ProDelphi User Guide

(Release 14.4)

Copyright Dipl. Inform. Helmuth J.H. Adolph 1998 - 2003

The Profiler for Delphi 2, 3, 4, 5, 6 and 7 (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 (Inprise, Corel or who ever) gave up the profiler for 32-bit applications, a new tool had to be created. ProDelphi with it's comfortable viewer, browser, history and programmers API meanwhile is more than the legendary Tubo Profiler. The viewer with it's sorted results enables the user to find the bottle necks of his program very fast. The history function shows the user, if a preceeding 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 7.4 even a simple coverage profiling can be performed. All methods that are not called while profiling are stored at the end of the testees run an can displayed by the built-in browser. If a coverage profiling on a line-by-line base is necessary, the coverage tool Discover can be strongly recommended (see links on the ProDelphi web page).

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

Also starting with release 8.0 measurement on another PC can be emulated. Instead of installing the complete IDE on a faster or slower PC now you can measure under your normal equipment and let ProDelphi recalculate the measurement results for any given PC.

With release 13.0 came a caller / called graph 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 pathes to profile.

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 / exception. ProDelphi realizes that function without the testee running under the IDE.

Differences between the freeware mode and the professional mode

In the freeware mode up to 20 procedures can be measured or tracked, in the professional mode 32000.
In the professional mode 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 profiled.

Date: 11/22/2003

0. Contents of this description

BEFORE using ProDelphi practically, please read Chapter 15 carefully !!!

A. Principle of Profiling

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

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

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 procedure.

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 (see A5).

A1 How to profile

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 developping 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 want to install ProDelphi manually, you need to perform the following steps:

Copy the files ProfCali.DLL, ProfMeas.DLL and ProfOnForm.DLL into the WINDOWS\SYSTEM - directory (for Windows 95/98) or into the WINNT\SYSTEM32 - directory (for Windows NT/2000/XP). Copy the files ProfInt.PAS and ProfIntc.PAS into into the Delphi LIB-directory. For opening a file in the editor by clicking on a measured procedure, the IDE-interface files need to be installed and entered in the registry in the section 'Known Packages'. These files have the name PDIFacx0.BPL (x = Delphi version number). this is possible for Delphi 5 to 7 only.

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

If you want to measure procedures in a program and in DLL's simultaneously, 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 profiled must be stored in directories of the units search path except those that have an explicite path in the USES-statement in the DPR-files of program or DLL. For profiling program and DLL simultaneously, the button 'Profiling program + DLL's / Multiple DLL's' must be checked (see also chapter 8).

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

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

Don't use the original units for profiling, 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 profiling your sources perform the following steps:

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

- Deactivate the Optimization option.

- Optionally deactivate all runtime checks.

- Use the Delphi 'Save All' command. This assures that the options file (*.DOF) is stored.

