ProDelphi User Guide

(Release 41.x / Versions for 32-bit-applications)

The Profiler for Delphi 5, 6, 7, 2005 .. 2010, XE, XE2 .. XE8, 10 .. 10.4, 11, 12 (for Pentium and compatible CPU's)

Profiling

The purpose of ProDelphi is to find out which parts of a program consume the most CPU-time. Because Borland gave up its Turbo profiler, a new tool had to be created. ProDelphi with its comfortable viewer, browser, history and programmer's API meanwhile is more than the legendary Turbo Profiler. The viewer with its sorted results enables the user to find the bottlenecks of his program very fast. The history function shows the user, if a preceding optimization was successful or not. ProDelphi's outstanding granularity makes it possible even to optimize time critical procedures. The built-in calibration routine adapts the measurement routines to the used processor and memory speed and guaranties results that do not include measurement overhead.

Starting with release 8.0 the dynamic activation becomes very easy to use. Instead of inserting API-Calls (which in all former releases was possible and still is), now by a handy dialogue the activation starting methods can be selected. The in all former versions built-in functionality of the measurement-DLL now is usable without recompilation.

With release 13.0 a caller / called graph came that lets navigating through the measurement results (in connection with opening a source file in the IDE editor just by mouse click) be a pure pleasure. The starting point list makes it easy to identify paths to instrument.

Post Mortem Review

Another reason to develop ProDelphi was the need for a tool that shows the call stack of a testee in case of an abortion or exception. ProDelphi realizes that function without the testee running under the IDE.

Differences between the freeware version and the professional version:

With the freeware version up to 20 procedures can be measured or tracked, in the professional version up to 64000. In the professional version additionally assembler procedures can be measured and tracked, minimum and maximum runtimes can be displayed in the viewer and the user part of the library path can be instrumented. The professional version is available with Unicode support or with Ansi Code support only, the freeware version has only Ansi code support.

Date: 11/7/2023

Contents of this description

BEFORE using ProDelphi practically, please read Chapter A.3 and A.16!!!

A. Profiling

A.1. Introduction

The source code of the program to be optimized is instrumented with calls to a time measuring unit. The insertions are made at the beginning and the end of a procedure or function.

Any time a procedure / function / method (in the following text method is used) is called, the start time of the method is memorized. At the end of the method the elapsed time is calculated. **When the program ends**, between three and five files are created that contain the runtime information for each method:

The **first** file (programname.txt) contains the elapsed times in CPU-Cycles. The format is ASCII, separated by semicolon (;) and can be used either for **Data Base import** or for the **built-in viewer of ProDelphi**. The format is described at the end of this description.

The **second** file (programname.tx2) contains additional information like a headline and how often measurements have been appended to the first file. It is relevant in connection with the online operation window or the programmers API.

The **third** file (programname.tx3) contains information used for opening a file in the editor and positioning the editor cursor to the measured method.

The **fourth** file (programname.nev) contains the names of all methods which have never been called when measuring the runtime of your program. It is used be the viewer, it is displayed as a hierarchical tree when you press the button named ‘Not called methods’. This button is not enabled if all methods have been called or if you display the measurement results of a former version of ProDelphi.

The **fifth** file is also optional and only created, if the automatic switching off is activated (Chapter A.5).

A.2. Basic profiling

Using ProDelphi is quite simple. It has been used in a project with a large program, which now already contains more than 370 000 lines of code written by 12 programmers. After more than two years of developing the program has been optimized with the help of ProDelphi. The programs runtime could be decreased by 50 %.

Use the Setup-program to install ProDelphi. The setup program can only then work correctly when Delphi or a previous version of ProDelphi is not started. If you update from a previous version of ProDelphi you need to uninstall the old version first. For that use the setup program stored in its installation directory.

After installation, try to compile your program to create the Delphi project file. **If no project-file exists, all files have to be in the same directory (*.PAS, *.INC, *.DPR, *.EXE and *.DLL).**

If you want to measure methods in a program and in DLL's simultaneously only:

Program and DLL's must have exactly the same units source path, their DPR-files need to be in the same directory, also the EXE-files and the DLL-file have to be in the same directory. In that case, compile both: program and DLL's. All files to be instrumented must be stored in directories of the units search path except those that have an explicit path in the USES-statement in the DPR-files of program or DLL. For profiling program and DLL simultaneously, the button 'Program + DLL's / Multiple DLL's' must be checked (see also chapter A.9).

If your files to instrument are very large, and you have opened them in the IDE, you should close them. It was reported, that Delphi does not properly actualize its window content if a file is very large and the file is changed on disk from outside IDE.

If no compilation errors occur, you may instrument your program (*and/or DLL*).

Don't use the original units for instrumenting, maybe ProDelphi still contains bugs. Just make a security copy of the program to be measured, e.g. by zipping all PAS-, DPR and INC-files.

For measuring the runtime of your program perform the following steps:

- Define the Compiler-Symbol PROFILE (project/options/conditional defines).

- Deactivating the Optimization option is strongly recommended (see A.7.1)

- Optionally deactivate all runtime checks.

- Use the Delphi 'Save All' command. This assures that the project file is stored.

- Start ProDelphi from the Delphi tools menu, from the Windows Start menu or somehow else.

- With ProDelphi select the project to profile (if it is not automatically selected).

- For the first example only those options that are checked in the following example are recommended.

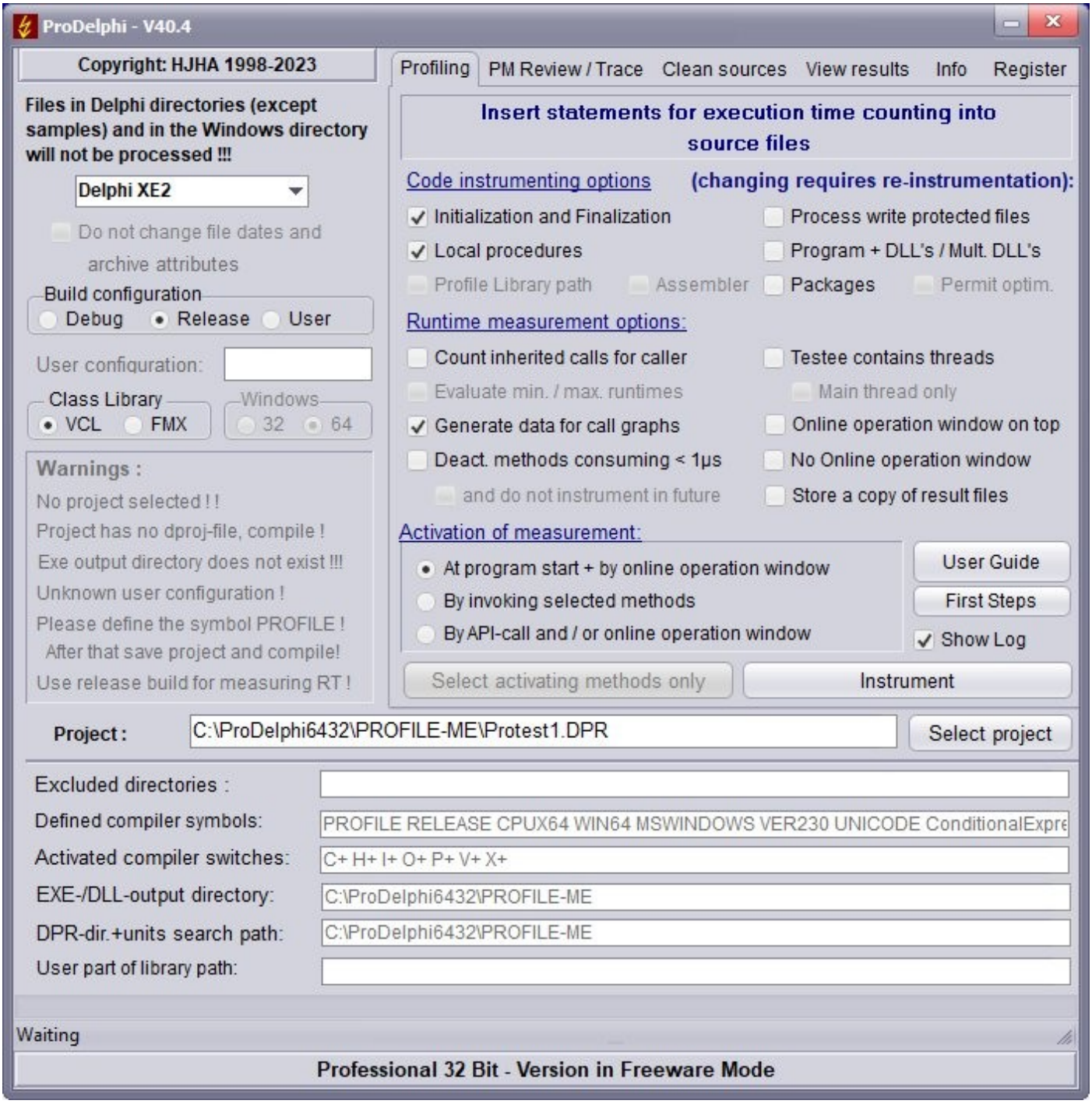
Following options are available in professional mode only:
- Measuring units in the library path
- Assembler procedures
- Evaluating minimum and maximum runtimes (in the freeware mode only the much more important average runtimes are available).
- Do not change file dates. Checking this option results in changing the file date/time by 2 seconds only when instrumenting, just enough to make Delphi realize that a file has changed. The file date/time is set back to the value that was set when doing the first instrumentation by cleaning the sources.

By the way: The most important buttons are 'First Steps' and 'User guide' !

- Select the kind of activation for measurement you like (in this example by start).

- Uncheck the button 'Show Log' (professional version only) if you don't want to see the profiling log.

- Click the Instrument-button. After a very short time all units are instrumented. The instrumented files are listed in a log window.



- If you want to measure methods in DLL's instrument the necessary DLL's too.

- Recompile the program (*or DLL's*).

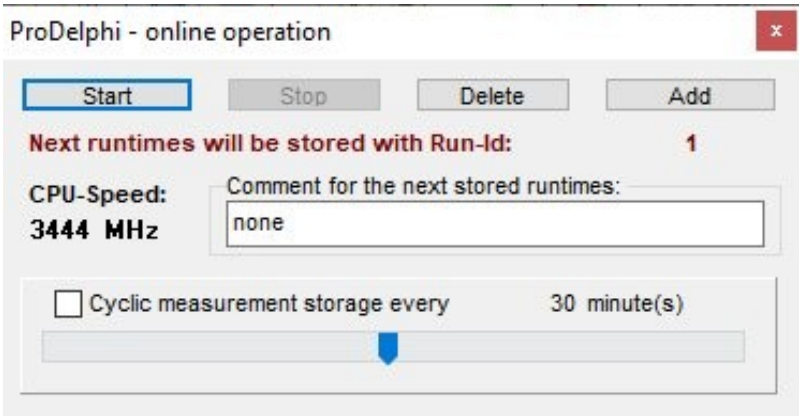
To allow simultaneous measuring of DLL's and programs, all files in the units search path are instrumented !!! (unless they are write protected !!!). The unit search path must be exactly the same for program an DLL, both DPR-files have to be in the same directory !!!

Files in and below the Delphi LIB and SOURCE directories path will not be instrumented.

After that, start the program and let it do its job.

A small window appears that allows you to start and stop the time measurement:

The Online operation window, see next page:



Depending on the profiling options, the button 'Start' is enabled (No Autostart option) or not (with autostart option). With autostart option, the measurement starts with the start of the testee. Without the autostart option, you have to press the start button in the online operation window when you want to start the measurement, define activating methods or insert calls into your sources for activation or deactivation. See chapter A.5.2 for the complete description. After the program has ended, you can

view the results of the measurement with the built-in viewer of ProDelphi,

For the Built-in viewer, just start ProDelphi again, go to the 'View results' page. If the name of your project is not automatically displayed, select it. Then click the 'Load and view' view-button.

In principle, this is all that has to be done. If you want to let the program run without time measurement, simply delete the compiler symbol PROFILE in the Delphi options and make a complete compilation.

A.2.1. Files created by ProDelphi or the measured program

ProDelphi creates the file 'proflst.asc', it contains information about the methods to be measured for measuring or traced for post-mortem review. The file profile.ini contains options for the time measurement and the last screen coordinates of the online operation window. The viewer can create a file named '*.hst' if you use the history function (see A4.1)

Your compiled program creates a file with the name 'progname.txt' that contains the data in the ASCII-semicolon-delimited format for database import and viewer and 'progname.tx2' for the headlines for the different intermediate results (for the built-in viewer). A file 'programname.tx3' is stored for the interface to the Delphi-IDE. The file 'progname.swo' with the list of methods that have to be deactivated for time measurement at next program start is stored optionally. Also, a file with the name 'progname.nev' is created in which the names of the uncalled methods are stored. The viewer also uses this file.

Your compiled program creates a file named 'progname.pmr' in case you have selected post-mortem review and an exception occurred and was trapped. It contains the call stack.

All files are stored in the output directory for the *.exe (*.dll) file.
To allow simultaneous measuring of DLL's and programs, all files in the units search path (except the Delphi LIB and SOURCE directories and below them) are instrumented if they are not write protected !!! Search path for program and DLL need to be identical in that case.

A.2.2. Checking the results with the Built-in Viewer

The most comfortable way to view the run times of your methods, is to use the built-in viewer. Just click 'View results' and 'Load and view'.

The results are stored into the result file either at the end of the tested program or any time the Store-button of the online-operation window is clicked.

You can choose if you want to view the results in µs, ms ... or in CPU-Cycles.

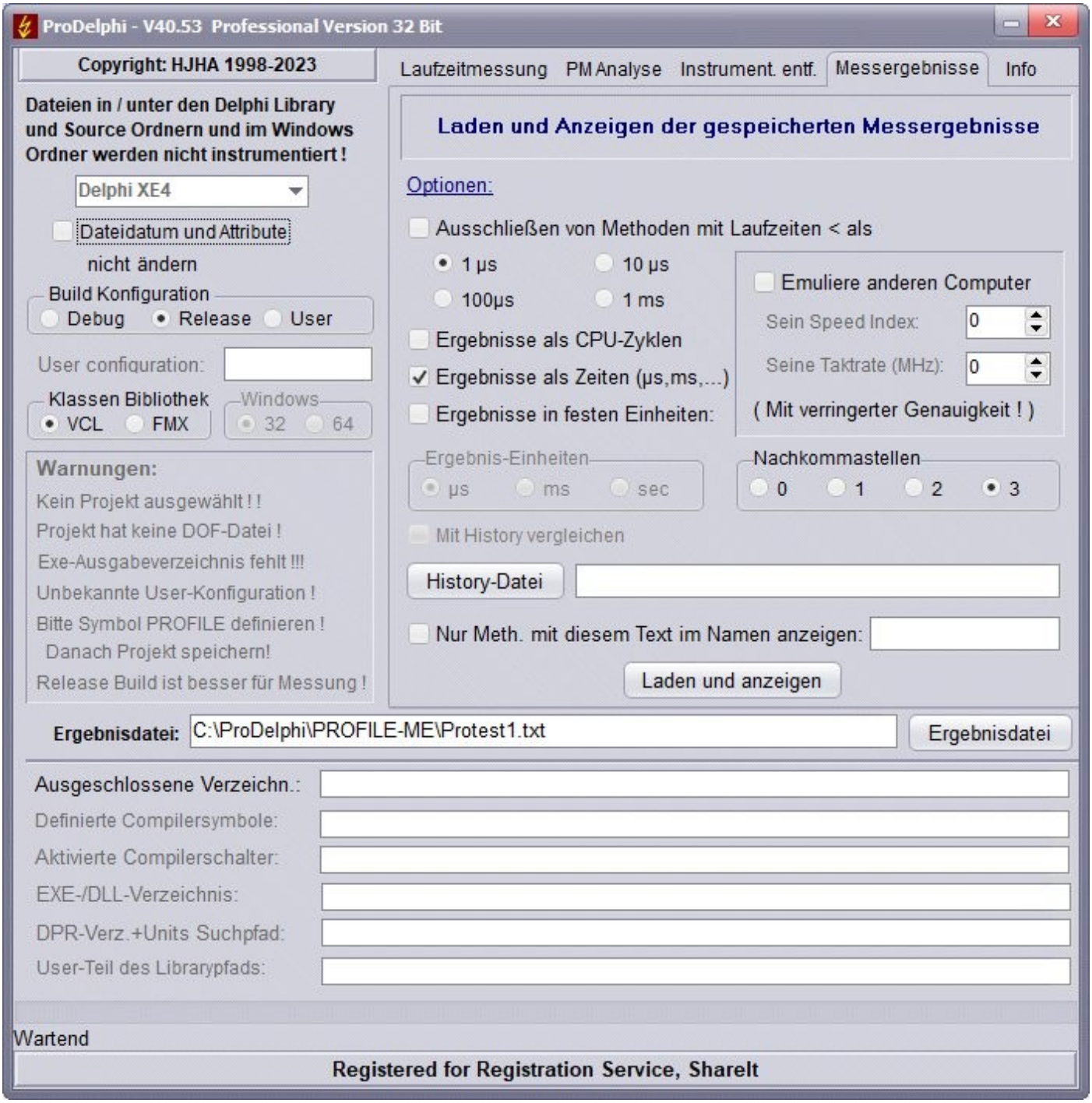
You can exclude methods with less than 1µs, 10µs,100µs or 1ms.

