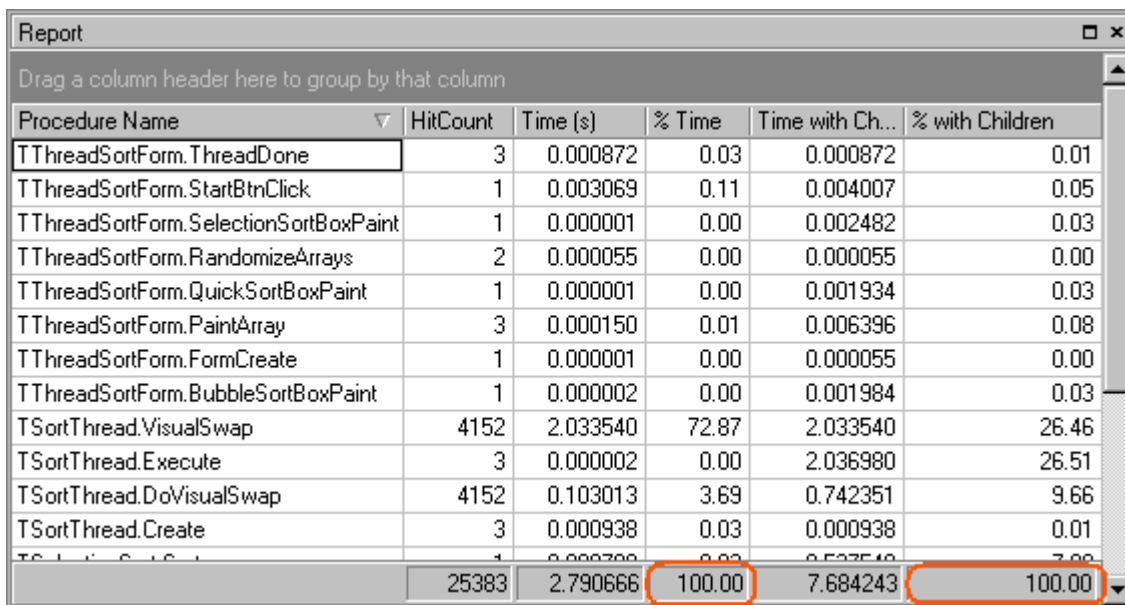
AQtime normally generates results after the profiled application has ended its run. When profiling a dynamic link library, this means results are generated when the host application exits.

However, you might be profiling an application that never stops until system shut-down (e.g. an NT service or and IIS application), or you may simply wish to obtain results without closing the running application (e.g. the host for a dll). You can do this by using  **Get Results** in the Project submenu from the main menu. Results will be generated as if the profiled application had terminated.

Calculating Percent Time With Children

Function Profiler options include a *% with children relative to real time* check box which is in the off state by default. This option affects how **% with children** is calculated in the Report panel.

In the default state, *% with children relative to real time* disabled, **% Time** is the **Time** value as a percentage of the total of all **Time** values (as shown in the footer of the Time column). Likewise **% with children** is the **Time with children** value as a percentage of the total of all values in that column. The total in the footer of **% with children** is 100%. For instance --



| Procedure Name | HitCount | Time (s) | % Time | Time with Ch... | % with Children |
|---------------------------------------|--------------|-----------------|---------------|-----------------|-----------------|
| TThreadSortForm.ThreadDone | 3 | 0.000872 | 0.03 | 0.000872 | 0.01 |
| TThreadSortForm.StartBtnClick | 1 | 0.003069 | 0.11 | 0.004007 | 0.05 |
| TThreadSortForm.SelectionSortBoxPaint | 1 | 0.000001 | 0.00 | 0.002482 | 0.03 |
| TThreadSortForm.RandomizeArrays | 2 | 0.000055 | 0.00 | 0.000055 | 0.00 |
| TThreadSortForm.QuickSortBoxPaint | 1 | 0.000001 | 0.00 | 0.001934 | 0.03 |
| TThreadSortForm.PaintArray | 3 | 0.000150 | 0.01 | 0.006396 | 0.08 |
| TThreadSortForm.FormCreate | 1 | 0.000001 | 0.00 | 0.000055 | 0.00 |
| TThreadSortForm.BubbleSortBoxPaint | 1 | 0.000002 | 0.00 | 0.001984 | 0.03 |
| TSortThread.VisualSwap | 4152 | 2.033540 | 72.87 | 2.033540 | 26.46 |
| TSortThread.Execute | 3 | 0.000002 | 0.00 | 2.036980 | 26.51 |
| TSortThread.DoVisualSwap | 4152 | 0.103013 | 3.69 | 0.742351 | 9.66 |
| TSortThread.Create | 3 | 0.000938 | 0.03 | 0.000938 | 0.01 |
| Total | 25383 | 2.790666 | 100.00 | 7.684243 | 100.00 |

When *% Relative To Real Time* is enabled, **% with children** is the **Time with children** value as a percentage of the total of all values in the **Time** (not with children) column. Normally, this will yield a total in the footer of **% with children** much greater than 100%, as child time is being added in more than once. The advantage of the setting is that **% with children** tells you what any profiled function **costs**, child calls included, as a proportion of the total profiled time. For instance --

| Report | | | | | |
|---|----------|----------|--------|-----------------|-----------------|
| Drag a column header here to group by that column | | | | | |
| Procedure Name | HitCount | Time (s) | % Time | Time with Ch... | % with Children |
| TThreadSortForm.ThreadDone | 3 | 0.000718 | 0.03 | 0.000718 | 0.03 |
| TThreadSortForm.StartBtnClick | 1 | 0.004701 | 0.17 | 0.005592 | 0.20 |
| TThreadSortForm.SelectionSortBoxPaint | 1 | 0.000001 | 0.00 | 0.001939 | 0.07 |
| TThreadSortForm.RandomizeArrays | 2 | 0.000048 | 0.00 | 0.000048 | 0.00 |
| TThreadSortForm.QuickSortBoxPaint | 1 | 0.000001 | 0.00 | 0.001907 | 0.07 |
| TThreadSortForm.PaintArray | 3 | 0.000158 | 0.01 | 0.005820 | 0.21 |
| TThreadSortForm.FormCreate | 1 | 0.000001 | 0.00 | 0.000049 | 0.00 |
| TThreadSortForm.BubbleSortBoxPaint | 1 | 0.000001 | 0.00 | 0.001977 | 0.07 |
| TSortThread.VisualSwap | 4196 | 1.969378 | 72.05 | 1.969378 | 72.05 |
| TSortThread.Execute | 3 | 0.000002 | 0.00 | 1.972819 | 72.18 |
| TSortThread.DoVisualSwap | 4196 | 0.099612 | 3.64 | 0.748231 | 27.38 |
| TSortThread.Create | 3 | 0.000891 | 0.03 | 0.000891 | 0.03 |
| TSortThread.Destroy | 1 | 0.000001 | 0.00 | 0.000001 | 0.00 |
| | 25640 | 2.733232 | 100.00 | 7.499379 | 274.38 |

Note that *% with children relative to real time* is shorthand for "% relative to total time spent profiling". The time during which profiling is turned off, for instance for System Files, is simply not counted. Using "real time", that is, total elapsed time, would include all the time spent waiting for the user to do something.

When results are stored, they are stored with the current column contents. **% with children** will not change when you retrieve the results, no matter what the current setting for *% with children relative to real time*. You can easily see under what setting the results were generated by checking the footer for **% with children**. If it is 100%, then the option was off. If it is greater, the option was on.

Optimizing the Profiling Process

Below you'll find some tips for getting the most out of a profile cycle in AQtime, with the least wasted effort:

- When you profile a project for the first time, use the Sampling or HitCount profilers as a preliminary tool to narrow down the problem sections of the application. Next, apply the Function Profiler to optimize these areas or individual functions.
- Even a small, quick function can impair performance if it is called very frequently. Always check the Hit Count for anomalies. Are there errant hit counts, for instance where a very common piece of code makes a needless call? If they are not errant, can the very high hit counts be decreased by tuning your algorithms?
- Do not assume you know a function is "no problem". The profilers are there to give you a health report; use them. Known problems may have unexpected roots.
- Restrict you profiling Areas when you can. The profilers need time to gather information about the sections they are set to analyze (Areas). And they may take more time while profiling the actual execution. The more precise you Area specification, the faster the profiling. Remember that an application includes much code that will never need profiling. For instance, user interface code normally does little but wait on the user. See *Excluding Code from Profiling* and *Selecting Code To Profile*.
- Long functions can be difficult to profile. One way around this is to break them down into several subfunctions for profiling purposes. (You will probably find that the code is also clearer once broken down, and easier to analyze.)

- No real-world execution of an application is identical to another. From one run to the next, you should expect some inconsistency in results. For instance, AQtime tracks time by the high-resolution CPU clock. This includes time spent outside the application (including Windows background processes), which changes with each test. The prudent thing to do is to average several runs. See *Merging Results*. Where you need high-precision results, keep a very detailed record of how the test was run.
- To help achieve consistent test results, reduce the number of processes running on your machine during profiling.
- The VCL Class, VCL Reference Count, ATL Reference Count and Memory and API Resource Check profilers track where memory leaks are created. This requires much processing by AQtime, and will slow down your application. If you don't need the leaks traced in a given run, turn off the *Show call stack* option of the profilers. See Profilers Options - VCL Profilers, Profiler Options – ATL RefCount and Profiler Options - Memory and API Resource Check Profiler. In the case of the latter, there is also a *Stack depth shown* option for further control (the less the depth, the less time spent).
- Also, when these three profilers track leaks, by default they track all functions calls, even those for which there is no source. It is useful to know that there are leaks in code you do not have the source for, but most of the time it is your own code that you wish to work on. In this case, turn off the *Show all parents*. Turning this off will let AQtime collect less information, and profiling speed will increase. See *VCL Profiler Options* and *Profiler Memory and API Resource Check Profiler Options*.
- The Memory and API Resource Check profiler divides API functions onto several groups: COM, Kernel, GDI, etc. This allows you to improve profiling speed (and simplify results) by unchecking unnecessary function groups in the Settings dialog. This dialog appears by default when the profiler is launched. You can also use it to uncheck unnecessary modules. Finally, if you are interested only in leaked resources and memory blocks, you can turn off the *Show API parameters* option, so that the profiler will not track parameters and results on each call.