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

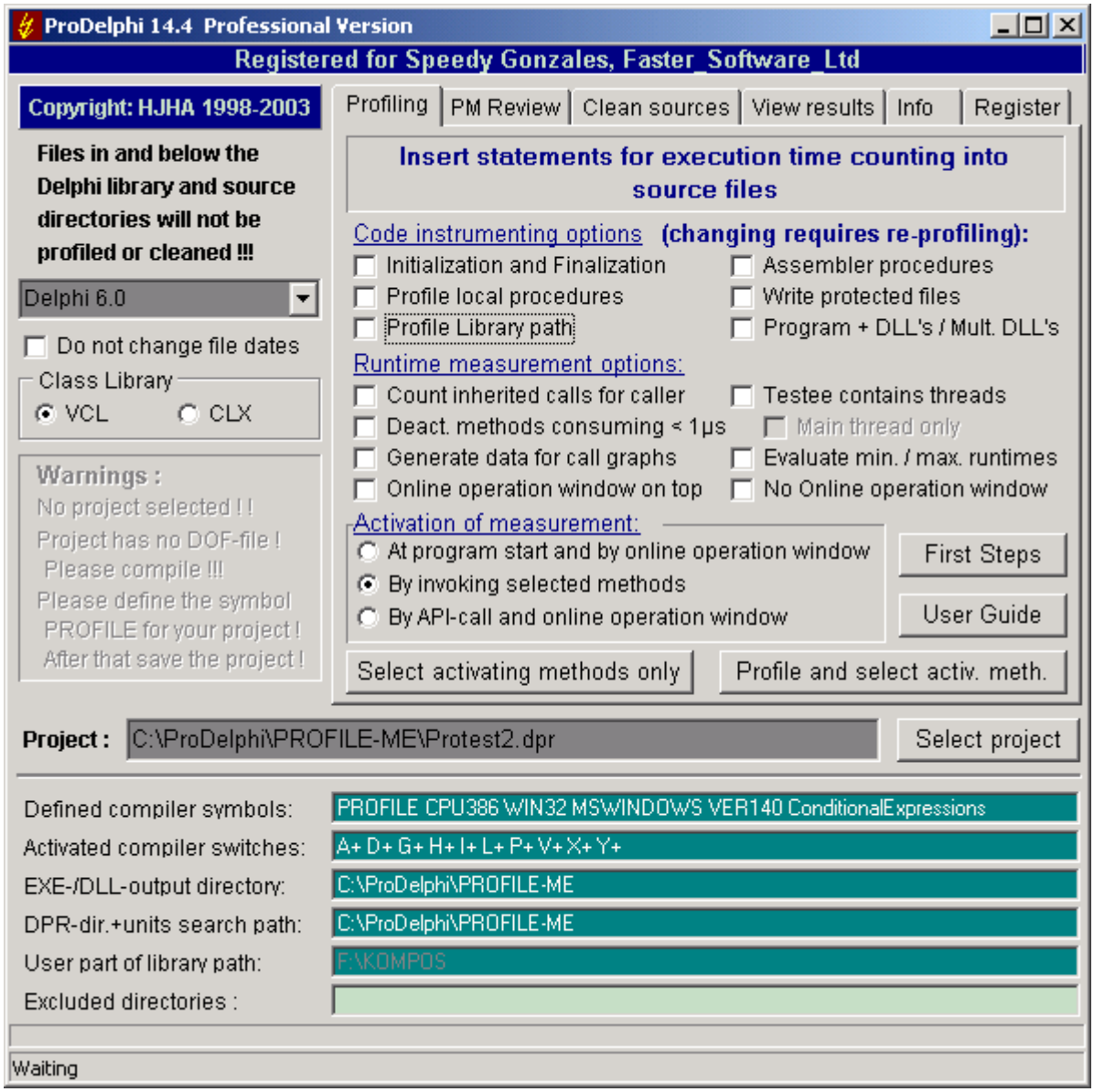
- With ProDelphi select the project to profile (if it is not automatically selected).

- 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 results in changing the filedate by 3 seconds only when profiling, just enough to make Delphi realize that a file has changed.

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



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

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

- *If you want to measure procedures in DLL's profile the necessary DLL's too.*

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

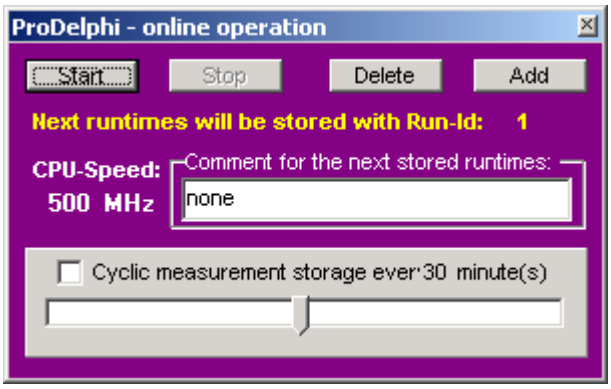
To allow simultaneous measuring of DLL's and programs, all files in the units search path are profiled!!! (unless they are write protected !!!). The unit search path must be exactly the same for program an DLL, both DPR-files have to be in the same directory !!!

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

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:

See next page for the Online operation window, please.



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 A7 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 principal 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.

A1.1 Files created by ProDelphi or the measured program

ProDelphi creates the file ‘proflst.asc’, it contains information about the procedures to be measured for profiling 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 A3).

Your compiled program the file with the name ‘progname.txt’ contains the data in the ASCII-semicolon-delimited format for data base export 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 procedures 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 into which the names of the uncalled methods are stored. This file is also used by the viewer.