Also, you can emulate (re-calculate) the measurements for a faster or slower PC. No need to install the IDE on that

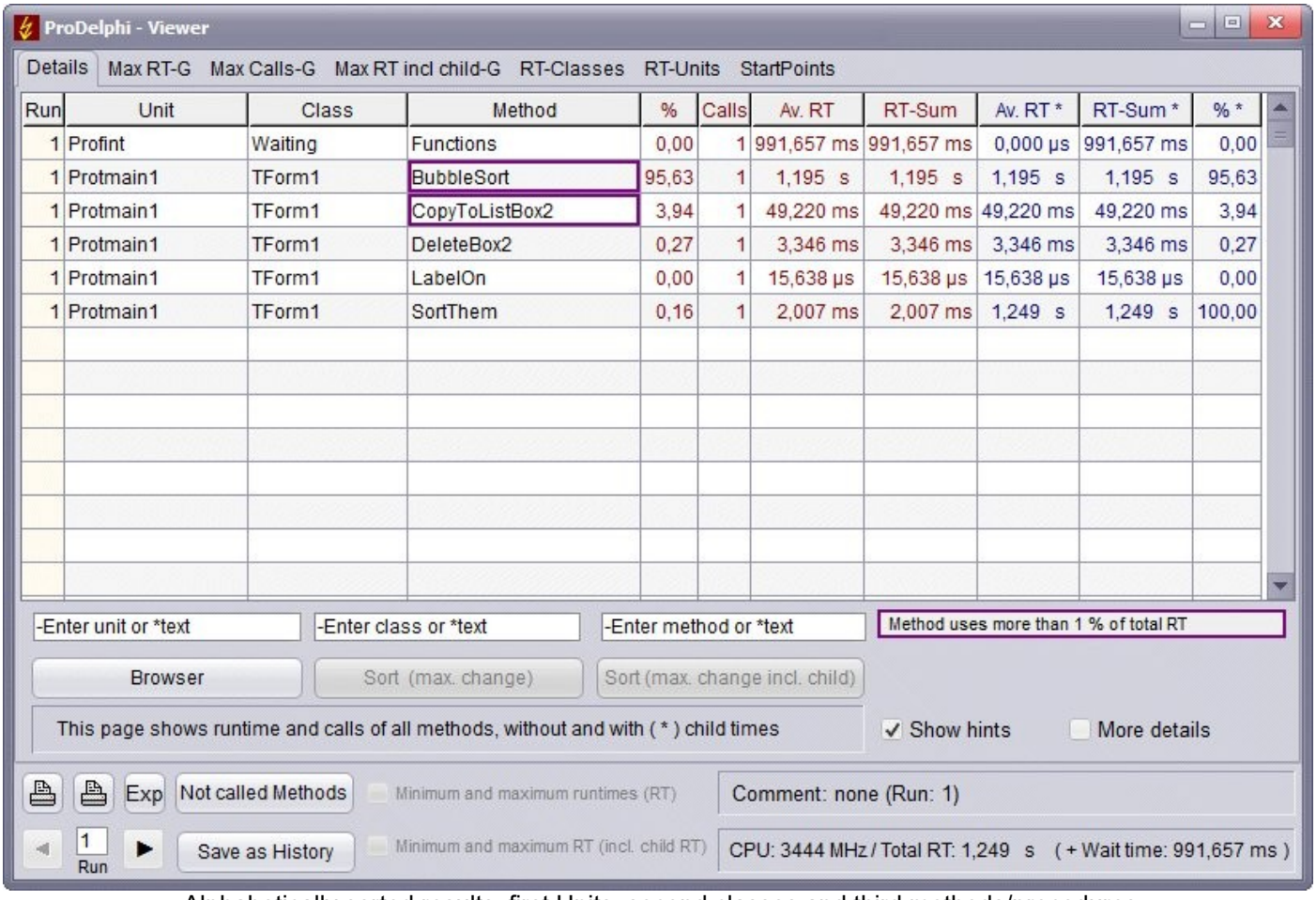
PC, just enter two constants in an edit field and let ProDelphi tell you how fast or how slow your program would

perform on that PC (see chapter A.2.3).

On clicking ‘Load and View’, a grid is shown, which gives you the results of the measurement. You can scroll through the results or e.g. search a specific unit, class or method.



See next page please.



Alphabetically sorted results, first Units, second classes and third methods/procedures

Explanation of this window:

CPU: nnn MHZ	giving the CPU - speed
Total RT: ttt	giving the runtime of all measured methods (alternatively in CPU-cycles)
Wait time:	appears when the program waited without doing anyting (eg Sleep- or MessageBox function)
Comment: ccccc	Text set as comment in the online operation window for intermediate results, 'At finishing application' when the results were automatically stored when the testee ended or date and time when the online operation window cyclically stored results.

Sorting the table:

The displayed table can be sorted after different criteria by clicking the tabs, just try it! Two extra buttons are supplied to sort by comparing the measured results with a stored history. The columns are sorted in a way that those methods that changed the most are displayed on top (see also history). The displayed order can be reversed by clicking a second time.

Navigating through the results:

Navigating through the results can be done by scrolling, using the browser or by searching for unit, class and object. The search is started by typing in the search text. With the F3-key the search can be repeated, also positioning by paging up and down is possible.

Navigating from the viewer to the source code:

Right mouse button click in a line of the viewer's grid causes Delphi to open the file in the editor and positioning the cursor on the start of the method. For using this function, Delphi has to be started.

The Print - buttons:

They print the actually displayed table or table + graphic. The table is automatically adjusted so that it fits on the paper. Using the first button, everything is printed in black, only if necessary, colour is used (colour save mode). Using the second button, everything is printed as displayed on the screen (full colour mode).

The Exp - button:

The content of the actual view is exported to a CSV file. The values are separated by ';'. The exported data then can be read by Excel, LibreOffice, or OpenOffice.

The Minimum/Maximum checkboxes (Professional version only):

Checking these options, minimum and maximum runtimes are displayed if they were collected while measuring. If the checkbox 'Evaluate min. / max. runtimes' in the profilers main window was disabled, no minimum and maximum values were evaluated and stored (see A.7.2)

The History - button: see chapter A.4.1.

Meaning of **Run**:

Any time the program stores data into the result file, it puts a leading number before the measured times: the number of the measurement. With the ◄ (Previous)- or ► (Next)- button, you can switch between different measurements. Also, it is possible to enter the run directly in the edit field between the ◄ - button and the ► - button.

At the next run of the program, the counting starts at 1 again.

Meaning of the columns with the RED text:

%	Percentage of the total runtime the method took without their child methods
Calls	How often the method was called
AV. RT	Average runtime of the method in CPU-cycles or in µs, ms, sec or hour units (in the professional version also minimum and maximum runtimes can be displayed)
RT-sumRT * Calls	

Meaning of the columns with the BLUE text:

Av. RT	Average runtime of the method inclusive its child methods in CPU-cycles or in µs, ms, ... (in the professional mode also minimum and maximum runtimes can be displayed)
RT-sumRT * Calls	
%	Percentage of the total runtime the method took inclusive her child methods.

Meaning of the ◄ -Button and the ►-Button:

If your program has stored intermediate results into the result file (by using the ProDelphi-API or by Online operation) you can page back or forward in the result file.

Meaning of **'Comment':**

It is the headline that was inserted when the measurement was stored. In the example you see the default.

The other available pages show:

The 12 sorted methods that consumed the most of the runtime (**exclusive** child methods) given in a text- and a graphical representation.

The 12 sorted methods that were called most often displayed in a text- and a graphical representation

The 12 sorted methods that consumed the most of the runtime (**inclusive** child methods) given in a text- and a graphic representation.

The 12 sorted classes that consumed the most runtime.

The 12 sorted units that consumed the most runtime.

Meaning of runtimes inside a red frame:

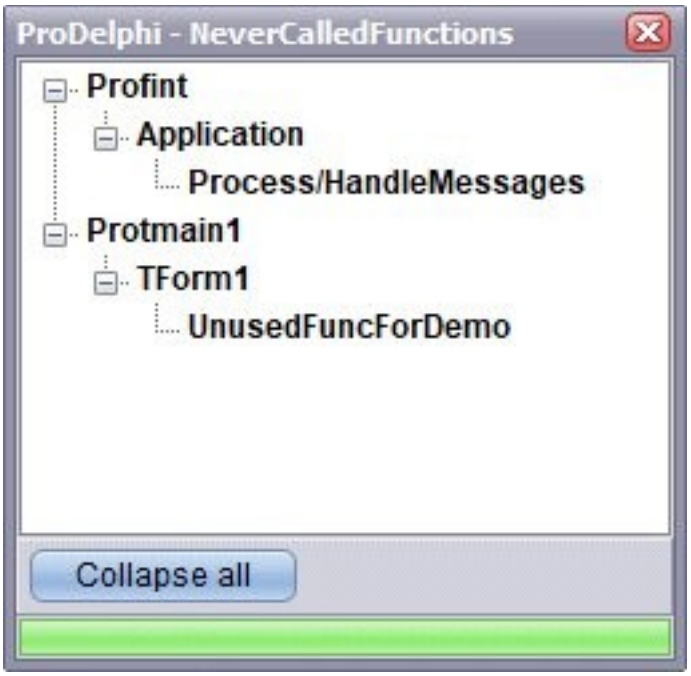
The runtime is greater than the time stored in the history file. The frame only then appears, if the change is greater than 1% of the total runtime of the application.

Meaning of runtimes inside a green frame:

The runtime is less than the time stored in the history file. The frame only then appears, if the change is greater than 1% of the total runtime of the application.

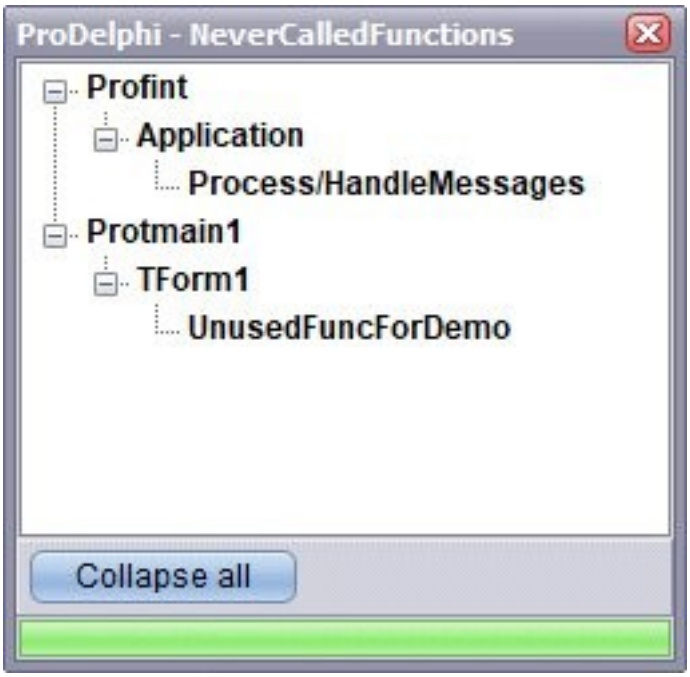
The Not called Methods - button:

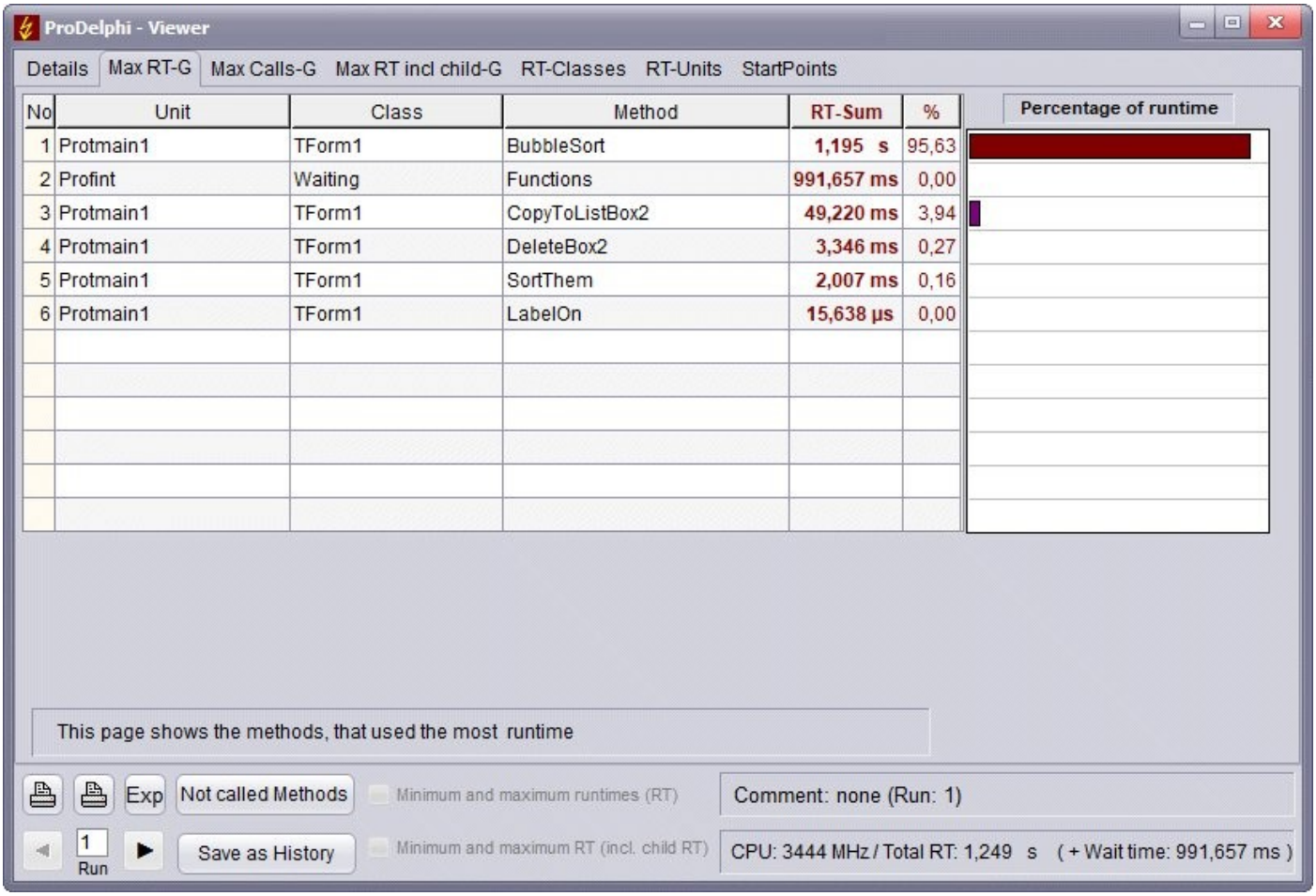
At the end of runtime, the testee creates a file with the names of all uncalled methods. Using this button, these methods are displayed in hierarchical order: Unit - Class – Method.



The Browser - button:

It opens a small browser window (similar to explorer) that shows units, classes, and methods in a brachial order. It can be used to quickly find the measuring results for a certain method.