Profiling Recursive Functions

We will call *recursive* any function that calls itself, or that is eventually called by some child function it called. In AQtime terms, a recursive function is one that belongs to its own descendance (children, grandchildren, etc.). For the Function Profiler, this poses an unavoidable problem concerning what AQtime should call Time With Children and what it should call Time (i.e., without children) for such a function.

This topic explains the Time With Children problem for recursive functions, and the solution adopted in the Function Profiler.

Remember first that Time and Time With Children apply not to one call, but to the sum of calls throughout the profile run. Now, imagine that one function, FuncA, calls itself three times in a row, so that the original call, FuncA1, gives rise to three more, FuncA2, FuncA3 and FuncA4. Imagine also that FuncA takes 2 seconds to execute its own code. If these are the only calls during the profile run, Time should be 8 seconds. But Time With Children?

FuncA4 = 2

FuncA3 = 4

FuncA2 = 6

FuncA1 = 8

Total = 20 seconds.

Now, imagine also that the entire run was simply the original FuncA1 call. The entire run lasted 8 seconds, but Time With Children for FuncA is 20 seconds. This is grossly misleading. The reason is that one single execution, FuncA4, is counted separately as part of the child time for FuncA3, FuncA2 and FuncA1, and it is also counted once

as its "own" time – it's counted four times in all. Likewise, FuncA3 is counted three times and FuncA2 is counted twice. These repeat counts for the same actual execution bloat up Time With Children as soon as there is recursion.

In the very simple example above, we also know what solution we'd like – Time With Children should be identical with run time, 8 seconds. That is, the same as Time itself, since both values count exactly the same calls (FuncA1 through FuncA4).

The way this is done in the Function Profiler is simply that **any call that is (even remotely) part of a call to the same function is not profiled**. On any call, AQtime first checks whether it is already counting execution time for the same function in the current thread and, if this is the case, simply does not add anything to Time or Time With Children for the coming call. Child calls to other functions are profiled, though.

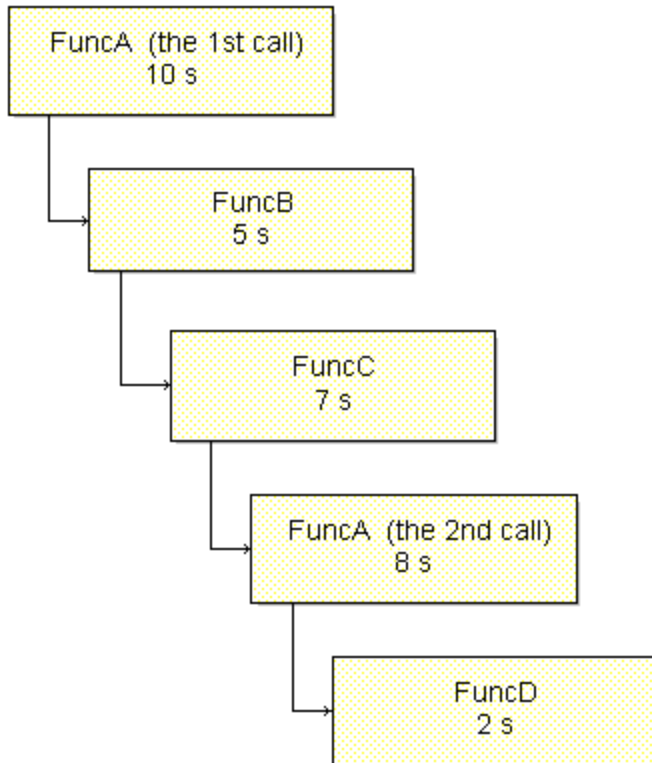
With our simple FuncA example, this means FuncA2, FuncA3 and FuncA4 contribute nothing to Time With Children, so it remains what it was for FuncA1 alone, 8 seconds, the same as the run time. This is what we wanted.

However, Time only counts FuncA1, 2 sec., dismissing the three recursive calls. This isn't what we would want. The problem is that the actual time information gathered by the profiler is Time With Children. Time is simply the difference between this and the Time With Children of each immediate child call. In other words, if Time With Children is counted as 0, then Time will be 0 also. The alternative would add code to the profiler that would be executed on each function call, recursive or not, thus slowing the entire profile run, while in most cases there is no recursion to account for anyway.

Not profiling the recursive calls is inexpensive, but it implies the compromise we've just seen. Time only counts the first, non-recursive call. To warn you about this, there is an *Track recursion depth* option for the Function Profiler. If this is enabled, the Report and Call Graph panels have an extra column showing the maximum recursion depth found during the run for each function.

As this is useful information, the option is also available for the Function HitCount profiler, which has no problem with recursion (all hits are counted). See *Function Profiler Options* and *Function HitCount Options*.

Just to make sure everything is clear, here is a somewhat more complex example:



The timings are time spent executing the function itself.

If the Function Profiler timed both the first and the second call to FuncA, we would get the following results:

| Name | Time | Time with Children |
|-------|------|--------------------|
| FuncA | 18 | 42 |
| FuncB | 5 | 22 |
| FuncC | 7 | 17 |
| FuncD | 2 | 2 |

Time With Children for the second call is counted twice (once as part of the first call, and once as a second call), so we get a total of 42 seconds for a run time of 32 seconds. What the Function Profiler will actually give you is this, rather:

| Name | Time | Time with Children |
|-------|------|--------------------|
| FuncA | 10 | 32 |
| FuncB | 5 | 22 |
| FuncC | 15 | 17 |
| FuncD | 2 | 2 |

Everything becomes quite obvious in the Details panel. FuncA is in the child list of FuncC, of course. But the **Time** and **Time with Children** columns for FuncA show 0, directly telling you this call was not profiled.

Note that if the first call to FuncA is not profiled for any reason (e.g. it is excluded by an off-trigger), the Function Profiler detects no recursion.

Overloaded Functions

AQtime profiles overloaded functions in the same manner as non-overloaded ones. As overloaded functions share the same name, the Report panel displays them using the following convention:

```
function_name overloaded n ;
```

The important element here is n , the sequence number of the overloaded function as it appears in source (earliest-linked file first, earliest line first). This system breaks down if the order of overloaded functions changes in source. Function n at the time the Areas or Triggers were defined, isn't function n in the current source or exe.

The solution is to remove the functions from Areas or Triggers, then put them back in after the source order has changed, for instance because you have inserted more overloaded versions.

Profiling Inline Functions

AQtime's *function* profilers track entry and exit points of a function. When a C++ function is set as inline, the compiler may insert a copy of the function body in each spot it is called (or it may disregard the directive). Obviously, with a true inlined function, there are no entry and exit points to track.

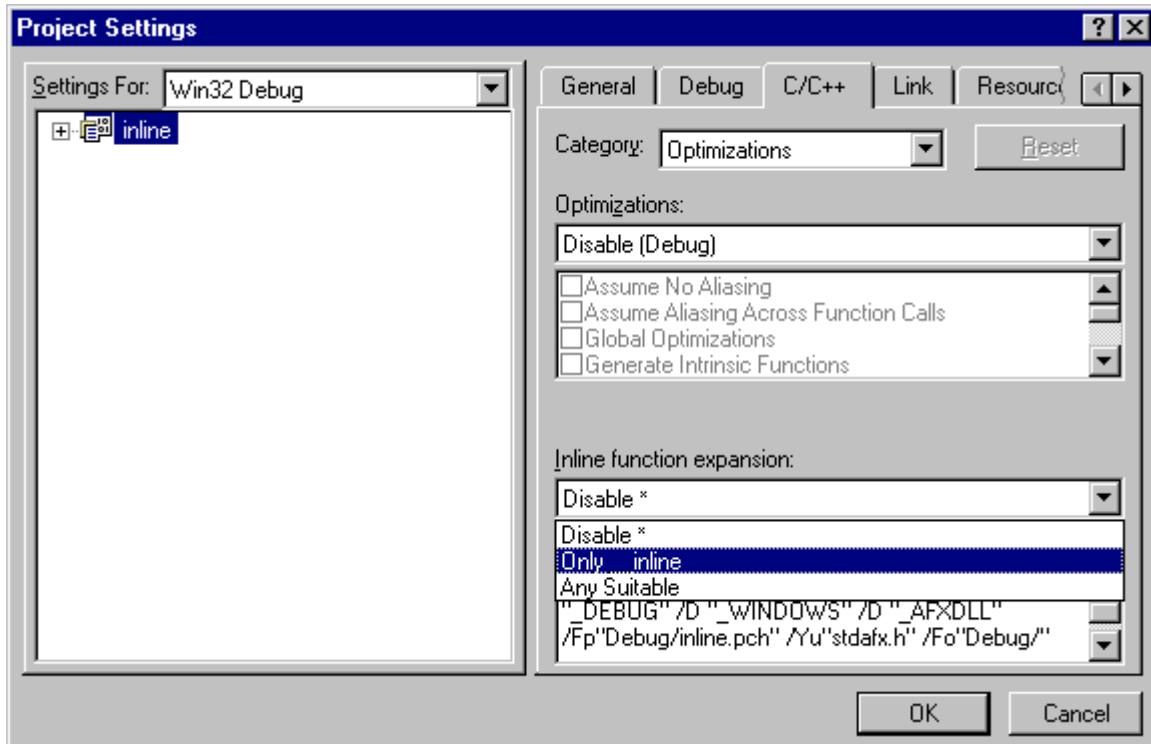
So, if you want an inline function profiled, you must set your compiler not to inline it.

Microsoft Visual C++ distinguishes the ways in which an inline function is specified as inline:

1. Using the *inline* keyword.
2. For a member function, having its body declared in the class definition.

Using `#pragma auto_inline` to tell the compiler to inline functions according to criteria or its own.

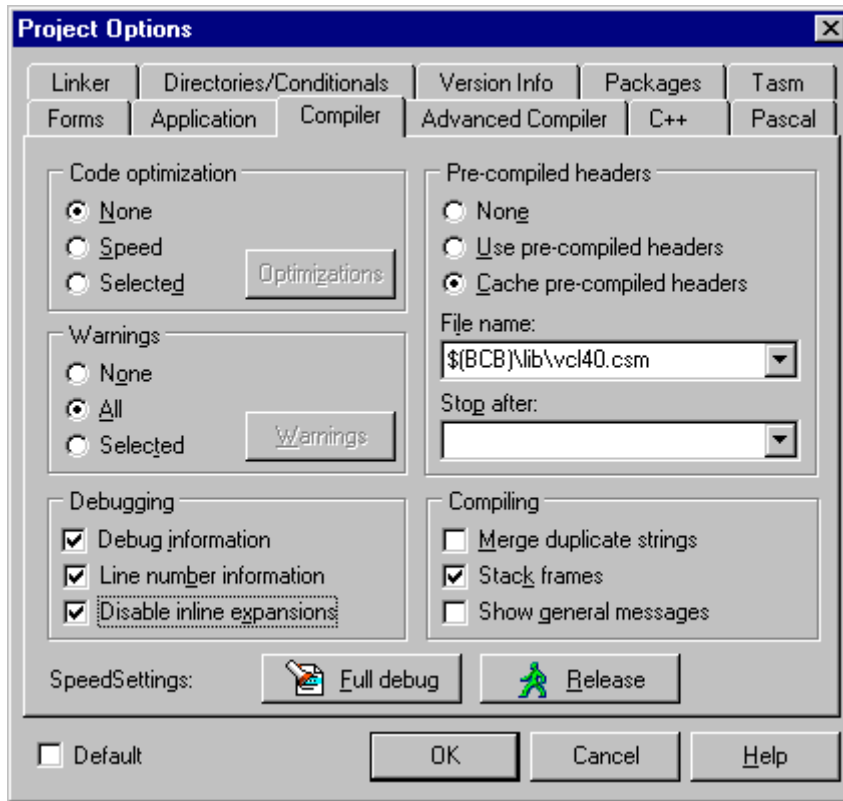
In the corresponding dialog:



you have three options:

- | | |
|--------------|--|
| Disable * | Simply inline nothing. This is the simplest for using AQtime. |
| Only Inline | Only member functions implemented in the class definition will be inlined. All other functions will be accessible to AQtime profilers. |
| Any Suitable | The compiler produces inline code for all functions marked as inline as well as for any suitable functions defined under <code>#pragma auto_inline</code> . This is the setting most likely to cause problems. |

In **Borland C++Builder** things are simpler:



Disable inline expansions means the compiler produces standard code for all inline functions, so AQtime can profile inline functions of any kind.

Profiling With Microsoft PDB or DBG Debug Info

Microsoft Visual C++ and Microsoft Visual Basic can generate debug info in several forms. The debug information can be included in the executable, or it can be put in a separate file, in the latter case using either PDB or DBG format.

PDB and DBG files are meant to be accessed through a special Microsoft DLL, DbgHelp.dll. It is part of your AQtime installation, and you will need it to profile VC++ and VB applications that use PDB or DBG files for their debug info.

Currently AQtime uses DbgHelp.dll v. 5.0.2195.1. There are several Microsoft sources for the dll (perhaps in a different version):

- Platform SDK (MSDN CD and web site)
- Windows 2000 DDK (MSDN CD and web site)
- Windows 2000 Resource Kits (CD and web site)
- Windows 2000 operating system CDs (\support subfolder)

At the time of writing, you can download the DLL free from the public Microsoft website --

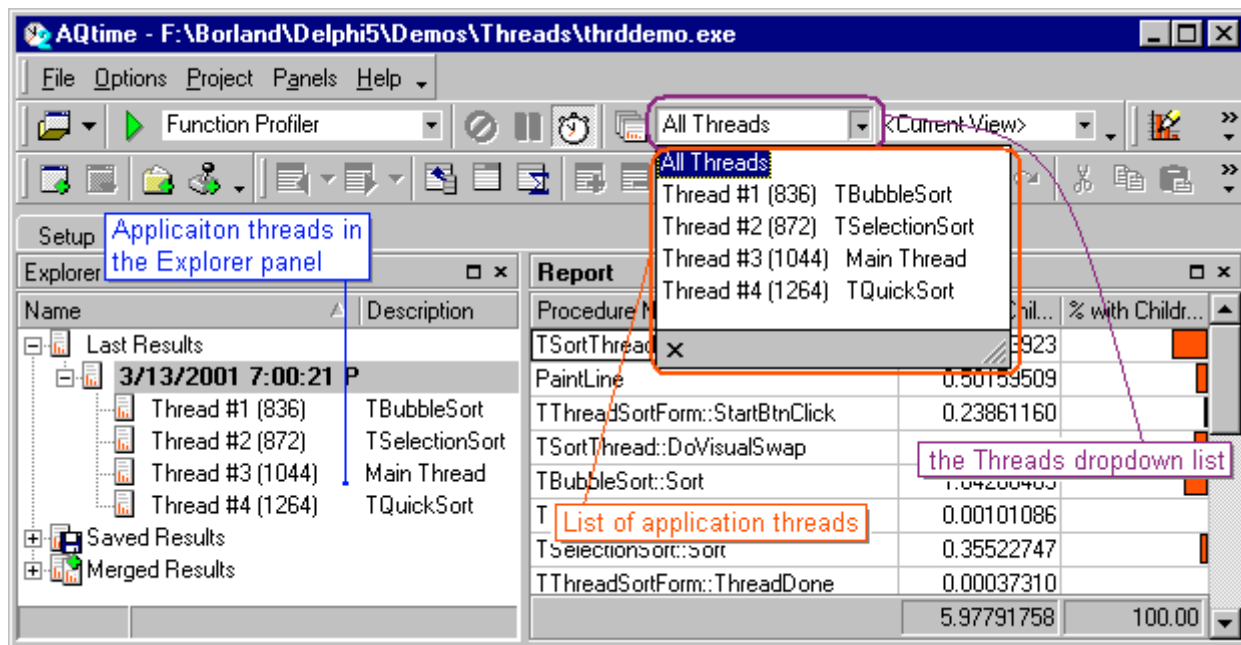
1. Open <http://msdn.microsoft.com/downloads/sdks/platform/windbg.asp>, and request download of the **Windows 2000 Debuggers Component**. This free software contains the necessary DLLs.
2. Select **Microsoft 2000 Debuggers (x86) for Windows 9x & NT** in the combo box and then download this software.

- When downloading is over, install the software and copy DbgHelp.dll into AQtime's folder. After that you can remove **Windows 2000 Debuggers Component** from your hard-drive.

Profiling Multithreaded Applications

Multithreaded application profiling is supported by the Function Profiler, Line HitCount, Function HitCount, Line Coverage, Line Coverage (Grouped by File) and Function Coverage profilers.

This is quite transparent. Each of these profilers logs and saves results by thread. If the profile run included more than one thread, you can select a single thread to show on the Report pane from the **Threads** dropdown list on the Standard toolbar (next to the **View** dropdown) or you can select a thread in the Explorer panel. Of course, you can display all threads together:



Function Coverage, Function HitCount and Function Profiler also can be controlled by Triggers, which turn profiling on or off for the particular thread a function (the Trigger) runs in. This is an essential tool to winnow out profile information from complex multithreaded applications.

Note that a thread is not a process. If a profiled application launches a new process, this will simply escape profiling by AQtime, which can only watch over the child process it has launched itself, that is, the main application process. For instance, to profile an ActiveX control (OLE server) when it is not an in-process server, you must load it in AQtime as the main process, as explained in Profiling ActiveX Controls and OLE Servers.

An important convenience is that you can assign readable names to threads. The thread names will be displayed in the **Description** column of the Explorer panel and in the Threads dropdown list on the Standard toolbar.

This is done by calling calling `AQTimeSetThreadName` from your application source code. It takes two parameters: The first (integer) specifies the thread id, the second (string) specifies the name for the thread.

For C++ and Delphi, follow these steps:

- Open your application's project in your development tool.
- Add the AQtimeCOMAPIProvider file to your project. This file is located in the following folder:

For C++ - <AQtime>\Plugins\MSVC

For Delphi - <AQtime>\Plugins

It holds AQTimeSetThreadName and other routines used to communicate with AQtime.

- In your code, first set up the connection to AQtest by calling InitializeCommunicationWithAQTime with the appropriate COINIT flag, e.g. COINIT_MULTITHREADED. (This routine is declared in AQtimeCOMAPIProvider).
- Then call AQTimeSetThreadName(*ThreadID*, *Name*).
- To get the id of the current thread, you can use the GetCurrentThreadID routine (Windows API). Note for Delphi and C++Builder users: If your thread is based on the TThread object, use its ThreadID property (but not Handle) to obtain the id.
- Close the connection by calling FinalizeCommunicationWithAQTime.

For Visual Basic the steps to follow are:

- Open your application in Visual Basic.
- Add the following lines to initialize a connection to AQtime:



```
Dim AQtime As Variant
. . .
Set AQtime = GetObject(, "AQTime.AQTimeManager")
```
- To assign a name to a thread, you will first need its ID. The following code calls GetCurrentThreadID (Win API) to get the current thread and assign it a name:


```
If Not AQtime Is Nothing Then
    AQtime.Manager.Project.SetThreadName GetCurrentThreadID, "MyThread"
End If
```
- Call `Set AQtime = Nothing` to close the connection when you are done.

Profiling Dynamic Link Libraries

You can profile both statically and dynamically linked DLL using AQtime. Profiling a dynamic link library is similar to profiling any standard application. You should perform the following steps:


- Compile your DLL with debug information. Preparing a Project for Profiling explains how to do this.
- Load the DLL in AQtime as a project.
- Open the Run Parameters dialog and specify the host application for a dynamic link library. When you start profiling, AQtime launches the specified host application. The host application, in turn, calls functions defined in the profiled dynamic link library. Note that AQtime does not profile the host application, so it can be compiled without the debug info.
- Continue profiling in the usual manner. AQtime profiles DLL functions only when they are included in one of the profiling areas or with active *FULL CHECK*.

In case the host application is already loaded into AQtime as a project, you may add the DLL as a module: Choose  **Add Module...** from the Setup toolbar or from the context menu, and then select the desired DLL using the standard Open File dialog.

Profiling ActiveX Controls, OLE Servers and DCOM Servers

AQtime distinguishes two types of OLE servers or ActiveX controls: In-process OLE servers (i.e., ActiveX controls) and OLE servers that are executed as a separate process (i.e. out-of-process servers). The two different types of OLE servers are profiled in different ways.

To profile an **ActiveX control or in-process OLE server**:

- Compile your ActiveX control with debug information. Preparing a Project for Profiling explains how to do this.
- Be sure the “debug” version of your control is registered in the system. If the control was compiled on your machine, it was registered during compilation. In any case, you can use the regsvr32 utility from the <Windows>\System folder to register the control.
- Add the .OCX (or .DLL) module, which includes the ActiveX control to the project being profiled: Select  **Add module...** from the Setup toolbar or context menu).
- Then perform profiling in the usual manner. Keep in mind that to profile a function (unit, class) you must check it within a profiling area or select *FULL CHECK* to profile all. Note that trigger routines are always profiled.