Example of: Maximum run time-consuming methods (graphical)

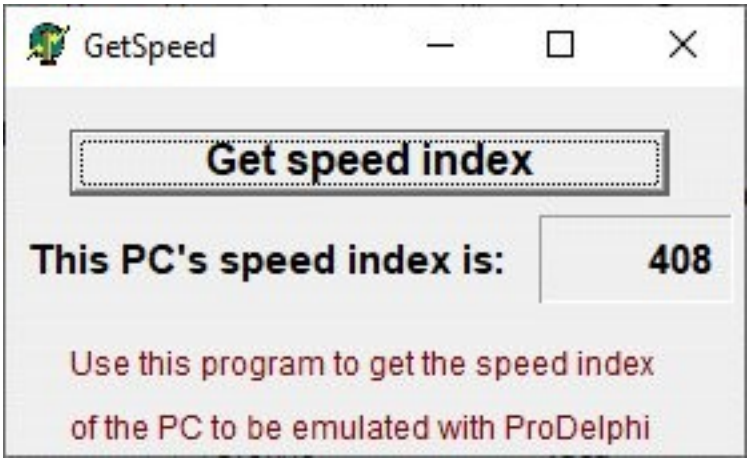
A.2.3. Emulation of a faster or slower PC

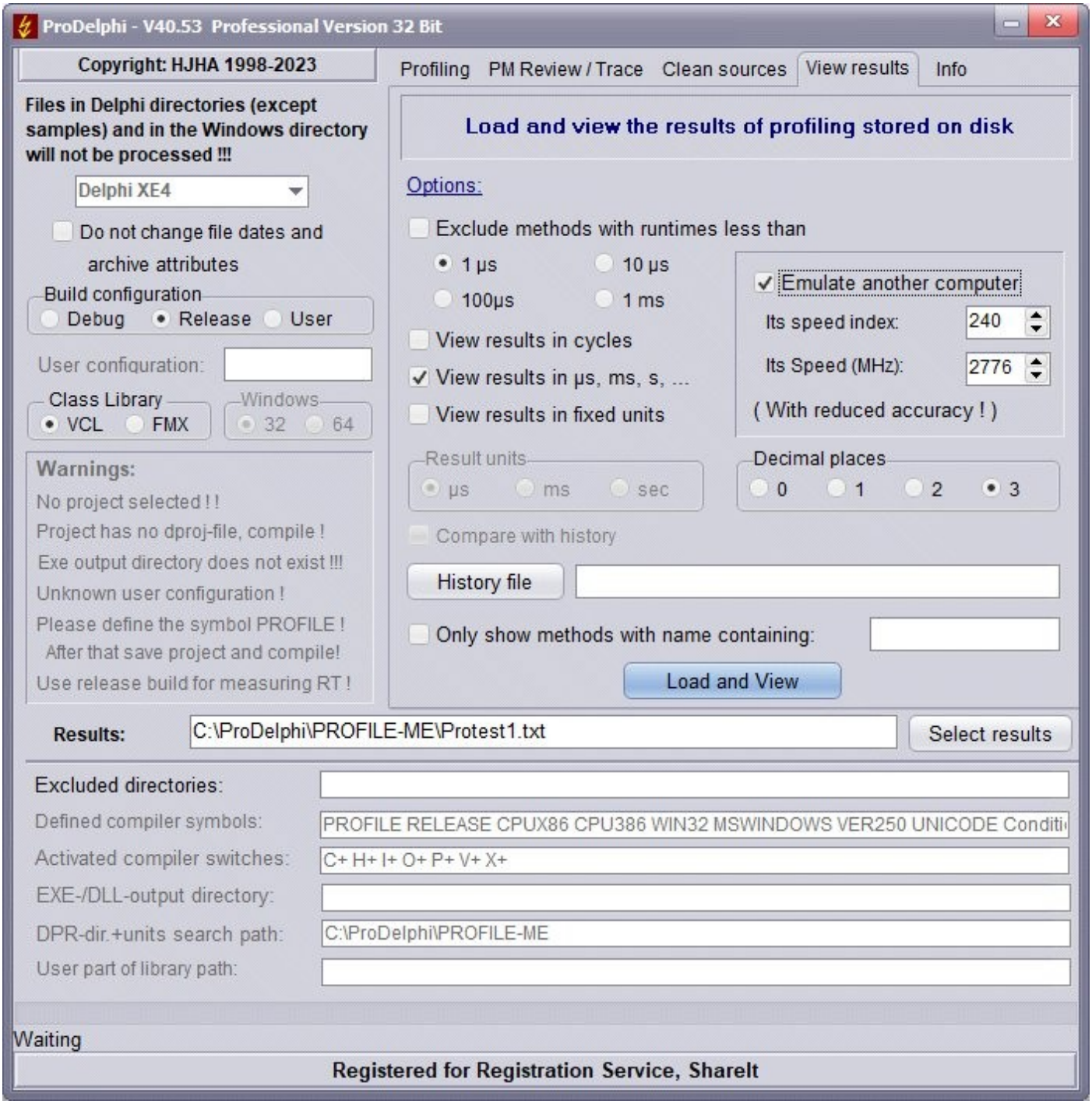
If you want to know, how fast (or slow) your program would perform on another PC, just use the program Getspeed.exe to get the other PC's speed index, enter it in ProDelphi, enter the speed in MHz of the other computer and start the viewer. Automatically, all measurements are recalculated for the other PC. ***Certainly, the results are not as accurate as if measured on the original PC.***

Limitation of use: If in your program you have a method that executes for a fixed time (e.g. for 1 sec), the emulation result for that method is wrong!

(The speed index and MHz'es of the PC on which ProDelphi is executed, is calculated automatically, so do not delete Getspeed.exe after installing ProDelphi, it is used for this purpose also on the PC on which ProDelphi is installed).

See next page for example, please.

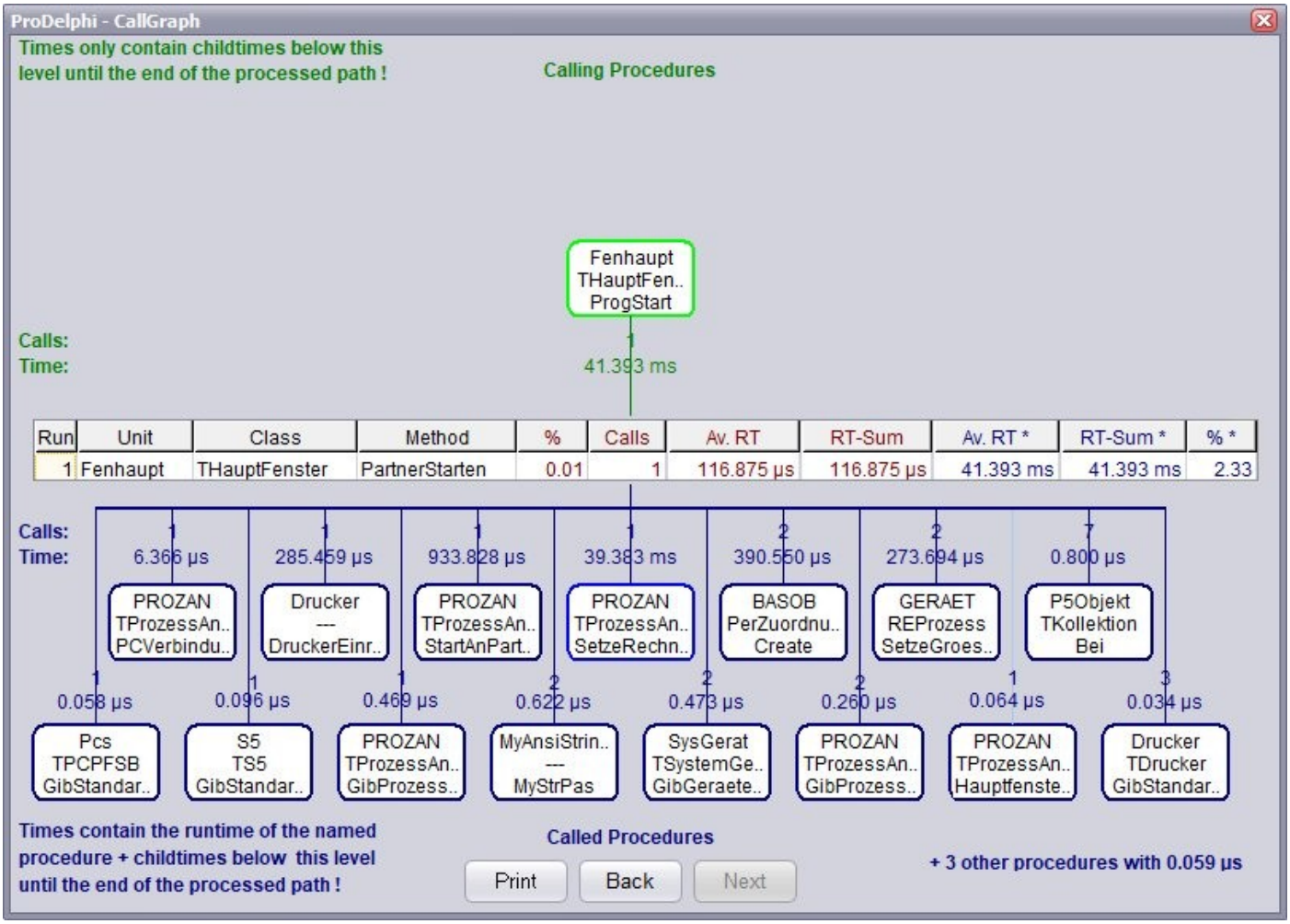

GetSpeed
Get speed index
This PC's speed index is:
408
Use this program to get the speed index
of the PC to be emulated with ProDelphi


ProDelphi - V40.53 Professional Version 32 Bit
Copyright: HJHA 1998-2023
Profiling
PM Review / Trace
Clean sources
View results
Info
Files in Delphi directories (except samples) and in the Windows directory will not be processed !!!
Delphi XE4
Do not change file dates and archive attributes
Build configuration
Debug
Release
User
User configuration:
Class Library
VCL
FMX
Windows
32
64
Warnings:
No project selected ! !
Project has no dproj-file, compile !
Exe output directory does not exist !!!
Unknown user configuration !
Please define the symbol PROFILE !
After that save project and compile!
Use release build for measuring RT !
Load and view the results of profiling stored on disk
Options:
Exclude methods with runtimes less than
1 µs
10 µs
100µs
1 ms
View results in cycles
View results in µs, ms, s, ...
View results in fixed units
Emulate another computer
Its speed index:
240
Its Speed (MHz):
2776
(With reduced accuracy !)
Result units
µs
ms
sec
Decimal places
0
1
2
3
Compare with history
History file
Only show methods with name containing:
Load and View
Results:
C:\ProDelphi\PROFILE-ME\Protest1.txt
Select results
Excluded directories:
Defined compiler symbols:
PROFILE RELEASE CPUX86 CPU386 WIN32 MSWINDOWS VER250 UNICODE Conditi
Activated compiler switches:
C+ H+ I+ O+ P+ V+ X+
EXE-/DLL-output directory:
DPR-dir.+units search path:
C:\ProDelphi\PROFILE-ME
User part of library path:
Waiting
Registered for Registration Service, ShareIt

A.2.4. Using the caller / called graph (call graph)

If the call graph data has been collected when measuring the runtime (checkbox in main window) the call graph can be displayed. When clicking with the left mouse button on a measured result in the viewer grid, a new form opens which displays the runtimes of the selected method in a grid in the middle of the form. Above that, up to 15 methods are displayed that have called this method. If more methods called the selected method, this is displayed at the top of the form. Always those methods are displayed, that consumed the most runtime. For each method, the number of calls for the selected method and the runtime inclusive all child methods is displayed.

Below the method shown in the grid, up to 15 methods called by the selected method are displayed. Again here those methods that consumed the most runtime are displayed with the number of calls and the runtime consumed inclusive child times **(Screen shot is not from the example program in this manual)**:



The ‘Critical path’ window displays the call sequence with the highest execution time beginning with the method displayed in the grid.

Left-clicking on the symbol for a called or calling method makes this method appear in the grid with its complete measurement results.

Left-clicking on the method actualizes the ‘Critical Path’ window.

Right-clicking on the method in the grid opens the concerned unit in the editor. The editor displays the method on top of the window.

A red ‘**R**’ on the left side of the grid in the middle of the window mean that the shown method was called recursively. Also, a red method name in one of the symbols has this meaning.

A.3. Getting exact results

If you measure program runtimes a few times, you will see that the measurement results differ from measurement to measurement without that you have changed your sources. Two kind of results will often differ: the runtime of a method and the percentage of their runtime of the complete program. The reasons are :

- there are events that disturb the measurement, e.g. programs running in the background.
- you measure methods which are activated by Windows more or less often,
- you measure operations which are started by an event a different number of times each measurement,
- you measure methods which perform disk transfer, the data can be transferred to disk or to disk cache.

Every profiler has these problems. Because of the highest possible granularity of ProDelphi (1 CPU-cycle), you see these differences.

To get comparable measurements you need to take care, that the influence of disturbances is kept low. Here are some hints:

A.3.1. Common causes of disturbing influences outside your program

Some disturbers everybody might be aware of:

- activated screen saver,
- Windows power management,
- background schedulers,
- online virus protection,
- automatic recognition of CD changing,
- temporary windows swap file causes memory transfers of different duration,
- dynamic Windows disk cache size causes a different amount of memory for each measurement.

These disturbing influences are easy to eliminate.

A.3.2. Common causes of disturbing influences inside your program

Some disturbances you might have inside your measured program itself, these occur when you measure everything, e.g. by using the autostart function of ProDelphi:

- defining a Default Handler procedure (is called for nearly every message your program receives),
- defining a procedure to handle mouse moves (called every time you are moving the mouse cursor),
- defining a timer routine.

The three influences are also easy to eliminate. You only need to exclude these procedures from measurement. Another way is not to use the autostart function of ProDelphi but start measurement at the starting point of a certain action. How to exclude methods is described in Chapter A5, how to measure defined actions only is described in chapter A.6 and A.7.3.

A.3.3. Intel SpeedStep Technology / Turbo Boost mode

These features change the CPU-speed dynamically, unfortunately not always at the same time. So no comparable measurements can be performed concerning time units, even when exact the same code under the same conditions is executed. When time units of measurements have to be compared, these features should be deactivated. Some PC's have this possibility in their BIOS. **Comparing measurements with using a history file gives you correct results: the comparison uses the CPU cycles of the measurements!**

A.3.4. Common cause of disturbing influence is the PC's processor cache

The influence of the cache can't be easily excluded. The only way is to produce exactly the same sequence of events two times every measurement and to start measurement with starting the second sequence by the programming API, switch it off at the end of the second sequence and store the measured data to disk (also by the ProDelphi API). This guarantees that as much code as possible is stored in the cache and that every measurement the same code and data is in the cache. Only if your program does exactly the same every measurement, you can compare the results and find out (e.g. by the history function of ProDelphi), if an optimization has decreased the runtime or not.

A.3.5. Measuring on mobile computers

Mobile computers have one problem: They change their CPU-speed dynamically. If a mobile computer is connected with AC power it normally uses the full CPU speed, if working with battery power, the CPU speed changes dynamically.
This does not directly affect the measurement: ProDelphi measures CPU cycles. If we look at the CPU - cycles displayed in the viewer, the measurement is correct. If times are displayed, it could be that too long or too short times are displayed. It depends on the CPU speed that was set when the CPU speed was measured. Different processors use different algorithms to change the speed. The only way to get correct results is to switch off the power safe mode.

A.3.6. Summary

If you eliminate the disturbances mentioned in A.3.1 / A.3.2 and A.3.3 and measure defined actions, you will see the differences between two measurements are very low, most times only a few CPU-cycles. Larger differences appear only in measuring methods with disk transfers. A good trick is, to use the second measurement for comparison with later optimizations, specially when the disk transfer is a reading transfer. The first run of the program will get the most data into disk cache, the second measurement reads the data from cache.

A.4. Interactive optimization

Interactive optimization means that you optimize something, check if it has brought you significant decreasing of runtime or not, make the next step of optimization and so on.

Important is, which method is worth to be optimized: A method, that uses 10 % runtime must be optimized by 50 % to decrease the total program runtime by 5 % !!!

There are different ways of comparing the measurement results:

- to use the viewer and print the measurement results or
- to use ProDelphi's history function.

A.4.1. The history function

The history function of the viewer enables you to compare your measurement results with a preceding run. So you can see, if an optimization has brought an increasing or a decreasing of runtimes.