Out-of-process OLE servers are applications and they are executed in a separate address space. If you need to profile such programs, you should load them in AQtime as a project, not as an additional module in another project:

- Compile the out-of-process OLE server with debug information. Preparing a Project for Profiling explains how to do this.
- Open the out-of-process OLE server in AQtime as a project and specify profiling areas (see Selecting Code to Profile).
- Start profiling. Make sure that the profiled application, that is, the server, can find all additional modules it requires.
- Launch the OLE client. This is your “user” for the profiled application, the server.
- Work with the OLE client and OLE server application as needed.
- Close the client and server applications. We recommended that you first close the OLE client and then the OLE server. Since you start the OLE server from AQtime, AQtime always profiles the initialization and finalization code.

DCOM servers normally simply await a remote procedure call from a client machine. They cannot be launched by AQtime in this way. On the other hand, if they are launched as specified above for an out-of-process OLE server, they will execute nothing (no remote procedure call) and exit immediately.

The solution is to add a bit of code to the DCOM server application so that, when launched, it does not close immediately. This only means –

- adding an empty form (the Close on the caption bar will allow the application to close when you are done);
- setting up code so that on launching the DCOM application opens the form (the exact means depend on your compiler).

You can then profile by using the rest of the recipe for out-of-process OLE server. Use a client machine to command operations on the DCOM server. When you close the form on the server machine, the server process will exit and AQtime will generate its results.

Profiling IIS and PWS Applications

Once compiled with debug information, IIS (Internet Information Server) applications can be profiled by using a special technique to run them from AQtime. The same technique will allow the profiling of PWS (Personal Web Server) applications. The technique depends on whether the IIS or PWS version is 3.0 or earlier, or later. Since it is identical for IIS and PWS applications, we'll use IIS for the explanation.

IIS v. 3.0 or earlier is simple:

- Stop the IIS service from **Control Panel | Services**.
- Load the IIS application project in AQtime.
- Open the Run Parameters dialog (**Project | Parameters...** on the main menu) and specify the host application and command line switches for the IIS application. The host application will be your Internet information server, normally `<Windows>\System32\Inetsrv\inetinfo.exe`. The command line parameters must be: `-e w3svc`.
- Select your profiler and launch the IIS application from AQtime.
- When your tests are done, use the **Get Results** button to get profiling results. See *Getting Results During Testing*.

IIS v. 4.0 or later is less simple. It requires running the IIS application in debugging mode:

- Stop the IIS service from Control Panel | Services.
- Use DCOMCnfg to set the identity of the IIS Admin Service to your user account.
- Use RegEdit or some other Registry editor to make a number of modifications to the Registry. The modifications will be listed below. You should first save to .reg files the subkeys that will be modified so as to make restoration easier. As you modify each subkey, you may save it again (to a different folder, for simplicity) so that you also have pre-defined settings you can load when next you want to profile an IIS application. But you cannot save the two main keys involved as two .reg files. Too many other settings are in those branches, and they can change at any time.
- In the properties dialog of Internet Information Server set the Application Protection option for your ISAPI dll to the value *Low (IIS Process)*. This will cause the dll to run in the address space of the Internet Information Server so AQtime will be able to profile it. Otherwise, the dll will be run as a separate process.
- Load the IIS application project in AQtime.
- Open the **Run Parameters** dialog (**Project | Parameters...** on the main menu) and specify the host application and command line switches for the IIS application. The host application will be your Internet information server, normally `<Windows>\System32\Inetsrv\inetinfo.exe`. The command line parameters must be: `-e w3svc`.
- Select your profiler and launch the IIS application from AQtime.
- When your tests are done, use the **Get Results** button to get profiling results. See *Getting Results During Testing*.
- Once you have generated results, make sure you restore IIS Admin Service to its original account, and that you restore all the Registry subkeys you modified.

Registry modifications for IIS 4.0 or later:

- There are several subkeys under *HKEY_CLASSES_ROOT/AppID*, repeated under *HKEY_CLASSES_ROOT/CLSID*, that include a *LocalService* value (keyword). The three following are related to IISADMIN:

```
{A9E69610-B80D-11D0-B9B9-00A0C922E750}      // IISADMIN Service
{9F0BD3A0-EC01-11D0-A6A0-00A0C922E752}      // IIS Admin Crypto Extension
{61738644-F196-11D0-9953-00C04FD919C1}      // IIS WAMREG admin Service
```

- From the first two, remove both the *LocalService* value and the *RunAs* value.
- From the third, remove the *LocalService* value, then add a *RunAs* value and set it to *Interactive User*.
- For these three subkeys and any other under *CLSID* that has a *LocalService* value, add a sub-subkey (that is, a new subkey under the existing one, not a value) called *LocalService32*. In each case, set *Default* for these new subkeys to the full-path name of your IIS server, plus parameter *-e w3svc*. For instance --

```
c:\winnt\system32\inetsrv\inetinfo.exe -e w3svc
```

- Finally, for the following subkeys, set the *Start* value (keyword) to data contents dword:3 --