Having made a measurement, you can store the results being displayed in the viewers table on disk. You can store multiple histories on disk for different kind of measurement.

Once you have stored results as history, you can select one of the history files to be compared with the results of the last measurement. Before loading the results into the viewer, select the history to compare with and check the button ‘Compare with history’. The viewer will colour the cells of the viewer's table, by this you have a quick overview about all changes of runtime: Red means method got slower, green means method got faster and white mean that no essential change occurred.

To get the cell coloured, the method's change of runtime must be essential. Essential means, it must have changed so much, that it influenced the program's runtime by 1 % or more.

To display the runtime of a method from the stored history, press the Ctrl key and left-click the concerned method.

If you succeed in excluding disturbing effects as mentioned before, you can use the history very well. E.g., I had to optimize the processing of measured values. I simply didn't use the auto start function and used the API to switch measurement on and off. I switched it on after processing 10 measurement values (all called methods were in the cache then), measured processing of 100 values, stopped measurement and stored the data on disk. To be sure that no disturbing actions occur any more, I repeated this and compared the measurement results with the history function. When there were nearly no differences between two measurements, I started to optimize and always used the history to compare, if my optimization was successful or not.

A.4.2. Practical use of the history function

- Make a measurement for the defined action you want to optimize.
- Load the results into the viewer.
- Click on the history button to store these results into the history file.
- Optimize a method that is worth to be optimized.
- Repeat your measurement.
- Load the new data into memory.
  If you made the function significantly faster, the optimized method should be coloured green now.
  If your method is slower now, it is coloured red.
  If there is no significant difference, it is coloured white.
- Select a cell in that line, where your changed method is displayed.
- A small window pops up. It shows the average runtime of a method stored in the history file. If '---' is displayed, the method is not present in the history file.

A.5. Measuring parts of the program

A.5.1. Exclusion of parts of the program

All Windows programs are message driven. So, if you define a function that, for instance, handles mouse moves, ProDelphi will give you a very big percentage of runtime for this method because it will be activated any time you move the mouse over a window of your program. But you might not be interested in this method.

What I described above, is the default setting of ProDelphi: all methods are measured, the measurement starts with the start of the program (if option 'Activation of measurement / At program start' is checked).

For normal, you would like to measure only certain actions of the program and might want to exclude functions which cannot be optimized (e.g. because they are very simple).

There are different ways of excluding parts of the program:

1. Files in and below the Delphi LIB- and SOURCE- directories are always excluded.

2. Methods which have the first 'BEGIN' statement and the last 'END' statement in the same line, are NOT measured. **It's not a bug !!! It's a feature !!!**

3. Exclusion of directories (see also A.13.5)

   Enter the directories in the field ‘Excluded directories’ of the ProDelphi main window.

4. Exclusion of complete units

   - Enable write protection for the units not to compile
     (unless you don't check 'Process write protected files', they are not instrumented) or
   - insert the following statement before the first line of the unit:

   //PROFILE-NO

5. Exclusion of DLL's but measuring the program

   Just compile the DLL without the compiler definition PROFILE and the program with that definition.

6. Exclusion of the whole program but measuring the DLL's

   Compile the program without the compiler definition PROFILE and the DLL with that definition.

7. Exclusion of functions

Before instrumenting, insert statements before and after the methods that have to be excluded to switch off the instrumentation by ProDelphi:

```
//PROFILE-NO          | This statement is not removed by cleaning
PROCEDURE A;
BEGIN
  :
END;
PROCEDURE B;
BEGIN
  :
END;
//PROFILE-YES         | This statement is not removed by cleaning
```

8. Automatic exclusion

You can exclude methods automatically by checking the option 'Deactivate functions consuming < 1 µs'. Checking this option means that those methods, which are at least called 10 times during the measurement period and consume an average of less than 1 µs will not be measured the next time the program is started. For that purpose, a file is created when the program ends. It contains all the methods which have to be deactivated. When you start your program next time, the file will be read and all named methods are deactivated. It might be that after the next run of your program again, some lines will be appended with methods to be deactivated.

The methods that are not to be measured are stored in the file 'ProgramName.swo'.

Caution, the next run of ProDelphi will delete this file. If you want to make the exclusion permanent, put a //PROFILE-NO statements into your source code.

A.5.2. Dynamic activation of measurement

This is the best way of measuring. Normally, one optimizes a certain function of a program, mostly that which takes too long time. E.g., if a program processes measured values and paints nice pictures and the number of processed values are not enough, one only wants to optimize that part of the program and not the painting.

In this example, it would be nice to switch on the measurement every time a measured value has to be processed and to switch off after. The advantage is, that the number of runtimes seen in the viewer is drastically reduced, the other is, that it is much easier to see, which function should be optimized.

There are three ways for dynamical activation of measurement in ProDelphi (1. and 2. can be used simultaneously):

1. By dialogue

In the main window of ProDelphi under the option ‘Activation of measurement’ select:
‘By entering a selected method’. After instrumenting you can select until 16 methods which should start the measuring. If you have profiled your program before already, you as well can use the button ‘Select activating methods only’. So you easily can change between different activating methods.
Measuring is switched on, when the selected method is entered and stops when the last statement of the method is processed.

2. By inserting special comments into the source code.

Inserting a comment //PROFILE-ACTIVATE into the source code, the next method or function after that comment automatically starts measurement. Also, here you have to check ‘By entering a selected method’ in the main window of ProDelphi. You can optionally select further activating methods, but it is not necessary.

3. By using API-calls.

This method is described in the next chapter . It is the only way versions of ProDelphi earlier than 8.0 could handle this problem. In principle, this way can still be used, but it is not very

comfortable. Using that third method, you always need to insert two calls, one for activation and one for deactivation (see chapter A.6).

A.5.3. Finding points for dynamic activation

If you need to measure an application you have not implemented yourself, it is not so easy to find out where an action starts. Most times there are a lot of events and windows messages, but which are the methods reacting on these events or messages?

To make it a little easier to find out this, all methods that start an action are entered into a list of starting points. Just perform a measurement run winch measures all methods and starts the measurement automatically with the start of the application. After performing the action to measure, end the application, start the profiler and view the results. Under the last tab of the viewer all methods are listed, that were not called by other measured methods, this means that they were started by events like mouse clicks, windows messages etc.. Starting with these functions and in connection with the call graph, it should be easy to find out where to set activation points for an action to measure. Just left-click on the method to display the call graph for a method.

A.5.4. Measuring specified parts of methods

For the case of very large methods, sometimes it might be interesting to know which ***part*** of it consumed the most run time. One way to find out this is to restructure the method into neat parts which are put into local procedure. Another idea would be that ProDelphi would measure each block of a structure and not the whole procedure. The last solution would cost a lot of measurement overhead and would make time critical applications stop working.

For the case that moving code into (local) procedures is too much work, ProDelphi has the feature of defining blocks to measure.

With the insertion of two simple statements, a block to measure can be defined. These statements are constructed as comments and remain in the sources even after cleaning.

Just insert this line before the block to measure:

```
//PROFILE-BEGIN:comment
```

and this one behind it:

```
//PROFILE-END
```

After that re-instrumenting with the option ‘Profile local procedures’ is necessary !!!

Instrumenting the sources after this change causes ProDelphi to insert measurement statements right after the comments. The runtime measured in this so defined block will be found in the viewer by searching for ‘method name(comment)’.

Using this feature is only possible when taking care to insert these statements in a way, that the block structure of the program remains unchanged. E.g. it is not possible to insert the statement into an ELSE-part without BEGIN and END, this would cause compiler errors.

The time measured in this part is not included in the runtime of the method, but it is included in the child time.

Example:

```
PROCEDURE DoSomething;
BEGIN
        part a of instructions using 5 ms
        part b of instructions using 10 ms
        part c of instructions using 3 ms
END;
```

The total runtime displayed by the viewer would be 18 ms (displayed in the line for the procedure DoSomething).

The same example with measuring part-b separately:

```
PROCEDURE DoSomething;
BEGIN
        part a of instructions using 5 ms
//PROFILE-BEGIN:part-b
        part b of instructions using 10 ms
//PROFILE-END
        part c of instructions using 3 ms
END;
```

In this case, the runtime of the method would be 8 ms (displayed in the line for method DoSomething), run time inclusive child time would be 18 ms.
In the line for method DoSomething-part-b 10 ms would be displayed.

It might be that the results are not exactly the same because the processor cache is used differently, especially processors with a small cache have the problem, that not the whole method inclusive measurement parts of ProDelphi fit into the cache, so additional wait states occur.

Remark:

It is possible to define more than one measurement block in a method, or to nest these blocks. Nesting might not be a good idea because the results might be misinterpreted.

Example for nesting:

```
PROCEDURE DoSomething;
BEGIN
//PROFILE-BEGIN:part-a-b
        part a of instructions using 5 ms
//PROFILE-BEGIN:part-b
        part b of instructions using 10 ms
//PROFILE-END
//PROFILE-END
        part c of instructions using 3 ms
END;
```

In this example, the runtime for part b is displayed separately AND also included as child time of part a (and, of course, also in the child time of DoSomething).

A.6. Programming API

A.6.1. Measuring defined program actions through Activation and Deactivation

A good way to make different result files comparable, is to measure only those actions of your program you want to optimize. In that case, do not check the button for 'automatic start' of measurement. Do the instrumenting of your source code and insert activation statements at the relevant places.

Example1 (for VCL-Applications*):

You only want to know how much time a sorting algorithm consumes and how much time all called child methods consume. You are not interested in any other method. The sorting is started by a method named button click.

```
PROCEDURE TForm1.ButtonClick;
BEGIN
{$IFDEF PROFILE}ProfStop; Try; ProfEnter; end; {$ENDIF}
   SortAll;  // the method of which you want to know the runtime
{$IFDEF PROFILE}finally; call ProfExit; end; {$ENDIF}
END;
```

You can modify the code in three different ways:

```
{ possibility 1 }
PROCEDURE TForm1.ButtonClick;
BEGIN
{$IFDEF PROFILE}ProfStop; Try; ProfEnter; end; {$ENDIF}
{$IFDEF      PROFILE}try; ProfInt.ProfActivate;{$ENDIF}
   SortAll;  // the method which you want to know the runtime of
{$IFDEF      PROFILE}finally; ProfInt.ProfDeactivate; end; {$ENDIF}
{$IFDEF PROFILE}finally; ProfExit; end; {$ENDIF}
END;

{ possibility 2 }
PROCEDURE TForm1.ButtonClick;
BEGIN
{$IFDEF      PROFILE}try; ProfInt.ProfActivate;{$ENDIF}
   SortAll;  // the method which you want to know the runtime of
{$IFDEF      PROFILE}finally; ProfInt.ProfDeactivate; end; {$ENDIF}
END;

{ possibility 3 }
//PROFILE-NO
PROCEDURE TForm1.ButtonClick;
BEGIN
{$IFDEF      PROFILE}try; ProfInt.ProfActivate;{$ENDIF}
   SortAll;  // the method which you want to know the runtime of
{$IFDEF      PROFILE}finally; ProfInt.ProfDeactivate; end; {$ENDIF}
END;
//PROFILE-YES
```

You should use possibility 1 or 3 because a new instrumenting does not change your code, Possibility 2 is changed by the next instrumenting into possibility 1.

Be sure that you use more than one space between $IFDEF and PROFILE you inserted, otherwise the statements will be deleted the next time that the source code is instrumented by ProDelphi. Alternatively, you also can use lower case letters.

**** For CLX-Applications instead of ProfInt..... use ProfIntc.....***
**** For FMX-Applications instead of ProfInt..... use ProfIntf.....***

Example 2 (for VCL-Applications *):

You want to activate the time measurement by a procedure named button1 and deactivate it by a procedure named button2 use the following construction:

```
//PROFILE-NO
PROCEDURE TForm1.Button1;
BEGIN
{$IFDEF      PROFILE}ProfInt.ProfActivate; {$ENDIF}
END;

PROCEDURE TForm1.Button2;
BEGIN
{$IFDEF      PROFILE}ProfInt.ProfDeactivate; {$ENDIF}
END;
//PROFILE-YES
```

Deactivation switches off the measurement totally. That means that no method call is measured until activation.

**** For CLX-Applications instead of ProfInt..... use ProfIntc.....***
**** For FMX-Applications instead of ProfInt..... use ProfIntf.....***

A.6.2. Preventing to measure idle times

Some Windows-API functions and Delphi functions interrupt the calling method and set the program into an idle mode. A well-known example is the Windows-call MessageBox. This call returns to the calling method after a button click. Between call and return to the calling method, the program consumes CPU cycles. In such a case, it would be nice, not to measure this idle time.

A lot of Windows-API calls and some Delphi-calls are replaced automatically by the Unit 'Profint.pas'. For the above named example MessageBox, there is a redefinition. It automatically interrupts the counting of CPU-cycles for the calling method and reactivates it after returning from windows.

If other methods are called while waiting for user action, they are measured normally, e.g. if a WM_TIMER messages is received, and you have defined a handler for it.

To make this possible, there are the ProDelphi-API-calls StopCounting and ContinueCounting. In chapter A.10 you can find the list of calls, which are redefined in the unit 'Profint.pas'. They automatically call these functions before using the original Windows- or Delphi calls. Some functions are replaced by the profiler (e.g. Application.HandleMessage).

Some functions cannot be replaced by 'Profint.pas', specially object-methods. If you use such methods and do not want to measure their idle times, just exclude these calls by inserting the following lines:

```
{$IFDEF      PROFILE}ProfInt.StopCounting; {$ENDIF}

      Object.IdleModeSettingMethod; // THIS METHOD SHOULD NOT BE INSTRUMENTED !!!
                                    // see //PROFILE-NO

{$IFDEF      PROFILE}ProfInt.ContinueCounting; {$ENDIF}
```

Important:

Use more than one space between $IFDEF and PROFILE, otherwise the statements will be removed with the next instrumenting or by cleaning the sources. Alternatively, you also can use lower case letters.

**** For CLX-Applications instead of ProfInt..... use ProfIntc.....***
**** For FMX-Applications instead of ProfInt..... use ProfIntf.....***

A.6.3. Programmed storing of measurement results

Normally at the start of the program the file for the measurement results is emptied and only at the end of the program the measurement results are appended. If you need more detailed information, you can insert statements into your sources to produce output information where you like to.

Just insert the statement

```
{$IFDEF     PROFILE}ProfInt.ProfAppendResults(FALSE); {$ENDIF }
```

into your source. In that case, a new output will be appended at the end of your file and all counters will be reset.

Normally the headline of the result file will be 'At finishing application' any time new results will be appended to the file.

For this example you might want to use a different headline. If so, you can set the text for the headline by inserting

```
{$IFDEF     PROFILE}ProfInt.ProfSetComment('your special comment'); {$ENDIF}
```

into your source.

Another way to produce intermediate results is to use the *online operation window*. Any time you click on the 'Append'-button, the actual measurement values are appended to the result file and all result counters are set to zero (see also chapter A.8)

Important:

Use more than one space between $IFDEF and PROFILE, otherwise the statements will be removed with the next instrumenting or by cleaning the sources. Alternatively, you also can use lower case letters.

More important:

This call should be used in an unmeasured function only and only then, when all measured functions have exited. For functions that have not exited yet, the runtime for the current call will be zero.

**** For CLX-Applications instead of ProfInt..... use ProfIntc.....***
**** For FMX-Applications instead of ProfInt..... use ProfIntf.....***

A.7. Options for profiling

Profiling options are divided into three groups:

- Code instrumenting options: What to instrument.

- Runtime measurement options: How to measure and what to do with the results.

- Activation of measurement: When to start measuring runtimes.

- General options: Which Delphi version / file date.

A.7.1. Code instrumenting options:

Changing these options after instrumenting DO afford a new instrumenting to take effect !!!

Assembler procedures (Professional version only)

Assembler code is normally not measured (often assembler is a result of an optimization process). In the professional version, this option can be used.

Initialization and finalization

Normally, the initialization and finalization parts of the units are not measured. In case you want to do this, check the appropriate option if you use the keywords INITIALIZATION and FINALIZATION in your units.
Permit opti(mization)

It is strongly advised to compile the measured application without optimization! ProDelphi checks the project file for the optimization option. If optimization is enabled, ProDelphi inserts an $O- statement into the sources. They are active only when the compiler symbol ‘PROFILE’ is defined.