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\IISADMIN
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\W3SVC
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\MSDTC
```

Profiling Services

Windows NT and Windows 2000 support an application type called a service application. You can profile such applications with AQtime:

1. As always, compile the service application with debugger information. (See Preparing a Project for AQtime.)
2. Open an AQtime project for the service application.
3. Launch the service application from its AQtime project.
4. Wait for the service application to be instrumented by AQtime. It will appear as a process in the Task Manager, but will not actually be launched as a process until it has been instrumented.
5. Then, within 10-20 seconds, launch the service again from the Control Panel (**Start | Settings | Control Panel | Services**). At this moment, the Task Manager will display two service processes. A bit later the process launched under AQtime will become a service (normally with an onscreen message). The process launched from the SCM will not start as a service (normally generating no error message).

What is happening here is that through the SCM you are notifying Windows that your application is to be launched as a service, but you are also coming in under the wire and getting the previous AQtime launch accepted as the actual service process, rather than the launch the SCM will undertake itself.

When your AQtime-instrumented process is registered as the service process, the later launch from the SCM becomes moot, and that second process is abandoned. If too much time elapses between the actual AQtime process launch and the SCM notification, then the AQtime-instrumented process will time out and terminate silently, and the SCM launch will work in the normal way.

If you have trouble because you are calling the SCM while AQtime is still instrumenting your application, you can try a longer wait (say 40-60 seconds), or you can make instrumentation go quicker by decreasing the number of functions to be profiled, that is, the extent of the profiled Areas. (See Defining Areas To Profile).

6. Once the profiled **application** is registered as a service, test it according to your plans.
7. When your tests are done, stop the profiled service from the Service Control Manager dialog. The service application will be closed **and** AQtime will generate the profiling results. Or you may prefer to use **Project |**


Get Results... from the main menu to generate results without closing the process. (See *Getting Results During Testing*.)

Note that service applications are often OLE servers. It may be easier simply to profile these as OLE servers. See *Profiling ActiveX Controls and OLE Servers*.

Profiler Options

Almost all AQtime profilers have options. These are set from one central dialog, the Profiler Options Dialog, with is brought up from the main menu by selecting **Options | Profiler Options...**:

The dialog is in two parts -

On the left is the profiler selector, or the right the options dialog for that profiler. The profiler is the same one you from the  dropdown list on the standard toolbar. It is organized as a tree view, with the main branches for the various profiler classes, and the leaves for the actual profilers. To see and select a profiler, you must first open its branch by clicking on the + symbol to the right. Using the navigation sidebar, select the profiler whose options you wish to change.

On the right is the list of options for the selected profilers. When you have made the modifications you wanted, press OK to apply the changes. If you only wanted to check the existing options, press Cancel to exit.

Coverage Profiler Options

The Function Coverage, Line Coverage (grouped by function) and Line Coverage (grouped by file) profilers provide a single option - **Warning level**. This option operates during the preliminary (instrumentation) phase of each profile run. It sets a number of functions or lines currently selected for profiling, beyond which a dialog box will ask you whether you wish to continue the run as specified, or to exit immediately. This is to allow you an early exit from an over-ambitious Areas specification which, might lead to sluggish execution, as masses of needless functions or lines are logged by the profiler. The valid range for *Warning level* is between 0 (never warn) and 1,000,000.

Function Profiler Options

- **Show non-hit functions** – Enables or disables the display, in the Report panel, of routines that have not been executed in the current profile run.
- **Warning level** – This option operates during the preliminary (instrumentation) phase of each profile run. It sets a number of functions currently selected for profiling, beyond which a dialog box will ask you whether you wish to continue the run as specified, or to exit immediately. This is to allow you an early exit from an over-ambitious Areas specification, which might lead to sluggish execution, as masses of needless functions are logged by the profiler. The valid range for *Warning level* is between 0 (never warn) and 1,000,000.
- **% with children relative to real time** – If this is enabled, **% With Children** will be figured relative to the total Time (without children). Else, relative to the total Time. See *Calculating Percents In the Report panel*. This option has no effect on results already calculated and displayed.
- **Include body time in Details** – Sets whether results for each function's own-code ("body") time will be listed along with the child-call results in the Details panel.
- **Smallest percent identified** – For each child function of any parent function, sets the minimum proportion of the time spent by the parent (child calls included) that the child must have occupied, to be listed separately in the *children* pane of the Details panel.
- **Sort functions in Call Graph by** – Column on which functions will be sorted in the Call Graph panel. The functions are sorted in the descending order for the values in this column.

- **Track recursion depth** - Sets whether recursion data will be tracked so that the maximum recursion depth reached for each function can be shown in the **Max Recursion Depth** column of the Report panel.
- **Calibration** – Settings for the timing loop run at the beginning of each Function Profiler run. This estimates the profiling overhead time that must be subtracted to figure reported times.
 - **Loop Count** – Number of loops to use for calibration 1 to 100,000, default 1000. More loops may improve calibration, at the cost of startup time.
 - **Tolerance** – Variation between calibration loops above which the calibration cycle will be rerun. 0% to 99%, default 15%. Lower values may improve reliability, at the cost of startup time. Values from 10 to 20 tend to give the best results.
- **Display in Editor gutter** – Selects results to display in the gutter of the Editor panel, at the beginning of the implementation code of each function. Choices: **HitCount**, **Total Time With Children**, **% Time With Children**, **Total Time**, **% Time**.
- **Display in Call Graph** - Selects results to display in the function rectangles of the Call Graph panel. Any combination of the following can be displayed: **Time**, **Time with Children**, **Average Time**, **Max Time**, **Min Time**, **Exceptions**, **Max Recursion Depth**.

Function HitCount Options

- **Warning level** – This option operates during the preliminary (instrumentation) phase of each profile run. It sets a number of functions currently selected for profiling, beyond which a dialog box will ask you whether you wish to continue the run as specified, or to exit immediately. This is to allow you an early exit from an over-ambitious Areas specification, which might lead to sluggish execution, as masses of needless functions are logged by the profiler. The valid range for *Warning level* is between 0 (never warn) and 1,000,000.
- **Track recursion depth** - Sets whether recursion data will be tracked so that the maximum recursion depth reached for each function can be shown in the **Max Recursion Depth** column of the Report panel.
- **Call relationship tracking** – Settings for the added information displayed by the Details panel.
 - **Enable** - Sets whether parent-child information will be logged and accumulated for each call, to be displayed (by parent and by child, for each function) in the Details and Call Graph panels.
 - **Smallest percent identified** - Sets the minimum call frequency, for each function, needed for a parent or child to appear as a separate line in the Details panel. This is relative to all parents or all children of the function for which details are being displayed.
- **Display in Call Graph** – Selects results to display in the function rectangles of the Call Graph panel. Available: **Hit count**, **Max recursion depth**.

Line HitCount Profiler Options

- **Warning level** – This option operates during the preliminary (instrumentation) phase of each profile run. It sets a number of lines currently selected for profiling, beyond which a dialog box will ask you whether you wish to continue the run as specified, or to exit immediately. This is to allow you an early exit from an over-ambitious Areas specification, which might lead to sluggish execution, as masses of needless lines are logged by the profiler. The valid range for *Warning level* is between 0 (never warn) and 1,000,000.
- **Max HitCount** - Sets the number of hits for a single line after which its hits will not be recorded anymore (thus saving time). 0 to 1,000,000. 0 = count always.

Sampling Profiler Options

- **Calibration Loop Count** – Number of loops to use during startup to figure the conversion ratio from sample counts to real time. 100 to 10,000,000, default 1000. More loops may improve calibration, at the cost of startup time.
- **Sampling interval (clock ticks)** – Approximate ticks between samples. Higher values will give invalid results for quick, infrequent functions (wide-mesh net effect). Depending on the system, lower values can lead to skipped samplings (the system may be busy elsewhere). 10 to 100,000.
- **Shift size** - Number of trailing bits to remove from code addresses before matching them to a function. High values will miss small functions but save memory when profiling large binary areas. 2 to 10 bits. The *Shift size* sets the grain size for identifying code addresses. For instance, *Shift size* 4 means that the currently executing address at sample time is rounded down to the closest 16-byte-even address. The function or line attribution is for this rounded-down address. Anything occurring in the next 15 bytes is treated as belonging to that function or line. Thus, allowed values go from four bytes (2) to a kilobyte (10). 2 is the general recommendation, but the *Sampling interval* must also be narrow enough. Larger *Sampling Intervals* make low Shift Sizes statistically useless.
- **Gutter information** - Sets whether the gutter in the Editor panel will show the sample count or the time count at the beginning of each function or line.

Function Trace Profiler Options

- **Warning level** – This option operates during the preliminary (instrumentation) phase of each profile run. It sets a number of functions currently selected for profiling, beyond which a dialog box will ask you whether you wish to continue the run as specified, or to exit immediately. This is to allow you an early exit from an over-ambitious Areas specification, which might lead to sluggish execution, as masses of needless functions are traced by the profiler. The valid range for *Warning level* is between 0 (never warn) and 100,000.
- **Max HitCount** - Sets the number of hits for a single function after which its hits will not be recorded anymore (thus saving time). 0 to 100,000. 0 = count always.
- **Trace with parameters** – Sets whether parameter values for calls will be recorded and shown. (On a method call, the first parameter will be the instance address).
- **Show call number** - Sets whether each call of a function will be counted and the current count shown.
- **Hierarchy display location** – Selects where the call hierarchy will be displayed onscreen during the run, and whether it will also go on file. Press the ellipsis button and select any combination of the following:
 - *External Console* - Displays either in CodeSite window or in Overseer window.
 - *File* - Logs all calls to the file specified by the *Log file name* option.
 - *Event View panel* - Logs calls to the Event View panel.
- **Log file name** – Name (path optional) of text file to which to write the entire series of calls logged.

VCL Profiler Options

- **Classes** - By default, the VCL Reference Count profiler works with descendants of the TInterfacedObject class only. It monitors calls to the *AddRef* and *Release* implementation methods of this class. The *Classes* option allows you to include independent base classes; each must implement IUnknown. See also *Base Classes Dialog* in on-line help.
- **Stack**

- **Show call stack** - Sets whether the call stack contents will be shown in the Details panel for leaked calls. For the VCL Class profiler, a leaked call is a Create call for which there was no matching Free. For the VCL Reference Count profiler, a leaked call is an `_AddRef` call for which there was no matching `_Release`. Call Stack tracing can significantly slow down the profiled application. If you are only interested in class or interface usage (how many classes have been created, etc.), you can disable it.
- **Show all parents** - If the call stack is shown, sets whether it will include callers for which there is no debug info.
- **Depth shown** – If the call stack is traced (see *Show call stack*), this option specifies the number of traced procedures in the stack. The less the depth, the faster the speed is. The default value is 20. 40 is the maximum value. 0 means no tracing.
- **Rely on stack frames** – This option indicates that the profiled application has been compiled with stack frames. In this case, AQtime uses stack frames to track the hierarchy of function calls. Else it must resort to a slower and perhaps less accurate algorithm.
- **Max leak count** – Sets the number of leaks for a single class or interface to be reported in the *Instances* pane of the Details panel (saving time by not repeating unneeded detail). This has no effect on the simple counts shown in the Report panel. 0 to 100000. 0 = always record.

Platform Compliance Options

- **Customize settings before profiling** – Sets whether to display the Platform Compliance Settings dialog at the start of each run.
- **Compliance level** - Sets what type of API call to check for. See Platform Compliance Analysis.

Unused VCL Units Profiler Options

- **Ignore units with names containing** – Specifies the string used to exclude units from analysis. If a unit name includes a string specified by this option, the profiler regards this unit as used. You can indicate several strings here. Use commas to separate them. The default string for the option is 'const,type,messages,comstr', that is the Unused VCL Units profiler ignores all units whose names include either a 'const', 'type', 'messages' or a 'comstr' string. An alternative to this option is using the IgnoredUnits.txt file (See the *Unused VCL Units Profiler - Principles of Operation*).
- **Initialization lines discarded** – This option is effective only for user-defined units. It specifies the available number of lines in the *initialization* section of a unit. If the number of lines in the *initialization* section is less than this number, the unit is considered unused.
- **Initialization bytes discarded** - This option is effective only for user-defined units. It specifies the available size of the *initialization* section (in bytes). If the size of the *initialization* section is less than this number, the unit is considered unused.
- **Delphi version** – Specifies the Delphi version used to compile an application. Possible values include 3, 4 and 5.
- **Show error if source not found** – Specifies whether the profiler displays Path Info Dialog when it cannot find source files for certain units. In most cases, the profiler reports that the application includes some VCL units that are not used in other VCL units. If you are not interested in messages about such errors, you can turn off the *Show error if source not found* option and tune the Search Directories to your project only. In this case, the profiler will report only about unused units in your units.

ATL RefCount Profiler Options




- **Show call stack** - Sets whether the call stack will be recorded at runtime for calls to *AddRef* or *Release*. If this is enabled, the call stack will be available in the Details panel for all calls involving a reference leak. The cost is that recording the call stack may slow execution. If you only need class usage statistics, rather than to debug leaks, then you may disable this option.
- **Show all parents** - This option has no effect unless *Show call stack* is enabled. Then, if *Show all parents* is enabled, all callers will be included in the Call Stack pane of the Details panel. Else, callers for which there is no debug info will be omitted, making the list more compact.

BDE SQL Profiler Options

- **Show call stack** Sets whether the BDE SQL profiler should track the call stack for each SQL call, in order to display it in Details at the end of profiling. This may slow down the application being profiled.
- **Show all parents** If *Show call stack* is enabled, some functions in the call stack may have been compiled without debug info. If *Show all parents*, the functions will be have lines in Details, but without specific info. If not, they will simply be skipped, making the display more compact.
- **Rely on stack frames** If *Show call stack* is enabled and the application has been compiled with stack frames enabled, then this option should be on. If *Show call stack* is enabled but the application was not so compiled, the option must be set to off, else errors will occur in the stack tracking, quickly making it useless. If the option is off, the BDE SQL profiler will use the best algorithm practicable to track the stack correctly, but errors may still occur. If *Show call stack* is disabled, this option has no meaning.

Memory and API Resource Check Profiler Options

- **Stack**
 - **Show call stack** - Sets whether the Details panel will track the hierarchy of function calls at runtime and display the call stack when reporting. Call stack tracking can slow down the profiled application; you may disable this option when you are only interested in resource usage. But see *Depth shown*. Note that if the call stack is not tracked, some filters may be inoperative. See *Leak Filters Dialog* and *Predefined Filters*.
 - **Show all parents** – Sets whether the Call Stack will include callers for which there is no debug info, rather than skip them.
 - **Rely on stack frames** - This option indicates that the profiled application has been compiled with stack frames. In this case, AQtime uses stack frames to track the hierarchy of function calls. Else it must resort to a slower and perhaps less accurate algorithm.
 - **Depth shown** – See *Show call stack*. If the stack is being tracked, this option sets how deep it will be tracked, that is how many callers-of-callers will be recorded. The less depth, the more speed. 0 means no tracking, 40 is the maximum, 20 the default.
 - **Cache check size** - When the profiler unwinds the stack, it keeps the unwound functions for later re-use. On the next unwinding, it checks a number of function addresses from the top of the stack. If they are the same as those in the cache, then it assumes the entire cache is still valid. This saves time and resources. The Stack Cache Check Size is the number of functions that will be checked each time to determine if the cache is still valid. The range is 0 to 40. 0 means "do not use cache". 40 makes the cache nearly pointless.
- **Customize settings before profiling** - Sets whether AQtime displays the Settings dialog every time you start profiling.
- **Show allocation parameters** – Sets whether the Resource column in the Report panel displays parameter values used for the function call.

- **Show API parameters** – Sets whether the profiler will record parameters and result values of resource-related function calls. This will also allow the Report panel to show warnings  warnings or  errors (see below). These warnings and errors are defined in the database you can edit using WinAPI Database Editor.
- **Show API warnings** – If *Show API parameters* is enabled, this sets whether  will be displayed. This option doesn't affect showing errors.
- **Profile memory management routines** – If this option is checked, AQtime monitors functions that create, change and dispose of memory blocks.
- **Profile new loaded modules** – If this option is checked, AQtime analyzes modules loaded into memory after the start of profiling. Else, it analyzes only modules selected in the **Modules** page of the Settings dialog.
- **Check memory bounds** – If this option is checked, AQtime traces whether the profiled application writes to memory below or above the allocated bounds of a memory block and whether it releases the allocated memory correctly. See *Checking Bounds of Memory Blocks*.
- **Ignore system modules** – If this option is on, the Memory and API Resource Check profiler does not profile memory allocations, resources and function calls made within system dlls, i.e. within modules located in the <Windows>\System32 folder. The **Modules** page of the Settings dialog does not display system libraries either except for MFC42D.DLL and VCLxx.BPL.

INDEX

| | |
|---|----------|
| % with children relative to real time | 144, 157 |
| 3D view | 86 |

A

| | |
|---|------------|
| About macros | 60 |
| Absent module..... | 68 |
| Activate after loading | 87 |
| Active | 84, 85 |
| Auto-merge | 85 |
| ActiveX controls -- Profiling..... | 154 |
| Add Trigger dialog | 74 |
| Adding..... | 74 |
| Columns to AQtime panels..... | 74 |
| Results to source files | 78 |
| Series to the Graph panel | 75 |
| Allow zoom..... | 85 |
| Always set up Compare params | 85 |
| Analyzing profiler results | 36 |
| Animated zoom | 85 |
| AQtest | 59, 61 |
| AQtime | |
| as an OLE server..... | 142 |
| Panels..... | 19, 20 |
| Profilers..... | 20 |
| AQtime Overview | 7 |
| AQtime.idl..... | 142 |
| AQtime.tlb..... | 142 |
| AQtime_TLB.pas | 142 |
| AQtimeCOMAPIProvider..... | 142 |
| Enable and disable profiling from application code..... | 142 |
| AQTimeSetThreadName..... | 152 |
| Areas | 21, 30, 31 |
| Excluding areas..... | 30 |
| Including areas | 30 |
| Profiling | 30, 32 |
| Arranging columns and lines | |
| Adding columns to panels..... | 74 |
| Displaying results in different formats..... | 74 |
| Formatting columns | 74 |
| Grouping | 79, 80 |
| Removing columns from panels | 74 |
| Sorting..... | 81 |
| ATL applications | |
| Compiler Settings - ATL RefCount profiler | 123 |
| Preparing for Memory and API Resource Check..... | 129 |
| ATL Reference Count profiler | 21, 121 |
| Compiler Settings..... | 123 |
| Description..... | 121 |
| Details panel | 122 |
| Options..... | 161 |

| | |
|--------------------------------|--------|
| Auto expand..... | 83, 85 |
| Auto start application..... | 86 |
| Auto-merge | |
| Active | 85 |
| Folder name | 85 |
| Auto-select new elements | 88 |

B

| | |
|---|--------------------|
| Background color..... | 83, 84, 85, 86, 87 |
| Details panel | 83 |
| Disassembly panel | 84 |
| Explorer panel | 85 |
| Graph panel | 86 |
| PEReader panel | 87 |
| Report panel..... | 87 |
| BDE SQL profiler..... | 22, 124 |
| Options | 161 |
| Binary instrumentation..... | 21 |
| Borland C++ | 12, 23 |
| Build with run-time packages option | 129, 133 |
| By Class Name..... | 39, 40 |
| By Source File..... | 39, 40 |
| By Unit Name | 39 |