If it is necessary that the measured application has to be compiled with optimization and the $O- statement shall NOT be inserted into the sources, this option needs to be checked.

Beware that the measurement results are less accurate when the measured application was compiled with optimization !!!

Packages

If this option is checked, also the DPK-files found in the directory in which the DPR-file is stored are processed. For further information, look in chapter A.9.

Profile local procedures

Normally local procedures are not instrumented and measured, if you activate this option they are.

Profile Library path (Professional Mode only)

Normally only the files belonging to a project are instrumented. These files are all the files in the unit search path of the actual project. Files which belong to components which are linked to the program by the linker are not instrumented. Usually they are profiled separately once and then not again with every project. That’s why they are normally excluded.

This option opens the possibility to measure also the files in the library path. For doing so, carefulness is necessary. If you include these files, the sources are instrumented with profiler statements. If you, after profiling one project, change to another project, the files are still instrumented. This means that you measure the runtime of the library sources in all other projects too. This measurement then slows down all other projects which are using these files. To prevent this, you should clean the sources before changing to another (unprofiled) project.

Also, when you want to profile another project, you need to be careful. As long as the files in the library path are not cleaned, you need to activate this option in all projects.

If you exclude directories from the library path, they need to be excluded in all projects, otherwise in the results you might find undefined methods.

Process write protected files

Checking this option means, that all write protections for your source files are ignored, and the files are instrumented. Without this option, write protected files are not processed.

Program + DLL’s / Mult. DLL’s

Checking this option means, that you either want to measure multiple DLL’s or a program + the used DLL(‘s). See chapter A.9 for details.

A.7.2. Runtime measurement options

Changing these options after instrumenting do NOT afford a new instrumenting.

Count Inherited calls for parent

This option is only valid for object methods (procedures and functions belonging to classes).
Normally, times are measured separately for each method. Use this option if you want, that, if a method is called by a method with the same name of an upper class (e.g. by INHERITED), the time of the inherited method is counted for the calling method.

Deactivate functions consuming < 1 µS

Any time the measurement results are stored in the result file, those methods are deactivated, which
- are called at least ten times AND
- have not called child methods, except deactivated ones AND
- consume less than 1 µs.

The names of the deactivated functions are stored in the file 'ProgramName.SWO' for the next run.
The runtime of the deactivated functions is added to the calling function.

And do not instrument in future (Professional version only) *(only if previous option is selected)*

The next time the sources are instrumented, the instrumentation code for all deactivated methods is deleted. The purpose of this feature is to reduce the overhead occurring by the code used for runtime measurement.

Generate data for call graph

Checking this option makes it possible that the ‘Caller/Called graph’ can be used when viewing the measurement results with the viewer. Of course, using this function needs more overhead than measuring without this function.

Online operation window on top

Normally the online operation window is displayed as a secondary window, that means that it is hidden by the main window. With this option, you can enforce to display it above the main window.

No Online operation window

The online operation window will not be displayed. So no intermediate measurement results can be stored.

Store a copy of result files

The measurement result files have the names

‘application name.xxx’

and contain the last measurements. If, e.g., more than one developer measures the same application this might be a problem, nobody knows whose results are actually stored. If this option is checked, the measurement results are additionally stored under the names:

‘application name-username-date-time.xxx’.

Also, like this, the different steps of optimization can be documented.

Testee contains threads

If this option is checked, the measurement is enhanced for handling threads. It is not useful to check this option if your program does not create threads, the program only runs slower. But it is necessary to check this option if you use threads, otherwise the results of the measurement are completely wrong.

Main thread only
If this option is checked, only the measured times of the main thread are measured. Times of child threads are ignored.

Evaluate minimum and maximum run times (Professional version only)

If this option is checked, the measurement routine of ProDelphi additionally estimates minimum and maximum run times of every method. Normally, only average times are estimated. Minimum and maximum times later can be displayed on demand by the built-in viewer. For some special purposes, this function can be used. Of course, using this function needs more overhead than measuring only average times.

A.7.3. Measurement activation options

Changing these options after instrumenting do NOT afford a new instrumenting.

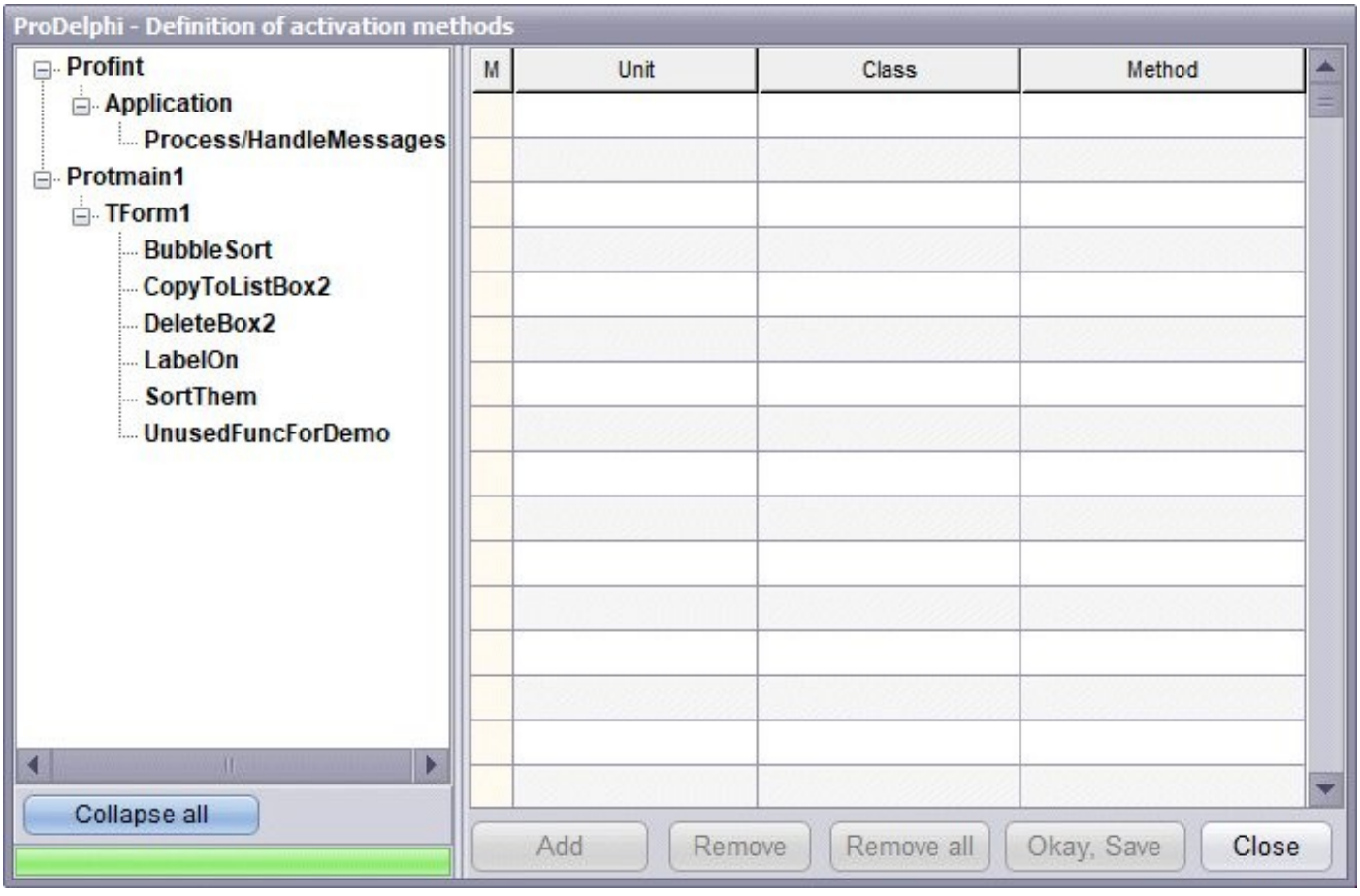
At program start (default)

If this option is checked, the time measurement will start when your program is started. In that case, the 'Start'-button in the online operation window is disabled and the stop button is enabled. If the option is not checked, the 'Start'-Button is enabled and the 'Stop'-button is disabled.

By entering a selected method

You'll be requested to enter methods (or you have already inserted //PROFILE-ACTIVATE statements into your source code (see also chapter A.5.2). If you use this option, you should not use the Online-operation window.

Example:



By API-Calls or online operation window

(see chapter A.6 and A.8 for details)

A.7.4. General options

Delphi version

You should check the version to that Delphi version you are going to compile the program with. This assures that ProDelphi uses the correct compiler switches. If ProDelphi is started via tools menu, the Delphi version is automatically set to the correct one.

File Date

The checkbox ‘Do not change file dates and archive attributes’ is available in the professional version only. Checking this results in increasing the file date/time by 2 seconds when instrumenting, enough to make Delphi realize that a file has changed. Unchecking means that actual date and time are used.

When cleaning the sources the file date/time is decreased by 2 sec’s for each instrumenting process resulting in a file date which is identical to that one when starting instrumentation (unless the file is changed by the editor). This makes possible that the file date keeps the same between checking out and in from a source code control system.

As some source code control systems also check the archive attribute, ProDelphi keeps its status as it is before instrumentation.

Build configuration

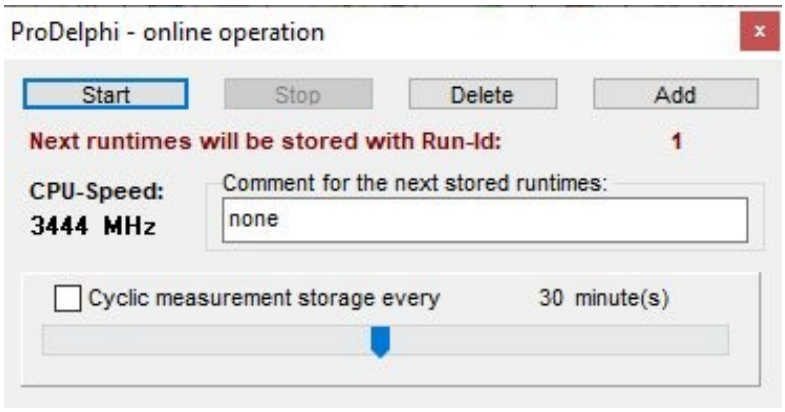
Set the build configuration to that you will use with the next compilation. If ProDelphi is started via tools menu, the build configuration is automatically set to the actual configuration. These buttons are enabled for projects of D2007 or better.

Class library

These buttons are enabled for Delphi 6, 7, XE and above only.
For Delphi 6 and 7 it is possible to select VCL- or CLX- class library.
For Delphi XE and above, it is possible to select VCL- or FMX- class library.

A.8. Online operation window

With the online-operation window



you can start and stop the time measurement. This enables you to measure only certain activities of your program. The 'Start'-button enables the measurement, the 'Stop'-button disables it. With the 'Delete'-button, all counters are set to zero. The 'Add' - button appends the actual counter values to the result file and sets the counters to zero.

You can edit the text which is the headline for the actually stored results. For the built-in viewer, any time, the results are stored, the 'Run-Id' is incremented, and you can switch between different runs with the viewer.

The default value for the headline for intermediate results is:

'none'.

Also, an automatic and cyclic storing of measurement results can be done. Use the slider to set the time cycle between 1 and 60 minutes. After that, check the box for cyclic measurement storage. After checking, the slider disappears until unchecked again. The results will automatically get date and time as headline. In the viewer, you can scroll through the results by the buttons ◄ and ►.

The online operation window is not available for Console applications !

A.9. Dynamic Link Libraries (DLL's) and packages

A.9.1. DLL’s

DLL's can be profiled the same way as programs. The only difference is, that, if you measure a DLL without the rest of the program, you won't have the online-operation window.

Some precautions are needed to avoid problems:

DLL’s can only be used with a calling program, no matter if you need the measurement results for code in the program or not. The DLL always expects the profiling information in the EXE-directory of the calling program. Also it stores the measurement results in that directory.

To ensure a problem-less measuring which works in all combinations (EXE only, DLL only, EXE + DLL, EXE + multiple DLL’s) with a minimum effort of handling, work as described in the following:

1. Check the option: Program + DLL’s / Mult. DLL’s in the profilers main window.

2. Make the units search path of all affected projects (EXE + DLL(‘s) identical.

3. Also, the directory for storing EXE- and DLL-file have to be identical.

4. If the DPR-file ‘USES’ files originating from directories which are not named in the search path, they need to be stored in the directory of the DPR-file..

> ***ProDelphi reads the search path and the compiler switches, depending on the Delphi version, from the DOF-, BDSPROJ- or DPROJ- file of the selected project. No matter which of the projects is profiled, you always have the instrumenting information and the measurement results of the correct units and all necessary code is instrumented.***

5. To select measurement results of a DLL or the program or both, just define the compiler switch PROFILE for the appropriate project and (re)compile the project. For the part you don’t want measurement results for, delete the symbol and (re)compile. Just by defining or not defining this compiler symbol, you can select the different measurement results.

If you measure the DLL without the program and need the online operation window an additional step is necessary:

In the USES-clause of the program, you’ll find:
Unitxyz{$IFNDEF PROFILE};{$ELSE}{},Profint;{$ENDIF}
before Application.Run; you’ll find:
{$IFDEF PROFILE } ProfInt.ProfOnlineOperation; {$ENDIF}

Just change / add two lines manually, so that the code looks like this:
*Unitxyz{$IFNDEF PROFILE};{$ELSE}{**,Profint;**},Profint;{$ENDIF}*
{$IFDEF PROFILE } ProfInt.ProfOnlineOperation; {$ENDIF}
{$IFNDEF PROFILE } ProfInt.ProfOnlineOperation; {$ENDIF}

(For VCL-Applications, for CLX-applications use ProfIntc instead of ProfInt, for FMX-applications use Profintf)

A.9.2. Packages

A.9.2.1. Delphi 2005 and above

Profiling design time packages is not recommended. Profiling runtime packages is supported partly. ProDelphi reads the compiler switches, compiler symbols and search directory from the project-file of a program. So for measuring methods in a package, one needs to profile the program that uses the package.

For enabling profiling of multiple packages at the same time, the DPK-files are instrumented: In the ‘Requires’ section a line for including an additional package necessary for that purpose is inserted.

There is no further evaluation of the DPK-files. All files of the packages to be instrumented have to be in directories which are named in the search path of the profiled program.

See below ‘Delphi - all versions’

A.9.2.2. Delphi 5 - 7

Profiling design time packages is not recommended. Profiling runtime packages is not supported active. The DOF-file of a package is not read, ProDelphi reads the compiler switches and compiler symbols from the DOF-file of a program. So for measuring methods in a package, one needs to instrument the program that uses the package.

In order to instrument the units belonging to the package, all files of the package have to be stored in directories that are named in the search path of the profiled program.

Because DPK-files are not evaluated, **only one package at a time** can be instrumented. Trying to instrument more than one package at a time will cause compiler errors.

See below ‘Delphi - all versions’

A.9.2.3. Delphi - all versions

The best way to profile a package is:

1. Put the sources of the package to be profiled into a separate directory.
2. Include that directory into the units search path of the program.
3. Instrument the program. This then includes to instrument the code of the package as well.
4. Recompile the package with the defined compiler symbol PROFILE.
5. Install the package.
6. Compile the program with the defined compiler symbol PROFILE.

If you now run the program, you’ll get the results for program + package.

Don’t forget:
Any time now you change the program by inserting or deleting functions and re-instrumentation, step 4 and 5 also have to be executed again.
Any time now you change the package by inserting or deleting functions and re-instrumentation, steps 4, 5 and 6 also have to be executed again.

A.10. Treatment of special Windows- and Delphi-API-functions

Some functions set the program into an idle mode until an event occurs and the function returns. It's not useful to measure these idle times. Because of that reason, some functions are redefined in the unit ‘Profint.pas’ or are replaced by the profiler in the source code. The result is that the idle time of the calling method is not counted, but other methods called while waiting are still counted.

Redefinition is always done the same way, this is shown be the example for the Windows Sleep function (defined in ‘Profint.pas):

```
PROCEDURE Sleep(time : DWORD);
BEGIN
  StopCounting;
  Windows.Sleep(time);
  ContinueCounting;
END;
```

Because of this redefinition, the unit Profint must be named after the units Windows and Dialogs. This is normally done. The only exception is, if you name these units in the implementation part of the unit. Delphi itself places them into the interface part.