C

| | |
|---|------------|
| C++ | 22, 23, 26 |
| Borland C++ | 23 |
| C++Builder | 22 |
| Compiler Settings - ATL RefCount profiler | 123 |
| GCC..... | 26 |
| Profiling VC++ Applications - Memory and API Resource | |
| Check..... | 129 |
| Visual C++..... | 23 |
| C++Builder | 12 |
| Compiler Settings | 22 |
| Compiler Settings – ATL RefCount profiler | 123 |
| Cache check size | 161 |
| Calculating percent time with children | 144 |
| Calibration..... | 158 |
| Function Profiler..... | 158 |
| Loop count..... | 159 |
| Loop Count..... | 158 |
| Tolerance | 158 |
| Calibration Loop Count | 159 |
| Call Graph panel | 42 |
| Options | 82 |
| Call relationship tracking | 99, 158 |
| Enable..... | 158 |
| Smallest percent identified | 158 |
| Call stack..... | 128 |

| | |
|---|-----------------|
| ATL RefCount profiler | 123 |
| BDE SQL profiler | 125 |
| Memory and API Resource Check | 126, 128 |
| VCL Class profiler | 47, 108 |
| VCL RefCount profiler | 48, 109 |
| Call stack on exceptions | 84 |
| Chart | 58, 59 |
| Chart style | 83 |
| Check memory bounds | 135, 162 |
| Checking bounds of memory blocks | 134 |
| Checking Elements to Profile | 32 |
| Checking resources and memory | 125 |
| Child function color | 83 |
| Child functions | 32 |
| Classes | 159, 160 |
| Clear on application start | 85 |
| Clipboard -- copying charts to | 58 |
| Code Editor | 52 |
| CodeSite | 104, 105 |
| COFF debug format | 23, 24 |
| Column Selection dialog | 74 |
| Columns | 74 |
| Adding and removing | 74 |
| Format | 74 |
| Common Series Color | 87 |
| Compare results | 55, 75, 76 |
| Compiler Settings | 22, 23, 26, 123 |
| ATL RefCount profiler | 123 |
| Borland C++ | 23 |
| Borland C++Builder | 22 |
| Borland Delphi | 22 |
| GCC | 26 |
| Microsoft Visual Basic | 25 |
| Microsoft Visual C++ | 23 |
| Compiler Settings for Visual Basic | |
| Debug info, generated as an external PDB file | 25 |
| Debug info, included into the executable file | 25 |
| Compiler Settings for Visual C++ | |
| Embedded debug information | 24 |
| Generating debug info as an external DBG file | 24 |
| Generating debug info as an external PDB file | 24 |
| Compilers | 12 |
| Compliance level | 160 |
| Context menus | 17 |
| Controlling what to profile | 28 |
| Copying results | 78 |
| Counter View | 64 |
| Coverage profilers | 21 |
| Description | 91 |
| Options | 157 |
| Covered -- view | 39 |
| Create Filter Condition Dialog | |
| Memory and API Resource Check page | 135 |
| Current - view | 39 |
| Current View | 40 |
| Customize settings before profiling | 160, 161 |
| Cycling trigger option | 34 |

D

| | |
|---|----------|
| DBG debug format | 23 |
| DBG debug info | 151 |
| DBGHELP.DLL | 24 |
| DCOM servers -- Profiling | 154 |
| Debug information | 22 |
| Generated as an external PDB file | 25 |
| Included into the executable file | 25 |
| PDB and DBG files | 151 |
| Stab format | 26 |
| Default - view | 39 |
| Defective module | 68 |
| Defining areas to profile | 30 |
| Delay between events (ms) | 86 |
| Delphi | 12, 22 |
| Delphi 3.0 -- Memory and API Resource Check | 137 |
| Delphi 4.0 -- Memory and API Resource Check | 137 |
| Delphi 5.0 -- Memory and API Resource Check | 137 |
| Delphi version | 160 |
| Depth shown | 160, 161 |
| Details panel | 43 |
| ATL RefCount profiler | 122 |
| BDE SQL profiler | 124 |
| Displaying results as % value or bar graph | 74 |
| Formatting columns | 74 |
| Function HitCount | 45 |
| Function Profiler | 43 |
| Memory and API Resource Check | 126 |
| Options | 83 |
| Unused VCL Units | 119 |
| VCL Class Profiler | 47 |
| VCL Reference Count Profiler | 48 |
| Disable/Enable Profiling button | 28 |
| Disassembly panel | 50 |
| Displaying results as % value or bar graph | 74 |
| Formatting columns | 74 |
| Options | 83 |
| Display in Call Graph | 158 |
| Function Profiler | 158 |
| Display in Editor gutter | 158 |
| Display source code | 83 |
| Displaying parameters of function calls | 125 |
| Memory and API Resource Check | 125 |
| Displaying results in different formats | 74 |
| DLL | 153 |
| Profiling | 153 |
| Docking | 18 |
| Docking Allowed | 20 |
| Restore Default Docking | 18 |
| Doing One Profile Run | 35 |
| Duplicated module | 68 |

E

| | |
|---------------------------|----|
| Editor | 86 |
| Macro Engine option | 86 |
| Editor panel | 52 |

| | |
|--|------------|
| Font | 84 |
| Options | 84 |
| Editor Panel | |
| Font | 84 |
| Elements to Profile -- checking | 32 |
| Embedded debug information | 24 |
| Enable/Disable Profiling button | 28 |
| Enabling and disabling profiling from application code | 142 |
| Event View panel | 53, 117 |
| Options | 84 |
| value of the Hierarchy display location option | 159 |
| Excel | 78 |
| Exporting results to | 78 |
| Exception Tracer | 117 |
| Exceptions | 84 |
| Active | 84 |
| Find call stack from frames | 85 |
| Hide IsBadPtr exceptions | 85 |
| in the Event View panel | 84 |
| Include no-debug in stack | 85 |
| Max consecutive exceptions | 84 |
| Show call stack | 84 |
| Excluding "system" files and functions | 29 |
| Explorer panel | 55 |
| Options | 85 |
| Exported functions | 67, 68 |
| Exporting | |
| Charts | 58 |
| Results | 55, 58, 78 |
| External Console | 159 |
| Extra memory usage restriction | 133 |

F

| | |
|--|----------------|
| Field Chooser menu option | 74 |
| File | |
| value of the Hierarchy display location option | 159 |
| File extensions for highlighting | 53 |
| Filter Item - Menu item | 136, 137 |
| Filtering results | 78 |
| Memory and API Resource Check | 135 |
| Find call stack from frames | 85 |
| Find Matching Methods | 105 |
| Finding | 81, 118 |
| Unused units in your application | 118 |
| Values in results | 81 |
| Fit | 43 |
| Flat grid | 83, 84, 87 |
| Report panel | 87 |
| Folder name | 85 |
| Font | 84 |
| Disassembly panel | 84 |
| In Editor panel | 84 |
| Font color | 83, 84, 85, 87 |
| Details panel | 83 |
| Disassembly panel | 84 |
| Explorer panel | 85 |
| PEReader | 87 |

| | |
|--------------------------------------|---------|
| Report panel | 87 |
| For All Threads trigger option | 34 |
| Formatting columns | 74 |
| Function Coverage profiler | 21, 91 |
| Description | 91 |
| Options | 157 |
| Results | 91 |
| Function HitCount profiler | 96 |
| Description | 96 |
| Details panel | 45 |
| Options | 158 |
| Function Information Panel | 68 |
| Function Profiler | 21, 100 |
| Description | 100 |
| Details panel | 43 |
| Options | 157 |
| Function rectangle background | |
| Details part | 83 |
| Header part | 83 |
| Function Sampling profiler | 111 |
| Description | 111 |
| Options | 159 |
| Function Trace profiler | 21, 104 |
| Description | 103 |
| Options | 159 |
| Function Trace Profiler | 104 |

G

| | |
|--------------------------------------|--------|
| GCC | 12, 26 |
| Compiler settings | 26 |
| Generating debug info | |
| As an external DBG file | 24 |
| As an external PDB file | 24 |
| Getting results during testing | 144 |
| Getting Started | 20 |
| Getting Support | 13 |
| GNU CC | 12, 26 |
| Go to Child Procedure | 43 |
| Go to Current Procedure | 43 |
| Go to Parent Procedure | 43 |
| Gradient | 86 |
| Active | 86 |
| Bottom color | 86 |
| Top color | 86 |
| Graph panel | 58 |
| Options | 85 |
| Properties | 85 |
| Series | 75 |
| Graph Panel | |
| Series | 75 |
| Graph View | 64, 65 |
| Grid settings | 83 |
| Details panel | 83 |
| Disassembly panel | 83 |
| Grouping results | 79 |
| Gutter font | 84 |
| Gutter information | 159 |

Gutter size in pixels 84

H

Headers 70
 Help 13
 Hex 51
 Hide IsBadPtr exceptions 85
 Hierarchy display location 54, 159
 Highlight 83
 Active 83
 Highlighting 84
 Histogram option 87
 Histogram View 65
 HitCount profilers 21, 96
 Description 96
 Options 158
 Hook extensions 130
 HTML 78
 Exporting results to 78

I

Ignore system file settings 29
 Ignore system modules 132, 162
 Ignore units with names containing 160
 IIS applications 144, 155
 Getting profiling results 144
 Profiling 155
 Imported functions 66, 69
 Importing results 55
 Include body time in Details 157
 Include no-debug in stack 85
 Increasing profiling speed 145
 Incremental search 81
 Initial Profiling Status for New Threads 33, 34
 Initial Profiling Status for Starting Thread 33, 34
 Initialization bytes discarded 160
 Initialization lines discarded 160
 Inline functions -- Profiling 149
 In-process servers -- Profiling 154
 Inserting Profiling Results into Source Code 80
 Install Extensions dialog 15
 Installation 12
 Installing Extensions 15
 Instruction 51
 Integration dialog 13
 Internet Information Server applications
 Profiling 155
 Interpret addresses 83
 Introduction 7
 ISAPI applications 155
 Profiling 155
 Items to Profile -- checking 32

L

Language for new macros 86
 Leak Filters dialog 136
 Memory and API Resource Check 135

Leak resources restriction 137
 Leaked Classes Only 39
 Line Coverage (Grouped by File) 21, 91
 Description 91
 Options 157
 Results description 94
 Line Coverage (Grouped by Function) 21
 Description 91
 Options 157
 Results description 93
 Line Coverage profiler 21
 Line HitCount profiler 96, 98
 Description 96
 Options 158
 Line Sampling profiler
 Description 111
 Options 159
 Link environment variable 25
 Linked DLLs 66
 Load Desktop 18
 Load From File 58
 Loading results from a file 78
 Log file name 159
 Loop Count 158

M

mac files 59
 Macro Engine
 About macros 60
 Description 59
 Macro Engine Panel 59
 Macro Recording and Playback 61
 Options 86
 Window and Process Recognition 61
 Macros 60, 61
 About 60
 Main menu 16
 Mark selected lines 83, 84, 87
 Reprot panel 87
 Max consecutive exceptions 53, 84
 Max HitCount 158, 159
 Max leak count 160
 Memory
 Tracing the usage 125
 Memory and API Resource Check profiler 21, 23
 Checking bounds of memory blocks 134
 Create Filter Condition dialog 135
 Description of results 126
 List of Checked Functions 138
 Non-existent resources in the Report panel 138
 Options 161
 Packages 132
 Predefined filters 137
 Preparing ATL applications 129
 Profiling memory management routines 133
 Profiling VC++ applications 129
 Profiling VCL Applications 129

| | |
|--|-------------|
| Resource leaks in the Report panel | 137 |
| Settings Dialog | 130 |
| Win API Database Editor | 125 |
| Memory and API Resource Check Profiler | 137 |
| Memory manager | 133 |
| Merge results | 55, 75, 77 |
| Modify Graph panel series | 75 |
| Modules Hierarchy Panel | 68 |
| Modules page – Settings dialog | 131 |
| Monitor | |
| Counter View | 64 |
| Monitor panel | |
| Description | 62 |
| Graph View | 64 |
| Options | 86 |
| Monitor Panel | |
| Histogram View | 65 |
| Monochrome | 86 |
| Mops | 51 |
| MSVC 6.0 MFC Applications -- Memory and API Resource | |
| Check | 137 |
| Multithreaded applications | 36, 37, 152 |
| Assigning names to threads | 152 |
| Profiling | 152 |
| Result display | 36 |

N

| | |
|--|--------|
| Navigating AQtme panels | 19 |
| No results for Function HitCount in Call Graph | 42, 96 |
| No Zoom | 43 |
| Non-Existent Resources in the Report panel | 138 |
| Notes panel | |
| Memory and API Resource Check | 126 |
| Number of child levels | 82 |
| Number of note lines | 83 |
| Number of parent levels | 82 |
| Number of recent results to keep | 85 |

O

| | |
|---|----------|
| Off-Triggers | 33, 34 |
| OLE server | 142, 143 |
| Profiling | 154 |
| Using AQtme as | 142 |
| On-Line Help | 13 |
| On-Triggers | 33, 34 |
| Opening a Project in AQtme | 27 |
| Optimizing the profiling process | 145 |
| Options | 85 |
| Ordinary module | 68 |
| Out-of-process servers -- Profiling | 154 |
| Overloaded functions | 149 |
| Overseer | 104, 105 |
| Overview | |
| AQtme | 7 |
| Panels | 19 |
| Profilers | 20 |

P

| | |
|---|------------------------|
| Packages | 22, 129, 130, 133 |
| Compiler settings for C++Builder | 22 |
| Compiler settings for Delphi | 22 |
| Memory and API Resource Check | 130 |
| Panel Options | 82 |
| Call Graph panel | 82 |
| Details panel options | 83 |
| Disassembly panel options | 83 |
| Editor panel options | 84 |
| Event View options | 84 |
| Explorer panel options | 85 |