If you find functions you want also to exclude from counting, you can make own definitions according to the example.

A.10.1. Redefined Windows-API functions

- DialogBox, DialogBoxIndirect, MessageBox, MessageBoxEx, SignalObjectAndWait
- WaitForSingleObject, WaitForSingleObjectEx, WaitForMultipleObjects, WaitForMultipleObjectsEx
- MsgWaitForMultipleObjects, MsgWaitForMultipleObjectsEx, Sleep, SleepEx, WaitCommEvent
- WaitForInputIdle, WaitMessage and WaitNamedPipe.

A.10.2. Redefined Delphi-API functions

- ShowMessage,
- ShowMessageFmt,
- MessageDlg,
- MessageDlgPos and
- MessageDlgPosHelp.

A.10.3. Replaced Delphi-API functions

- Application.MessageBox,
- Application.ProcessMessage and
- Application.Handle Message.

There are some VCL-functions which can't be replaced or redefined because they are class methods, it would be much to complicated. If you encounter measurement problems, just include them into StopCounting and ContinueCounting. An example for such method is TControl.Show.

A.11. Conditional compilation

A.11.1. Delphi 5

Conditional compilation is fully supported.

A.11.2. Delphi 6 and above

Conditional compilation is, except arithmetic expressions (like comparison with constants) supported.

The directives $IFDEF, $IFNDEF, $ELSE, $ENDIF, $DEFINE and $UNDEF are fully supported.

The directives $IF, $ELSEIF, DEFINED(switch) and $IFEND are completely evaluated inclusive the boolean expressions AND and NOT. Arithmetic expressions are always evaluated as TRUE.

These are the limitations:

{$IF const > x }	evaluated as TRUE	comparison with a constant
{$IF SizeOf(Integer) > 10}	evaluated as TRUE	Arithmetic expression

This is evaluated correctly:

```
{$IF NOT DEFINED(switch1) AND (DEFINED(switch2))}
```

This example causes problems:

```
CONST
  xxx = 6;
{$IF xxx > 5 }
  PROCEDURE AddIt(VAR first, second, sum : Int64);
  BEGIN
{$ELSE }
  PROCEDURE AddIt(VAR first, second, sum : Comp);
  BEGIN                    <- first Profiler statement is inserted after this BEGIN instead of after the previous
{$ENDIF }
    sum := first + second;  <- second Profiler statement inserted correctly here before END
  END;
```

Omitting the problem is very easy, just write it this way:

```
CONST
  xxx = 6;
{$IF xxx > 5 }
  PROCEDURE AddIt(VAR first, second, sum : Int64);
{$ELSE }
  PROCEDURE AddIt(VAR first, second, sum : Comp);
{$ENDIF }
  BEGIN                    <- first Profiler statement is inserted correctly after this BEGIN
    sum := first + second;  <- second Profiler statement inserted correctly here before END
  END;
```

A.12. Measuring on a customer PC

If an application has to be measured on a customer PC instead on the development PC, proceed like follows.

Beside the files belonging to the application copy also following files to the customers PC:

- Profmeas.dll
- ProfOnFo.dll
- ProDVer.dll
- Profcali.dll
- Proflst.asc
- Profile.ini

All files are stored in the exe-file directory of the application to be measured on the development PC.

A.13. Limitations of use

A.13.1. General

Console applications have no online operation window.

Methods in a DPR-file can not be measured.

Anonymous methods are not instrumented and so are not measured. This problem can be solved by insertion of two comments: //PROFILE-BEGIN:comment at the start of the body and //PROFILE-END before the end of the body. See chapter A.5.4.

The measured times always differ about 10 % from those of an unmeasured program. The reason is that the program code is not present in the cache as often as without measuring. On a multiprocessor machine, the results might differ even more.

For the purpose of instrumenting the source code, ProDelphi reads the sources. It is absolutely necessary, that the program can be compiled without any compiler errors. ProDelphi expects code to be syntactically correct.

While measuring, the profiler unit uses a user stack. The maximum stack depth is 16000 calls.
The maximum number of threads that can be measured simultaneously by ProDelphi is 32.
In the freeware version of ProDelphi only 20 methods can be measured, in the professional version 64000.

If the TForm methods WndProc or DefaultHandler are overwritten, measurement should be deactivated for these methods. If this is not done, a lot of measurement overhead is produced. In threaded applications the runtime inclusive child time can not be measured properly. This could even mean that the measured time for these methods is larger than the runtime of the complete program. Exclusion can be easily done by including these methods in //PROFILE-NO and //PROFILE-YES statements.

A problem for measurement is Windows itself. Because it is a multitasking system, it may let other tasks run besides the one you are just measuring. Maybe only for a few microseconds. So your program can be interrupted by a task switch to another application. I've made tests and measured the same routine again and again and each time I've got slightly differing results.

Don't forget also the influence of the processor cache. You might get different results for each measurement, just because sometimes the instructions are loaded into the cache already and sometimes not. This might be the reason, that sometimes an empty method needs some CPU-cycles for loading the entry code of the method into the cache.
The larger the cache size, the better the results ! The measuring methods use the cache too !

Then there is the CPU itself. The modern CPU's like Intel's Pentium or AMD's Athlon are able to execute instructions in parallel. When the profiler inserts instructions, the parallelism is different from without these instructions. That's another reason, why the runtime with measurement differs from that without measuring.

All my tests have shown, that the larger the cache is, the smaller the difference between the real runtime and the measured runtime is. With an AMD K6, the differences were only a few CPU-cycles.

If your measured program uses threads, the results are less correct. The reason is, that a thread change is not recognized at the time of change. It is recognized at the next method entry.

Be aware that, if you measure methods that make I/O-calls, you might also get different results each time. The reason is the disk cache of Windows. Sometimes Windows writes into the cache sometimes directly to the disk.

A.13.2. Delphi SpeedUp / FastMM units

The DelphiSpeedUp units RtlVclOptimize and VclFixUpPack units are excluded from instrumentation because they copy their own code into the Delphi standard units. This causes a crash of the application if these units are changed by instrumentation.

FastMM units also are not instrumented because of the above-mentioned reasons.

A.13.3. Aborted methods

If methods are aborted, e.g. when a program ends without that all threads have ended, no measurement results are available for the method that has not executed its exit code.

A.13.4. Measuring multiple applications

If more than one application have to be measured, it is important that their Exe files are stored into separate directories. The reason is that the files with the measurement settings and the method names need to have fixed names (profile.ini and proflst.asc).

It is not possible to measure more than one application at the same time !!! Only one instrumented application can be executed at the same time, otherwise wrong results are produced !!!

A.13.5. Excluding instrumentation of directories for all projects

You can do this by entering these directories as a character sequence under the registry key for ProDelphi:
- Enter the character sequence 'GLOBALEX' in the registry under HCU\Software\ProDelphi
- Set the value to the excluded directories (e.g. 'D:\components\Visual;D:\components\Graphic')

If a directory and all its subdirectories have to be excluded, add '*' at the end of a directory (e.g. 'D:\components*')

A.14. Assembler Code

Pure Assembler procedures and functions (e.g. FUNCTION Assi : Integer; asm mov eax,2; end;) are measured only in the Professional version.

A.15. Modifying code instrumented by ProDelphi

While working on the optimization of your program, you can of course modify your code. The only limitation is, that, if you define new methods and want them to be measured, you have to let ProDelphi instrument your code again. It is NOT necessary to delete the old statements inserted by ProDelphi before.

A.16. Hidden performance losses / Tips for optimization

ProDelphi measures runtimes of method bodies. This means that the entry part of a method which, e.g. writes variables to the stack, is measured in the calling method! The first possibility to take a time stamp is right behind the BEGIN-statement. This might be seen as a disadvantage compared to other profilers. But once you know this fact, it's no disadvantage any more. Anyway, changing of the number of parameters of a method changes always the runtime of the calling method (also for other profilers).

Below three examples for this.

- Passing Parameters:

```
FUNCTION TestFunction( s : String) : Integer;              // Runtime 5 CPU-Cycles + 983 in the calling procedure
BEGIN
  Result := Ord(s[1]);
END;

FUNCTION TestFunction(CONST  s : String) : Integer;    // Runtime 5 CPU-Cycles + 645 in the calling procedure (-33%)
BEGIN
  Result := Ord(s[1]);
END;
```

- Local variables:

```
FUNCTION TestFunction : Integer;          // Runtime 159 CPU-cycles + 126 cycles in the calling procedure
VAR
  i : Integer;
BEGIN
  FOR i := 1 TO 10 DO
    Result := LastFunction;
  IF Result > 0 THEN
    Exit
  ELSE
    Result := -1;
END;

FUNCTION TestFunction : Integer;          // Runtime 159 CPU-cycles + 6.932.128 cycles in the calling procedure
VAR
  i : Integer;
  yys : array [1..32000] of Integer;          // increasing caused by initialization of these local variables !!!
  yyv : array [1..32000] of String;
BEGIN
  FOR i := 1 TO 10 DO
    Result := LastFunction;
  IF Result > 0 THEN
    Exit
  ELSE
    Result := -1;
END;
```

- GoTo statements

```
FUNCTION TestFunction : Integer;          // Runtime 159 CPU-cycles + 126 cycles in the calling procedure
VAR
  i : Integer;
BEGIN
  FOR i := 1 TO 10 DO
    Result := LastFunction;
  IF Result > 0 THEN
    Exit
  ELSE
    Result := -1;
END;

FUNCTION TestFunction : Integer;          // Runtime 159 CPU-cycles + 177 cycles in the calling procedure (+ 40%)
VAR
  i : Integer;
  Label final;                            // Cause the additional runtime
BEGIN
  FOR i := 1 TO 10 DO
    Result := LastFunction;
  IF Result > 0 THEN
    GoTo final                            // in connection with this GoTo
  ELSE
    Result := -1;

  Final:

END;
```

A.17. Error messages

In case of errors, an error message is displayed by ProDelphi at the bottom line of its main window (e.g. file-I/O-errors). If that occurs, have a look into the directories with instrumented units.

Instrumenting a file is done in this way:

- the original file *.pas is renamed into *.pas_sav (or *.dpr into *.dpr_sav and *.inc into *.inc_sav),

- after that the renamed file is parsed and instrumented, the output is stored into a *.pas- , *.dpr- or *.inc- file),

- the last step to process a file is to delete the saved file, except an error occurs before.

This is done for all files of a directory. In case that an error occurs, you can rename the saved file to *.pas / *.dpr / *.inc.

Before doing so, maybe it's worth to have a look into the output file. In case of a parsing error, you can send the original file + the incomplete output file to the author for the purpose of analysis.

A.18. Security aspects

- Save all your sources before instrumenting (e.g. by zipping them into an archive).

- ProDelphi checks, if you have enough space on disk to store a instrumented file before instrumenting it. ProDelphi assumes that the output file uses 3 times the space of the original file (normally it uses less). If there is no sufficient space, it will stop instrumenting.

A.19. Automatic instrumenting, cleaning or viewing by start from command line

ProDelphi can be started from command line (or batch file). Depending on the command line parameters, ProDelphi can perform instrumenting of the source files, clean the sources or start the viewer.

A.19.1. Automatic instrumenting

If as arguments the Delphi version and the parameter /PROFILE are named, ProDelphi automatically instruments the named program and terminates after that.

Syntax:

Profiler path\application.dpr /Ddelphi-version /PROFILE
Profiler path\application.dpr /Ddelphi-version /PROFILE [/C=configuration] (for Delphi 2009 and better)
configuration : Debug, Release or user defined configuration
If not specified, it is read from the project file of the program.

Precautions:
- Instrumenting should have been done before interactively to be sure that all necessary data (e.g. DOF-file) exists.
- Instrumenting should not cause any warning in the profile log.

Example:
cd ProDelphi
Profiler F:\AppDir\Testprogram.dpr /D15 /PROFILE /C=Debug (for Delphi XE2)

Delphi Versions:	5 .. 7	2005 .. 2007	2009	2010	XE	XE2 .. XE8	10.0 .. 10.4
/D parameter:	5 .. 7	9 .. 11	12	13	14	15 .. 21	22 .. 26

A.19.2. Automatic cleaning

If as arguments the Delphi version and the parameter /CLEAN are named, ProDelphi automatically cleans the named program and terminates after that.

Syntax:

Profiler path\application.dpr /Ddelphi-version /CLEAN

Example:
cd ProDelphi
Profiler F:\AppDir\Testprogram.dpr /D6 /CLEAN (for Delphi 6)

A.19.3. Automatic opening of the viewer

If as arguments the Delphi version and the parameter /VIEW are named, ProDelphi automatically starts the viewer and shows the measurement results.

Syntax:

Profiler path\application.dpr /Ddelphi-version /VIEW

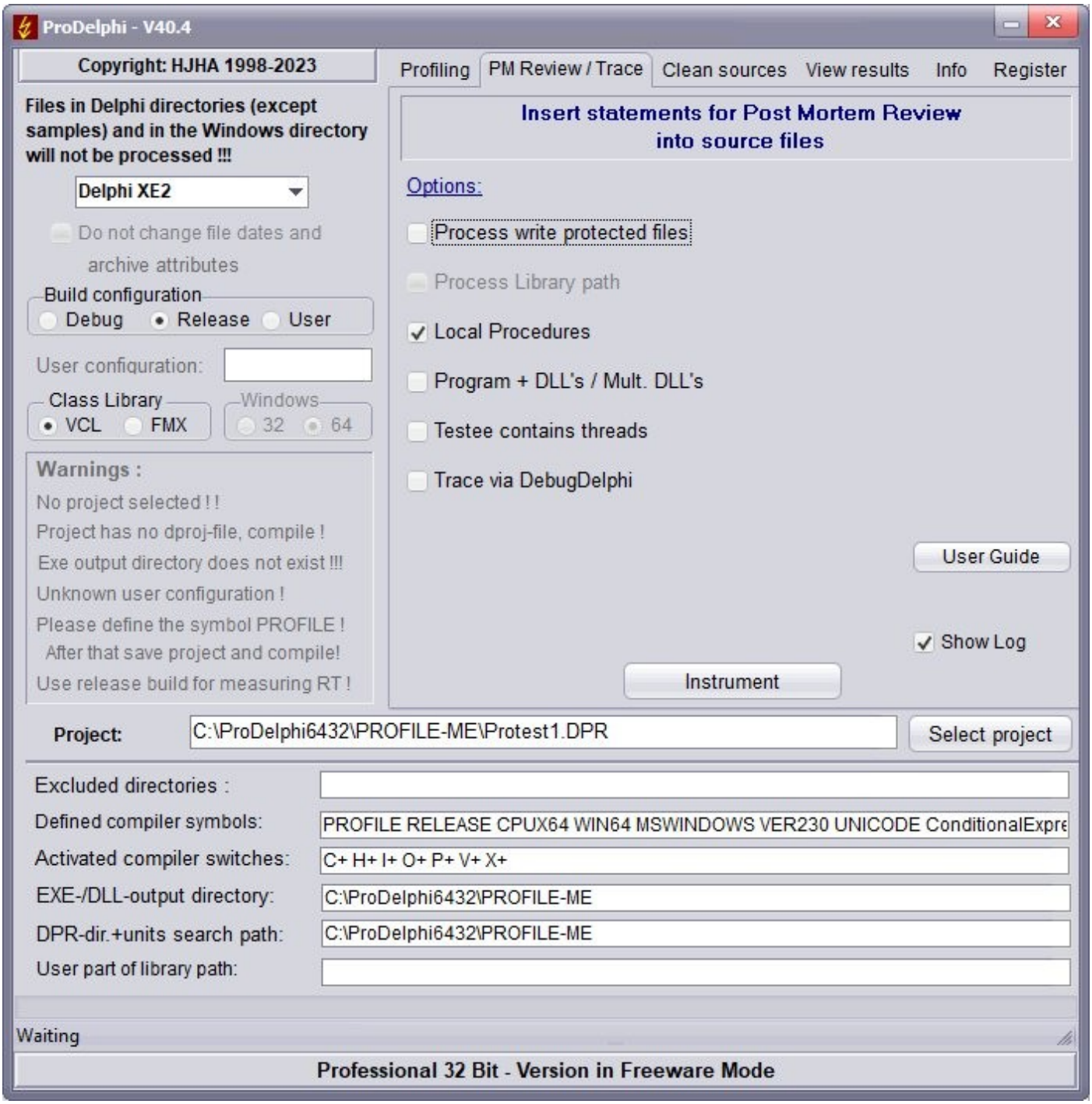
Example:
cd ProDelphi
Profiler F:\AppDir\Testprogram.dpr /D26 /VIEW (for Delphi 10.4)

A.20. National language support

Currently, an English and a German user interface is supported. Which to use is specified when installing ProDelphi with the setup program. But later ProDelphi can also switch between these two languages. This can be done by the system menu. It has two additional entries: ‘English’ and ‘Deutsch’ (= German).
If another language should be used: There is a file installed with the name ‘TranslateMe.LAN’. It contains the English text strings. By translating these and renaming the file into ‘Deutsch.LAN’ one can produce a user interface for his own language (and loose the German translation).

B. Post mortem review

As mentioned above, ProDelphi can instrument your sources with statements for post mortem review. It also interprets the sources and inserts statements at the beginning and at the end of a method.



In case of an abortion because of an exception, a message box will open which will give you the filename where the call stack is listed (ProgramName.PMR).

Also, here the source comments //PROFILE-NO and //PROFILE-YES can exclude parts of your sources.

The handling of ProDelphi is the same as for profiling. You also have to define the compiler symbol PROFILE.

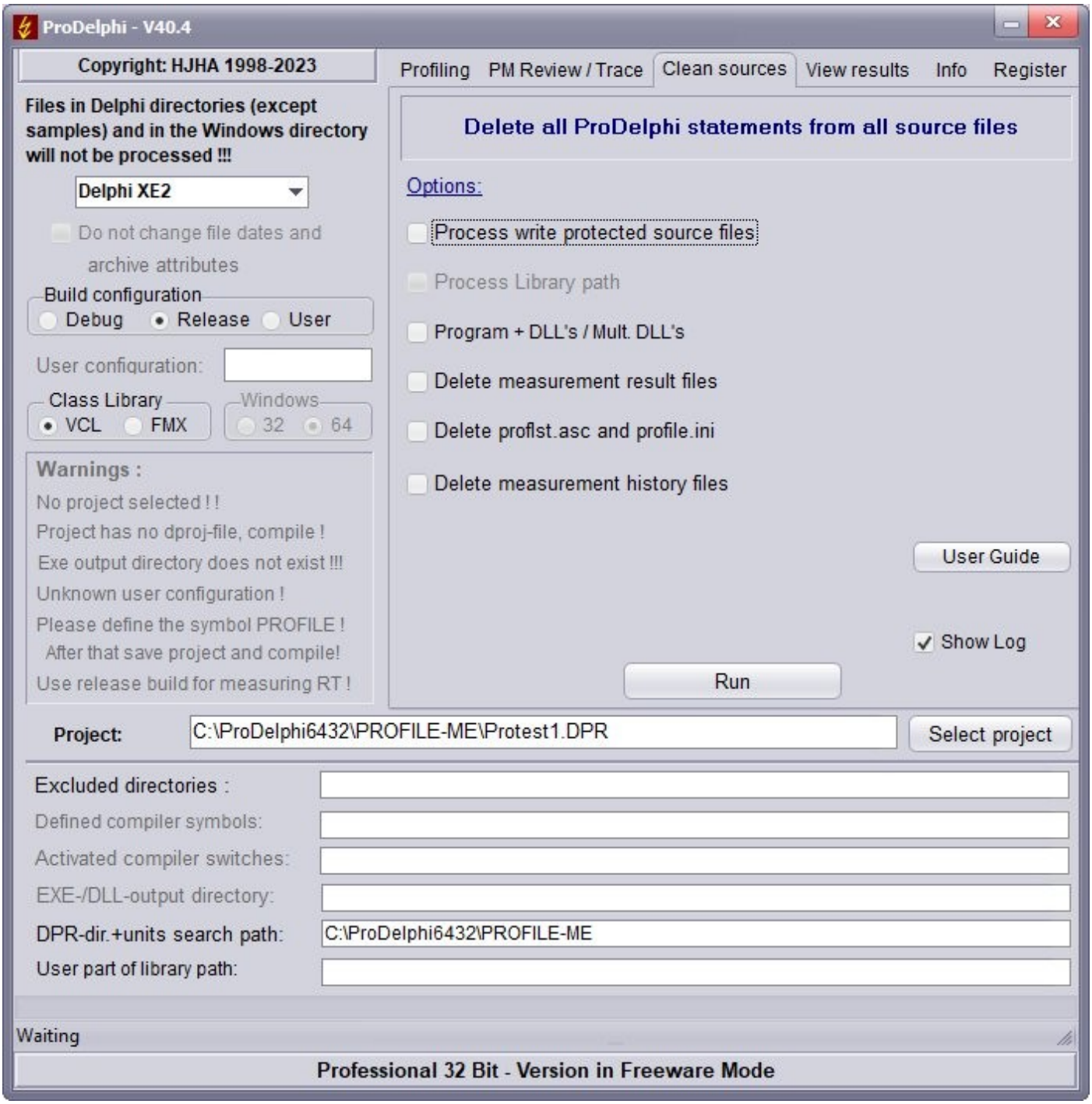
If you have instrumented ProDelphi with statements for post-mortem review and work with the IDE of Delphi and an exception occurs, you must continue your program unless you have deactivated the option 'Stop at exception'.

Limitation of use: Stack overflows are not caught because ProDelphi itself needs stack space. And if there is no stack any more, ProDelphi can not work properly. The overflow might as well appear in the ProDelphi stack tracing routines. ProDelphi can not handle this.

If the **Trace option** is checked, additional WriteLn calls are inserted into the source file. These WriteLn calls produce trace information which can be viewed with DebugDelphi. For this purpose, DebugDelphi needs to be installed and started.

C. Cleaning the sources

If you want to delete all lines that ProDelphi inserted into your sources, use the 'Clean' command.



It is not necessary to clean the sources if you simply want to let your program run without time measurement for a short time only. In that case, just delete the compiler symbol 'PROFILE' in your projects options.

It is also not necessary to clean the sources if you want to use the 'Instrument' command another time. Each instrumenting process automatically deletes all old ProDelphi statements in the source code and inserts new statements. For that purpose, it scans the code for statement that start with

```
{$IFDEF PROFILE}  and with  {$IFNDEF PROFILE }
```

and deletes them completely (except you have more than 1 space between IFDEF and PROFILE).
The option ‘Do not change file dates’ makes that the file date is increased at instrumenting by 2 sec and decreased at cleaning by 2 sec. This makes possible that the file date keeps the same between checking out and in from a source code control system.

D. Compatibility

ProDelphi was tested under
- Windows 95, 98, Windows NT 4.0, Windows 2000, Windows XP and Windows Vista, Windows 7, 8 and 10
- AMD K6 166 / 233 MHz, AMD K6-2 266 / 300 / 500 MHz, AMD K6-3 400 MHz, AMD Athlon 3000 MHz,
- AMD Turion X2 64, AMD Turion X2 64
- AMD Duron 1100 MHz,
- Pentium Overdrive 120 MHz, Pentium II / 400 MHz, Pentium III 750 MHz, Pentium Celeron 400 MHz, Pentium IV 1 GHz, Core i5 3.2 GHz, Pentium 2.5 GHz

E. Installation of ProDelphi

ProDelphi is most comfortably installed with the included setup program (Setup.Exe). This program copies all necessary DLL's into the installation directory and all needed units into the Delphi-LIB-directory. The editor interface is registered. Also, it creates an entry in the list of programs (Windows Start menu / Programs) and integrates ProDelphi into the Delphi tools menu.

F. Description of the result files (for data base export and viewer)

The result file can also be used for export to a data base (e.g. Paradox) or a spreadsheet program like OpenOffice Calc.

File content of 'progname.txt' (one line for each procedure):

run; unitname; classname; procedurename; % of RT; calls; minimum RT excl. child or 0; average RT excl. child; maxim. RT excl. child or 0;
RT-sum excl. child; minimum RT incl. child or 0; average RT incl. child; maximum RT incl. child or 0; RT-sum incl. child; % incl. child; procedure-no; call graph data;

Description of call graph data:

0;0; if no call graph data exists

or

Called-From-Information Calling-To-Information

Description of Called-From-Information:

0;	if not called (top-level procedure)
1..15; between 1 and 15 sets of Type-1-Info	if called by up to 15 procedures
16; 15 sets of Type-1-Info and 1 set of Type-3-Info	if called by 16 or more procedures

Description of Calling-To-Information:

0;	if not calling any procedure
1..15; between 1 and 15 sets of Type-2-Info	if calling up to 15 procedures
16; 15 sets of Type-1-Info and 1 set of Type-4-Info	if calling 16 or more procedures

Description of Type-1-Info:
No of procedure called from ; No of calls ; Runtime incl. child times used by the calling procedure ;

Description of Type-2-Info:
No of procedure called to ; No of calls ; Runtime incl. child times used by the called procedure ;

Description of Type-3-Info:
0 ; No of procedures called from not included in the first 15 procedures ; Runtime incl. child times used by these procedures ;

Description of Type-4-Info:
0 ; No of procedures called to not included in the first 15 procedures ; Runtime incl. child times used by these procedures ;

The procedure names can be found by searching the procedure number in Proflist.Asc. For each procedure measured there is following information:

Procedure number; Unit name; Class name; Method name; Filename; line number in the file

File content of 'progname.tx2' (one line for each run):
run; CPU-clock-rate; keyword; headline for that run //keyword is either MINIMAXON or MINIMAXOFF

G. Updating / Upgrading of ProDelphi

Updates and upgrades of the freeware version can be loaded via author's home page. Every new release will automatically be stored there. Just click on 'News' to see which version is actual.

H. How to order the professional version

Customers who want to use the professional version, can order it via ShareIt Registration Service. A special download link for downloading the professional version and a registration key will be sent via e-mail. Just start the professional version of ProDelphi (Profiler.exe), select the page for registration and enter the registration number. At the next start of ProDelphi, the Professional mode is unlocked. This key is also valid for upgrading to later versions.

I. Author

Helmuth J.H. Adolph (Dipl. Inform.)
Am Grünerpark 17
90766 Fürth
Germany

E-Mail: hejo.adolph@prodelphi.de
Home page: http://www.prodelphi.de

J. History

Version 1.0 : 9/97	First release
Version 2.0 : 2/98	Successfully used to optimize VICOS P500 for Sixth Railways project (China).
Version 3.0 : 4/98	Enhanced accuracy, brought to the public via Compuserve
Version 3.1 : 5/98	Enhanced granularity (1 CPU - cycle), published by Torry's Delphi Pages
Version 4.0 : 10/98	Viewer Added, export to data base, support of Delphi 4.
Version 5.0 : 11/98	Instrumenting statements changed to assembler (less overhead)
Version 5.3 : 12/98	DLL-Support added
Version 6.0 : 2/99	Treating of Read Only attribute, DLL-support enhanced, ProDelphi homepage
Version 6.3 : 5/99	Profiling assembler routines
Version 6.4 : 5/99	Setup program added
Version 6.5 : 7/99	Instrumenting of multiple directories added
Version 6.6 : 8/99	History function added
Version 7.0 : 9/99	Adaption to Delphi 5
Version 7.2 : 11/99	Profiler enhanced: Processing of relative pathnames.
Version 7.3 : 01/00	Profiler enhanced: Better accuracy, lower overhead, $IFOPT processing, In Professional mode 16000 methods can be measured (before 10000).
Version 7.4 : 02/00	Browser added for checking which procedure was not called.
Version 7.5 - 7.61:03-04/00	Different bug fixes
Version 7.62:04/00	In Professional mode 32000 methods can be measured (before 16000). Units search path editable, minor bugfixes
Version 8.0 : 05/00	Dynamic activation and deactivation of measurement by dialogue and by special comments in the source files, emulation of other PC's, main form arranged nicer, instrumenting log added.
Version 8.3 : 09/00	Viewer and Parser enhanced, Bugfix concerning not existing drive C: .
Version 8.4 : 10/00	Viewer consumes less memory, browser enhanced (TOutline replaced by TTreeView)
Version 8.5 : 12/00	More security in the user interface (warning if no DOF-file exists), the last project is stored and automatically selected in case of restart, for the viewer automatically, the result file is selected, sorting in the viewer is easier now (just click on the headline of the grid).
Version 8.51:01/01	Bug in counting inherited for parent fixed, new feature 'Measure only the main thread.
Version 8.54:02/01	New feature to keep the online operation window on top. Bug fixes: Class methods and class forward definitions can be handled, Units without Uses-statements can be processed, Processing of relative pathnames improved.
Version 8.55:04/01	Size optimization for one of the measurement DLL's (Profmeas.DLL). Search function of the viewer optimized. Processing of initialization and finalization part corrected.
Version 9.0 : 04/01	Delphi 6 support added.
Version 9.1 : 08/01	Printing of reports added.
Version 9.2 : 11/01	Documentation as PDF-File, button texts in window for selecting methods corrected, CLX support for Delphi 6, optical improvements, exclusion of directories, bug fixes.
Version 9.3 : 12/01	Bug in DOF-file processing (concerning Delphi 6)
Version 9.4 : 01/02	Printed report enhanced: alternatively printing in full colour or in colour saving mode
Version 9.5 : 01/02	Bugfixes: Names of local procedures were converted to upper case. When processing files with the read only attribute, the read only attribute was not set to true after instrumenting or cleaning. Overloaded functions were partly not recognized, same with some class definitions (this occurred in the implementation part and in include files). Improvement: The file creation date is optionally changed by 3 seconds only (Profess. mode)
Version 9.6 : 04/02	Bugfix: Two 'END'-statement in one line could have caused an instrumenting error, Improvement: Cyclic automatic storage of measurement results

Version 10.0:05/02	Evaluation of minimum and maximum runtimes, instrumenting of the library path, measuring of specified parts of methods, stack depth increased to 9600 entries, improvement of UI, rounding bug in sorting after change of runtime fixed.
Version 10.1:07/02	Upgraded for cross platform development (Windows / Linux), bug in treating Case-statement fixed, Setup program improved (handling of missing registry entries).
Version 10.2:07/02	Buggy
Version 10.3:07/02	Processing of environment variables in pathnames added. Instrumenting of files named in the USES-statement in the DPR-file of a program added. Simultaneous measuring of a program and DLL's or multiple DLL's improved.
Version 11.0:07/02	Adaption to Delphi 7.
Version 11.1:08/02	Parser enhancement: Procedure declarations over more than one line are handled now.
Version 11.2:09/02	Information for measured specified parts of procedures was missing if 'Local procedures' was not activated.
Version 11.3:09/02	Bugfix: Files named in the Uses - statement of a DPR-file were instrumented even they were write protected. Bugfix: Project settings were not persistent if DPR- and EXE-file were in different directories.
Version 11.4 10/02	Setup program improved: it is no longer necessary to manually install ProfMeas.dll. For upgrading from freeware version to professional version a complete distribution has to be downloaded from internet. Bugfix: Definition of SleepEx in the interface file corrected.
Version 11.5:11/02	Bug in displaying data after sorting the results with class or method as sort criteria (the columns which include child times were not actualized).
Version 11.6 01/03	Bugfix in parser resolved. Setup program fixed: start from network drive now possible.
Version 11.7 03/03	Viewer enhanced, bug in API resolved, user guide corrected
Version 12.0 03/03	Automatic opening of the source file in Delphi by clicking the method in the viewer, Multiple history files, print dialogue bug fixed.
Version 12.1 03/03	Correction of algorithm for estimating the CPU-speed regarding mobile processors.
Version 13.0 04/03	New features: caller / called graph, starting point list. Bugfix concerning Initialization part.
Version 13.2 05/03	Bugfixes: Calculation of recursive functions completely worked over, Emulation profiling did not emulate minimum and maximum values.
Version 13.3 05/03	Bugfix: It was not possible to have characters in a call for Application.MessaageBox that start a comment. Improvement: In the call graph recursively called method are marked.
Version 13.4 06/03	Bugfix: Compiler definitions in a unit were not valid in an include file, Compiler definitions in an include file were not valid in the including unit. Bugfix: Directories with a dot in their name could disable the history function.
Version 13.5 06/03	Bugfix: Applications with a language resource file aborted with exception. Bugfix: Check for instrumenting more than 32000 methods was missing, which caused partly wrong measurement results when this case occurred.
Version 13.6 07/03	Bugfix: History always used the file progname.hst in the exe-directory even another file was selected. Bugfix: A local FORWARD declaration caused that following procedures were not instrumented.
Version 13.7 08/03	Bugfix: A local FORWARD declaration caused wrong procedure names in ProfList.ASC so that result data was assigned to the wrong method. Bugfix: Environment variables in path names were not processed correctly if noted at the beginning of the path.
Version 13.8 08/03	Bugfix: Closing the Online operation window terminated the profiled application. Bugfix: Setup program deleted existing tools from the tools menu.
Version 13.9 08/03	Feature: Online operation window can be disabled Feature: Form for selecting activating methods improved Bugfix: Ini-file settings were lost
Version 14.0 10/03	Bugfix: Units with one uses statement were not instrumented if Uses was in the same line with implementation statement Feature: Number of activating methods increased from 16 to 32. User guide: New chapter about hidden performance losses. Bugfix: Setup was not possible on a PC with no C-drive (10/03)
Version 14.1 11/03	Enhancement: The option 'Do not change file date' now makes that the file date is increased at instrumenting by 5 sec and decreased at cleaning by 5 sec. This makes possible that the file date keeps the same between checking out and in from a source code control system. Enhancement: The Professional Version can measure 64000 methods (before 32000).
Version 14.2 11/03	Bugfix: Results for methods in the library path were missing (bug appeared in 14.1 only)
Version 14.3 11/03	Bugfix: Parser bug fixed
Version 14.4 11/03	Feature: Editing the path for directories to be excluded from instrumenting made easier and more safe. Trying now to analyse if the application contains threads and displaying a warning