| Graph panel | 85 |
| Macro Engine panel | 86 |
| PEReader | 87 |
| Real-Time Monitor Panel | 86 |
| Report panel options | 87 |
| Setup panel options | 87 |
| Panels | 43, 50, 51, 56, 58, 71 |
| Call Graph Panel | 42 |
| Details panel | 43 |
| Disassembly panel | 50 |
| Disassembly Panel | 52 |
| Editor panel | 52 |
| Explorer Panel | 56 |
| Graph panel | 58 |
| How To | 74 |
| Macro Engine Panel | 59 |
| Navigating | 19 |
| Organization | 37 |
| Overview | 19 |
| Panel Options | 82 |
| Real-Time Monitor | 62 |
| Report Panel | 71 |
| Setup panel | 72 |
| Setup Panel | 73 |
| Parameters of function calls | 125 |
| Memory and API Resource Check | 126 |
| Parent function color | 83 |
| Pass Count trigger option | 34 |
| PDB debug format | 23, 25, 151 |
| PE Information | 70 |
| Percent with children -- Calculating | 144 |
| PEReader | 20 |
| Description | 66 |
| Function Information | 69 |
| List of imported and exported functions | 68 |
| Modules Hierarchy | 68 |
| Options | 87 |
| PE Information | 70 |
| PEReader panel | 67 |
| PEReaderPlug-In | 66 |
| Personal Web Server applications | |
| Profiling | 155 |
| Platform Compliance Analysis | 115 |
| Description | 115 |

| | | | |
|---|-----------------|--|----------------------|
| Options..... | 160 | Static Analysis | 20, 89 |
| Play shortcut..... | 86 | Unused VCL Units | 118 |
| Playback Macro..... | 61 | VCL Class profiler..... | 107 |
| Predefined filters | | VCL profilers..... | 107 |
| Memory and API Resource Check..... | 137 | VCL Profilers | 21 |
| Preparing your project | | VCL Reference Count profiler | 107 |
| Borland C++ | 23 | Profiling | 28, 29, 145, 146 |
| Borland C++Builder | 22 | % with children..... | 144 |
| Borland Delphi..... | 22 | ActiveX controls..... | 154 |
| GCC | 26 | Controlling what to profile | 28 |
| Visual Basic | 25, 26 | DBG files..... | 151 |
| Visual C++..... | 23 | DCOM servers..... | 154 |
| Principles of operation | | DLLs..... | 153 |
| Unused VCL Units profiler..... | 120 | Enable/disable..... | 29 |
| Printing charts | 58 | Getting results during profiling..... | 144 |
| Procedures Covered less than 50%..... | 39 | IIS applications | 155 |
| Profile memory management routines..... | 162 | Increasing profiling speed..... | 145 |
| Profile new loaded modules | 162 | Inline functions | 149, 151 |
| Profiler Options | 157 | Memory management routines - Memory and API Resource | |
| ATL Reference Count Profiler..... | 161 | Check..... | 133 |
| BDE SQL profiler | 161 | Multithreaded applications | 152 |
| Coverage profilers..... | 157 | OLE servers | 154 |
| Function HitCount | 158 | Overloaded functions - Profiling | 149 |
| Function Profiler | 157 | PWS applications..... | 155 |
| Function Trace profiler | 159 | Recursive Functions | 146 |
| Line HitCount profiler | 158 | Services..... | 156 |
| Memory and API Resource Check..... | 161 | SQL queries | 124 |
| Platform Compliance Analysis | 160 | Stored procedures | 124 |
| Sampling Profiler | 159 | Tips | 145 |
| Unused VCL Units..... | 160 | VC++ Applications using Memory and API Resource | |
| Unused VCL Units profiler..... | 160 | Check..... | 129 |
| VCL Profilers..... | 159 | VCL Applications using Memory and API Resource Check | |
| Profiler Options Dialog | 157 | | 129 |
| Profiler results | 36 | What to profile -- controlling..... | 28 |
| Profilers | 20, 35, 91, 118 | With PDB or DBG debug Info | 151 |
| ATL Reference Count..... | 21, 121 | Profiling areas | 30 |
| BDE SQL profiler | 22, 124 | Excluding areas..... | 30 |
| Binary instrumentation..... | 21 | Including areas..... | 30 |
| Coverage profilers..... | 21, 91 | Project Filter | |
| Exception Tracer..... | 117 | Memory and API Resource Check | 135 |
| Function HitCount | 96 | Project Search Directories..... | 120 |
| Function Profiler | 21, 100 | Project Setup | 27 |
| Function Sampling..... | 110 | Properties | 85 |
| Function Trace | 21, 103 | PWS applications | 155 |
| HitCount profilers | 21, 96 | | |
| Line Coverage (Grouped by File) | 94 | R | |
| Line Coverage (Grouped by Function) | 91 | Real-Time Monitor | 22, 62, 63, 126, 134 |
| Line Coverage results | 93 | Counter View..... | 64 |
| Line HitCount | 96 | Description | 62 |
| Line Sampling..... | 110 | Graph View..... | 64 |
| Memory and API Resource Check..... | 21, 125 | Histogram View..... | 65, 66 |
| PEReader | 20, 66 | Options | 86 |
| Platform Compliance Analysis | 115 | Rebase.exe utility | 24 |
| Profiling areas | 21 | Record all events..... | 86 |
| Real-Time Monitor | 22, 62 | Record application events | 86 |
| Sampling profilers..... | 20, 110 | Record AQtime events | 86 |
| Selecting..... | 35 | Record shortcut | 86 |

| | | | |
|---|-------------------|---|--------------|
| Recording Macro..... | 61 | Searching..... | 81, 118 |
| Recursive Functions..... | 146 | Unused units in your application..... | 118 |
| Reference Count Profiler..... | 107, 109, 110 | Using incremental search..... | 81 |
| Refresh Intervals (ms)..... | 86 | Values in results..... | 81 |
| Relative to real time..... | 144 | Sections..... | 70, 71 |
| Rely on stack frames..... | 160, 161 | Select code to profile..... | 30 |
| Remove Graph panel series..... | 75 | Selecting a profiler..... | 35 |
| Removing columns in panels..... | 74 | Selecting Several Records in a Panel..... | 75 |
| Report panel..... | 74 | Series in the Graph panel..... | 75 |
| Displaying results as % value or bar graph..... | 74 | Adding new..... | 75 |
| Formatting columns..... | 74 | Removing..... | 75 |
| Options..... | 87 | Services..... | 144, 156 |
| Report Panel..... | 71 | Getting profiling results..... | 144 |
| Resource leaks restriction..... | 137 | Profiling..... | 156 |
| Resources..... | | Setting Up Triggers..... | 34 |
| Tracing the usage..... | 125 | Settings dialog..... | 129 |
| Restore Default Docking..... | 18 | Settings Dialog -- Memory and API Resource Check..... | 130 |
| Restrictions..... | 133, 137, 138 | Setup panel..... | |
| Extra memory usage restriction..... | 133 | Options..... | 87 |
| Leak resources restriction..... | 137 | Setup Panel..... | 30, 72 |
| Non-existent resources in the Report panel..... | 138 | Setup profiling areas..... | 30 |
| PDB and DBG files..... | 151 | Shift size..... | 114, 159 |
| Results..... | 37, 38, 56, 74 | Shortcut menus..... | 17 |
| % with children..... | 144, 145 | Show all parents..... | 160, 161 |
| Adding as comments to source files..... | 78 | Show allocation parameters..... | 161 |
| Analyzing..... | 36 | Show API parameters..... | 162 |
| Comparison..... | 76, 77 | Show API warnings..... | 162 |
| Copying..... | 78 | Show Axes..... | 86 |
| Displaying as % value or bar graph..... | 74 | Show call number..... | 159 |
| Explorer panel..... | 55 | Show call stack..... | 84, 160, 161 |
| Exporting..... | 78 | Show error if source not found..... | 160 |
| Getting results during profiling..... | 144 | Show grid lines..... | 83, 84, 87 |
| Grouping..... | 79 | Report panel..... | 87 |
| in Report panel..... | 36 | Show group Summary..... | 87 |
| Managing..... | 38 | Show grouping panel..... | 79 |
| Merging..... | 77 | Show indicator..... | 86 |
| Searching..... | 81 | Show instruction note as hint..... | 83 |
| Sorting..... | 81 | Show instruction note as lines..... | 83 |
| Transferring..... | 38 | Show methods only under class..... | 88 |
| Results tab..... | 36 | Show non-hit functions..... | 157 |
| Run Parameters dialog..... | 155 | Show paths..... | 87 |
| Running the profiler..... | 35 | Show pointing-hand cursor..... | 82 |
| Run-time packages..... | 22, 129, 130, 133 | Show results for all profilers..... | 85 |
| Compiler settings in C++Builder..... | 22 | Show source line summary..... | 84 |
| Compiler settings in Delphi..... | 22 | Show summary..... | 83, 87 |
| Memory and API Resource Check..... | 130, 132 | Show the function body in Details..... | 45 |
| Run-time type information..... | 123 | Single-click details..... | 87 |
| S | | Smallest percent identified..... | 157, 158 |
| Sampling interval (clock ticks)..... | 159 | Sort functions in Call Graph by..... | 157 |
| Sampling profilers..... | 20, 111 | Sorting results..... | 81 |
| Description..... | 110 | Source code..... | |
| Options..... | 159 | File in the Editor panel..... | 52 |
| Save Desktop..... | 18 | SQL queries -- Profiling..... | 124 |
| Save To File..... | 58 | Stab debug info format..... | 26 |
| Saving results to a file..... | 78 | Stack..... | 161 |
| Search Directories..... | 120 | Stack frames..... | 107, 161 |
| | | Static Analysis..... | 20, 89 |

| | |
|-------------------------------------|-----|
| Description..... | 89 |
| Stop shortcut..... | 86 |
| Stored procedures -- Profiling..... | 124 |
| Sum of All Series..... | 87 |
| Sum of All Visible Series..... | 87 |
| Support..... | 13 |
| Supported Compilers..... | 12 |
| System files..... | 29 |
| System functions..... | 29 |
| System Requirements..... | 11 |

T

| | |
|---|------------|
| Tab size in spaces..... | 84 |
| Tab-delimited text..... | 78 |
| Exporting results to..... | 78 |
| TD32 debug info..... | 22 |
| Text file..... | 78 |
| Exporting results to..... | 78 |
| Threads..... | 36, 152 |
| Assigning names to..... | 152 |
| Result display..... | 36 |
| Time..... | 53 |
| Column in the Event View panel..... | 53 |
| Time from application start..... | 53 |
| Tips..... | 145 |
| Tolerance..... | 158 |
| Toolbars..... | 16 |
| Top 10..... | 40 |
| Top 10 %..... | 39 |
| Top 10 % (Net Time)..... | 39 |
| Top 10 % (Time w Children)..... | 39 |
| Top 10 Executed Procedures..... | 39, 40 |
| Top 10 Procedures..... | 39, 40 |
| Top 10 Procedures (Net Time)..... | 39 |
| Top 10 Procedures (Time w. Children)..... | 39 |
| Top 20..... | 39, 40 |
| Top 5 %..... | 39 |
| Trace with parameters..... | 159 |
| Tracing profiler..... | 103 |
| Tracing resources and memory..... | 125 |
| Track recursion depth..... | 158 |
| Triggers..... | 33, 34, 35 |
| Options..... | 35 |
| Setting up..... | 34 |
| Using..... | 33 |

U

| | |
|---|-----|
| Uncovered --view..... | 39 |
| Uninstalling AQtime..... | 13 |
| Unreleased Classes Only..... | 39 |
| Unused VCL Units Profiler..... | 118 |
| Description..... | 118 |
| Options..... | 160 |
| Principles of operation..... | 120 |
| Update procedure names to Delphi 5..... | 85 |

| | |
|------------------------------------|-----|
| Upper case..... | 83 |
| Use debug libraries..... | 129 |
| Used DLLs..... | 66 |
| User Interface - Overview..... | 16 |
| Using AQtime as an OLE server..... | 142 |
| Using triggers..... | 33 |

V

| | |
|--|---------------------------|
| VCL applications | |
| Profiling with Memory and API Resource Check..... | 129 |
| VCL Class Profiler..... | 107 |
| Description..... | 107 |
| Details panel..... | 47 |
| Options..... | 159 |
| VCL Profilers..... | 21, 22, 23, 107, 108, 109 |
| Options..... | 159 |
| Preparing applications for..... | 22 |
| VCL Reference Count Profiler..... | 107 |
| Description..... | 107 |
| Details panel..... | 48 |
| Options..... | 159 |
| Views..... | 82 |
| Views implementation..... | 38 |
| Visual Basic..... | 12, 25 |
| Visual C++..... | 12, 23, 123 |
| Analyzing applications using Memory and API Resource Check..... | 129 |
| Compiler settings..... | 23 |
| Compiler Settings - ATL RefCount profiler..... | 123 |
| Preparing a project..... | 23 |

W

| | |
|-------------------------------------|---------------|
| Warning level..... | 157, 158, 159 |
| Coverage Profilers..... | 157 |
| Function HitCount..... | 158 |
| Function Profiler..... | 157 |
| Function Trace profiler..... | 159 |
| Line HitCount profiler..... | 158 |
| What to profile..... | 28 |
| Controlling..... | 29 |
| WinAPI Database Editor..... | 162 |
| Window and Process Recognition..... | 61 |
| Windows Script Components..... | 60 |
| Work Count trigger option..... | 34 |

X

| | |
|---------------------------|----|
| XLS..... | 78 |
| Exporting results to..... | 78 |
| XML..... | 78 |
| Exporting results to..... | 78 |

Z

| | |
|---------------|----|
| Zoom In..... | 43 |
| Zoom Out..... | 43 |