	if so and the option ‘Testee contains threads’ is not checked.
Version 14.5 12/03	Bugfix: ProDelphi ended after clicking OK after thread warning message Bugfix: Result file was reported to be empty when results were added with the online operation window and no measured method was called.
Version 14.6 12/03	Feature: Environment variable in the output path possible now. Compatibility: Navigation interface can be used together with packages using ShareMem now. Bugfix: Parser bug for processing a line with Application.Messagebox containing an empty string now.
Version 14.7/8 1/04	Bugfix: An $IFDEF-/$ENDIF in a Uses statement caused problems when the ';' was inside $IFDEF-/$ENDIF.
Version 14.9 2/04	Bugfix: A space in the pathname of a directory below the Delphi source or lib directory caused files not to be instrumented if that directory was in the search path of the profiled application. Bugfix: A space on the path of the instrumented application caused that ProDelphi did not automatically select the project when started by the IDE.
Version 15.0 2/04	Improvement: Option ‘Do not change file dates’ improved. The file date is set back to the date valid when instrumenting the first time without cleaning before each instrumentation.
Version 15.1 3/04	Compatibility to DUnit added
Version 15.2 3/04	Bugfix: Option to keep the file date fixed.
Version 15.3 4/04	Improvement: Viewer hints + message 'no call graph data' can be disabled now. Improvement: Viewer window made better readable (colour change between odd end even line numbers). Improvement: Less overhead when measuring threaded applications. Improvement: Cursor gets hour glass when profiler is busy. Bugfix: To measure up to 64000 methods was possible only when not more than 32000 were in the search path (rest in the library). Bugfix: Viewer could not display more than 32000 lines. Bugfix: Threaded applications were not measured properly under very seldom conditions. Bugfix: Instrumenting a threaded application without checking 'Testee contains threads' could cause that the viewer could not read the measurement results.
Version 15.4 4/04	Bugfix: Some methods were displayed double in the call graph. Feature: Automatic instrumenting by start from command line added.
Version 15.5 5/04	Feature: Check if files in an Exe-file and a DLL/package are instrumented in the same instrumenting run to guarantee correct results. Feature: Warning if WndProc or DefaultHandler methods are measured Feature: Better error messages for some I/O-errors.
Version 15.5a 5/04	Security: Excluding directories from instrumenting enforces cleaning of sources to prevent double assignment to method numbers which would cause wrong measurement results. Bugfix: When ending an application with 'Halt' some measurement results were wrong. Bugfix: 'Application.Run;' in an IF-statement was not treated properly and so caused compilation errors.
Version 15.6 5/04	Bugfix: Comment in a MessageBox call caused Compilation errors.
Version 15.7 6/04	Improvement: Storing measurement results much faster now, important for cyclic storing of results.
Version 15.8 7/04	Improvement: Time for reading of measurement results drastically decreased. Improvement: Warning for inconsistent timestamp can be switched off
Version 15.9 8/04	Bugfix: A $DEFINE in an inactive $IFDEF caused that profiler statements were inserted at the wrong place in a $IFDEF - $ELSE construction. Bugfix: A unit statement with the unit name not in the same line as the unit keyword caused that the unit was not instrumented.
Version 16.0 8/04	Bugfix: Include files which are not in the search path were not cleaned.
Version 16.0a 8/04	Bugfix: A project with a dot (.) in the project name (e.g. xxx.yyy.dpr) can now be instrumented.
Version 17.0 11/04	Delphi 2005 support, improved automatic deactivation of measurements for methods consuming little CPU-time only.
Version 17.3 11/04	Minor fixes due to Delphi 2005, User Guide corrected
Version 17.4 1/05	Handling Delphi 2005 inline functions + printing of call graphs.
Version 17.5 1/05	Bugfix: Deleted handling of inline functions for D2..D7
Version 17.6 2/05	Bugfix: Exception occurred when processing nested include files
Version 18.0 2/05	Bugfix: Sorting in the call graph sometimes incorrect, Windows XP style skins for user interface
Version 18.1 5/05	Bugfix: IFDEF in Initialization made problems Feature: Viewer can filter for method name
Version 18.2 6/05	Improvement: Support for UTF-8 coded files added Feature: Automatic cleaning of sources by start from command line added.

	Feature: Automatic opening of the viewer by start from command line added.
Version 18.3 11/05	Bugfix: First unit named in the Uses-statement was instrumented even if stored in a directory which was excluded from measurement. Bugfix: Directory not found when a space was typed before the path separator in search path. Bugfix: Partly wrong measurement results for threaded applications if threads have a very long runtime. Feature: Optional German user interface.
Version 19.0 1/06	Delphi 2006 support
Version 19.1 8/06	Bugfix: //PROFILE-NO in the first 2048 characters of a file made that the file wasn't instrumented. Bugfix: Unit was not instrumented after end-statement with a preceding ')' without space or colon. Improved: Fonts for call graph changed to make form readable when large fonts are installed. Feature: When excluding directories from instrumenting all its subdirectories can optionally be excluded as well. Feature: Optional de-instrumentation of methods consuming less than 1 micro second. Improved: Package support for Delphi 2005 and above.
Version 20.0 9/06	Turbo Delphi Support. Exclude path length extended from 2047 to 4095 characters.
Version 20.0a 10/06	Bugfix: Inserted profiler statement to deactivate inline functions for Delphi 2006 was wrong (was for Delphi 2005)
Version 20.0b 2/07	Bugfix: Empty initialization section caused finalization statement to be deleted.
Version 20.0c 3/07	Bugfix: For Delphi 2005 and 2006 the INLINE OFF statement was inserted only then when optimization is switched on.
Version 21.0 4/07	Delphi 2007 support.
Version 22.x 11/07	Vista support and various small bugs and incompatibilities fixed.
Version 23.0 10/08	Delphi 2009 support.
Version 23.1 10/08	Problem with non-standard line ends solved (e.g. Citadel example code). Instrumentation optimized
Version 23.2 11/08	Problem with path name containing non-English characters solved.
Version 23.3 12/08	Viewer window opens with size and position of last program start.
Version 23.4 12/08	Unicode version only: An incompatibility in Delphi 2009 caused that source files stored in UTF8 code could not be instrumented. Empty lines were not stored. Instead of writing the empty line the previous (not empty) line was stored twice.
Version 23.5 2/09	Command line start without full pathname of DPR-file does not lead to crash any more The DelphiSpeedUp units RtlVclOptimize and VclFixPack are not instrumented any more, (instrumenting these caused a crash of the using application)
Version 23.6 2/09	Option 'Do not change file dates' did not have any effect, fixed.
Version 23.7 3/09	Command line start of ProDelphi with relative pathname is possible now.
Version 23.8 4/09	Profiling log visible now also for cleaning run Try or Case block in Initialisation or Finalisation part caused wrong instrumentation. Cleaning caused exception when a DPR-file has many comments at the end. Number of instrumented methods now displayed in status bar and profiling log. Maximum windows size for call graph window set. Measured runtime of methods calling DispatchMessage corrected.
Version 23.9 5/09	Paths containing characters '()' and environment variables, e.g. '($XYZ)' caused 'OutOfMemory', fixed. FastMM units are not instrumented any more because that stopped using FastMM memory manager.
Version 23.10 6/09	Checked option 'Do not change file dates' now also makes that the archive attributes are not changed. Paths containing characters 2 subsequent spaces were not instrumented. Delphi directories QCX and ObjRepos are excluded from instrumentation. Initialization parts starting with multiple comment lines caused wrong instrumentation and compilation error. Settings for excluded directories were lost for new projects when set before first instrumentation. CLX applications could not be measured (Unicode version only). German UI was not possible (Unicode version only). Profiling log can be copied to clipboard now. Nested include files could have caused errors. Cleaning sources did not clean all include files. Maximized viewer window could not be resized.
Version 23.11 6/09	Changed settings now are stored to file immediately instead at instrumenting time to make sure that no setting is lost.

	Include files were instrumented even they are stored in an excluded directory when a path was given in the include statement.
Version 23.12 7/09	Default directory for loading measurement results set to the correct directory. Explanation in 'First steps' made better understandable. Include files with file extension other than PAS or INC possible now.
Version 24.0 8/09	Support of Delphi 2010.
Version 24.x 6/10	Some smaller bugs fixed. Result files can be copied to files with unique file names.
Version 25.0 8/10	Adaption to Delphi XE.
Version 25.1 9/10	Improved viewer.
Version 25.2 10/10	Cosmetic improvements.
Version 25.3 3/11/	Besides Debug and Release build configuration now also user configuration is possible.
Version 25.4 4/11	Solved: Online-operation window did not remember last position Name clash with default handler procedures Ansi-strings in MessageBox calls were not possible.
Version 25.5 5/11	Solved: Checked option 'Program + DLLs' caused that files which are not stored in files of the units search path were not cleaned.
Version 25.6 7/11	All DLL's for runtime measurement moved to measured application instead of using Windows\System32 directory.
Version 26.0 8/11	Adaption to Delphi XE2
Version 26.1 9/11	Bugfix concerning anonymous method.
Version 26.2 10/11	Post Mortem Review now with Trace option. If this option is checked, every entering and leaving a method is listed in the DebugDelphi log. To use this option, DebugDelphi must be installed and started.
Version 26.3 2/12	A search path containing two path separators without a directory between caused exception.
Version 26.4 2/12	A Compatibility problem with JEDI solved. Problem with creation of temporary files on virtual machines solved.
Version 26.5 2/12	Compatibility problem in setup for Delphi 5 under Windows 7 solved. Enhancement of trace option for post mortem review.
Version 26.6 2/12	Setup improved for Windows 7. Cosmetic improvement for online operation window.
Version 26.7 2/12	Multiple entries of the same directory in the Delphi search path now without problems (multiple instrumenting of the same file or problems with exclusion of directories).
Version 27.0 2/12	Adaption to Delphi XE3.
Version 27.1 11/12	Problem with excluding directories solved.
Version 27.2 11/12	Viewer improved. Result can be exported to CSV-files (';' - separated)
Version 27.3 1/13	Improvement to make it possible that also instrumented files which are reformatted with the Delphi IDE can be cleaned or re-instrumented.
Version 27.4 3/13	Firemonkey apps could not be measured, fixed.
Version 27.5 3/13	UI beautified. German texts for UI added.
Version 28.0 4/13	UI Adaption to Delphi XE4.
Version 28.1 5/13	Bugfix: Exe-output directory was not read from all project files for XE .. XE4.
Version 28.2 6/13	Option to measure applications which were compiled with optimization (with less measurement accuracy).
Version 28.3 6/13	Optional German User Guide.
Version 29.0 9/12	UI Adaption to Delphi XE5.
Version 29.1 10/13	Environment variable in exe-path was not evaluated, fixed. For instrumenting programs to be compiled with D2007 selecting Debug or Release build is possible (they have different conditionals: DEBUG/RELEASE).
Version 29.2 11/13	Source code navigation with Delphi XE5 fixed.
Version 30.0 4/14	Adaption to Delphi XE6.
Version 30.1 5/14	Exclusion of instrumentation of directories for all projects.
Version 30.2 5/14	Viewer exception and size problem fixed.
Version 30.3 (6/14)	One error message added, one fixed.
Version 30.4 (7/14)	Missing German translations added. Warning 'No online operation window for console apps' added. Missing unit name for console apps in viewer added.
Version 31.0 (9/14)	Adaption to Delphi XE7.
Version 31.0a (1/15)	Bugfix in interface file (MessageDLG call could not be used with Delphi XE6 and XE7). Default value for measurement start set to automatic start (activates automatic start of measurement when Profile.ini is missing). Warning in online operation window when profile.ini is missing.

	Setup program did not detect Delphi 2007.
Version 31.1 (2//15)	Historical writing of compiler directives [e.g. (*$I filename *) instead of {$I filename }] added.
Version 31.2 (4/15)	Viewer improved.
Version 32.0 (4/15)	Adaption to Delphi XE8.
Version 33.0 (9/15)	Adaption to Delphi D10 - Seattle.
Version 34.0 (4/16)	Adaption to Delphi D10.1 - Berlin
Version 34.1 (8/16)	Start with command line parameters improved
Version 35.1 (4/17)	Adaption to Delphi D10.2 - Tokyo
Version 35.2 (4/17)	Viewer problem solved, Profiling log can be hidden
Version 35.3 (6/17)	One warning message extended, problem with abnormal dproj file fixed
Version 36.0 (3/18)	Optical improvement
Version 36.1 (7/18)	Print function for tables fixed
Version 37.0 (11/18)	Adaption to Delphi 10.3 Rio
Version 37.1 (3/19)	Issue with very long units search path solved
Version 37.2 (4/19)	Exclude-directory window made sizeable
Version 37.3 (5/19)	RunUp made faster
Version 37.4 (6/19)	Missing headlines for export file and exclusion window added
Version 37.5 (9/19)	Bugfix: Activation of measurement did not work when HandleMessages or ProcessMessages was called.
Version 38.0 (6/20)	Adaption to Delphi 10.4 Sydney
Version 38.1 (9/20)	Bugfix concerning measurement result of threaded application.dpr
Version 38.2 (10/10)	Automatic profiling improved: selection of configuration (Debug, Release, user)
Version 38.3 (3/21)	Call graph now resizeable, Call graph can be closed by ESC key Selected method in viewer is marked in blue Ctrl + Home and Ctrl + End in viewer realized Unit, class and method search improved
Version 38.4 (8/21)	Problem with Windows scaling text by 150 % solved Displaying of excluded directories fixed
Version 39.0 (11/21)	Adaption to Delphi 11
Version 39.1 (12/21)	Interface unit improved Closing the online operation window caused exception at program end, fixed
Version 39.2 (3/22)	Internal optimizations
Version 39.3 (6/22)	UI Beautyfied
Version 40.0 (10/22)	Wait time measuring
Version 40.1 (2/23)	Problem with Delphi 11 solved (Starting a measured program by IDE with debugger)
Version 40.2 (4/23)	Bugfix: Closing online operation window caused error when ending measured app Improvement: Viewer displays values with localized decimal separator (. ,)
Version 40.3 (4/23)	Wrong warning message deleted
Version 40.4 (5/23)	One click expanding for browsers added Callgraph window enlargement added
Version 40.5 (7/23)	Cosmetical improvement of the viewer Log improvement Some wrong hints corrected Progress bar fixed Switching off hints in the viewer deactivates hints in the call graph too English corrected Precision in the viewer selectable: 0, 1, 2 or 3 decimal places
Version 41.0 (11/23)	Adaption to Delphi 12 Viewer font changed to improve readability Paper printout font also changed to improve readability

K. Literature

How to optimize for the Pentium family of microprocessors by Agner Fog / 1998-08-01
C/C++ user journal 'A Testjig Tool for Pentium Optimization' by Steve Durham (December 1996).