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

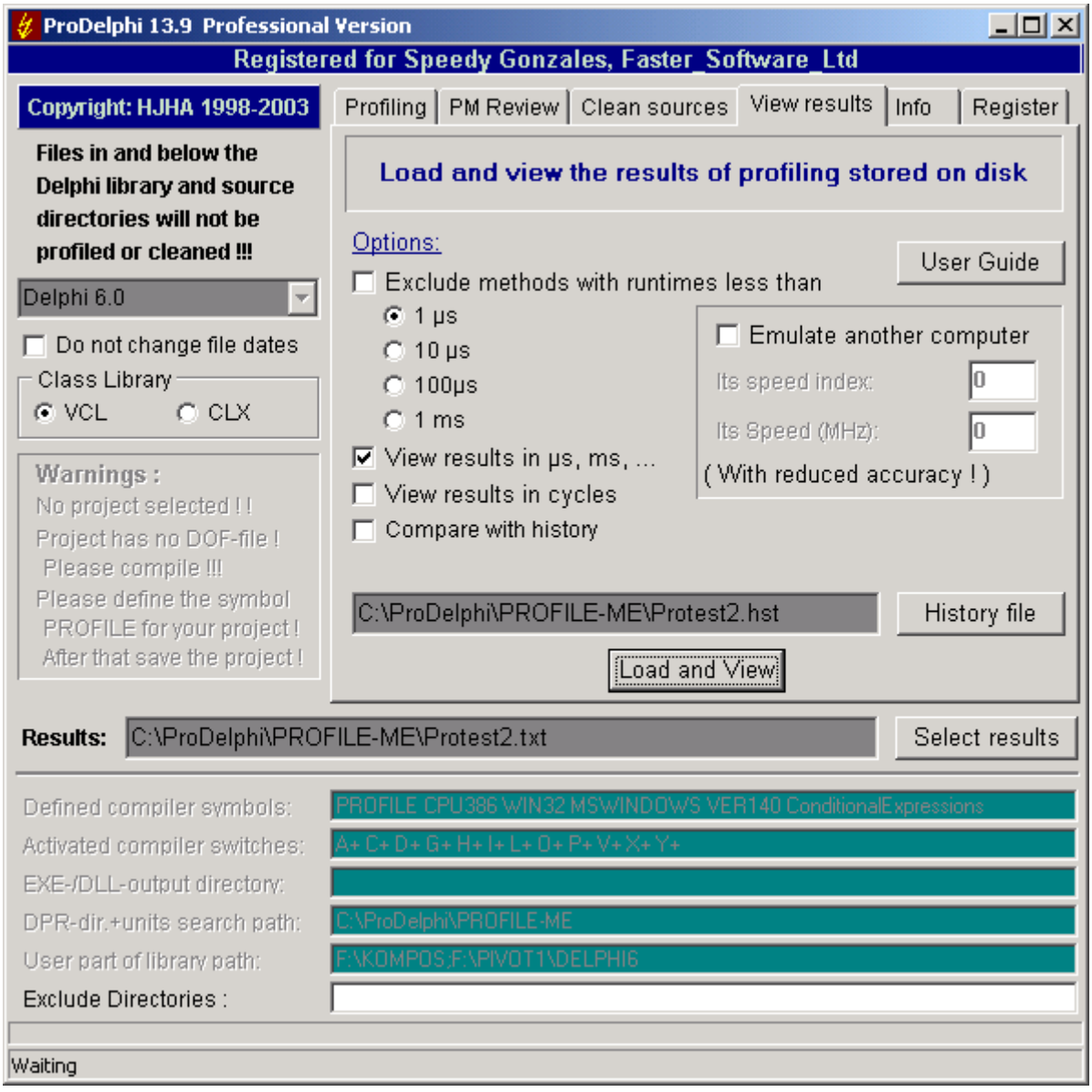
All files are stored in the output directory for the *.exe (*.dll) file.

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 profiled if they are not write protected !!! Seach path for program and DLL need to be identical in that case.

A1.2 Checking the results with the Built-in Viewer

The most comfortable way to view the run times of your procedures, is to use the built-in viewer. Just click 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.



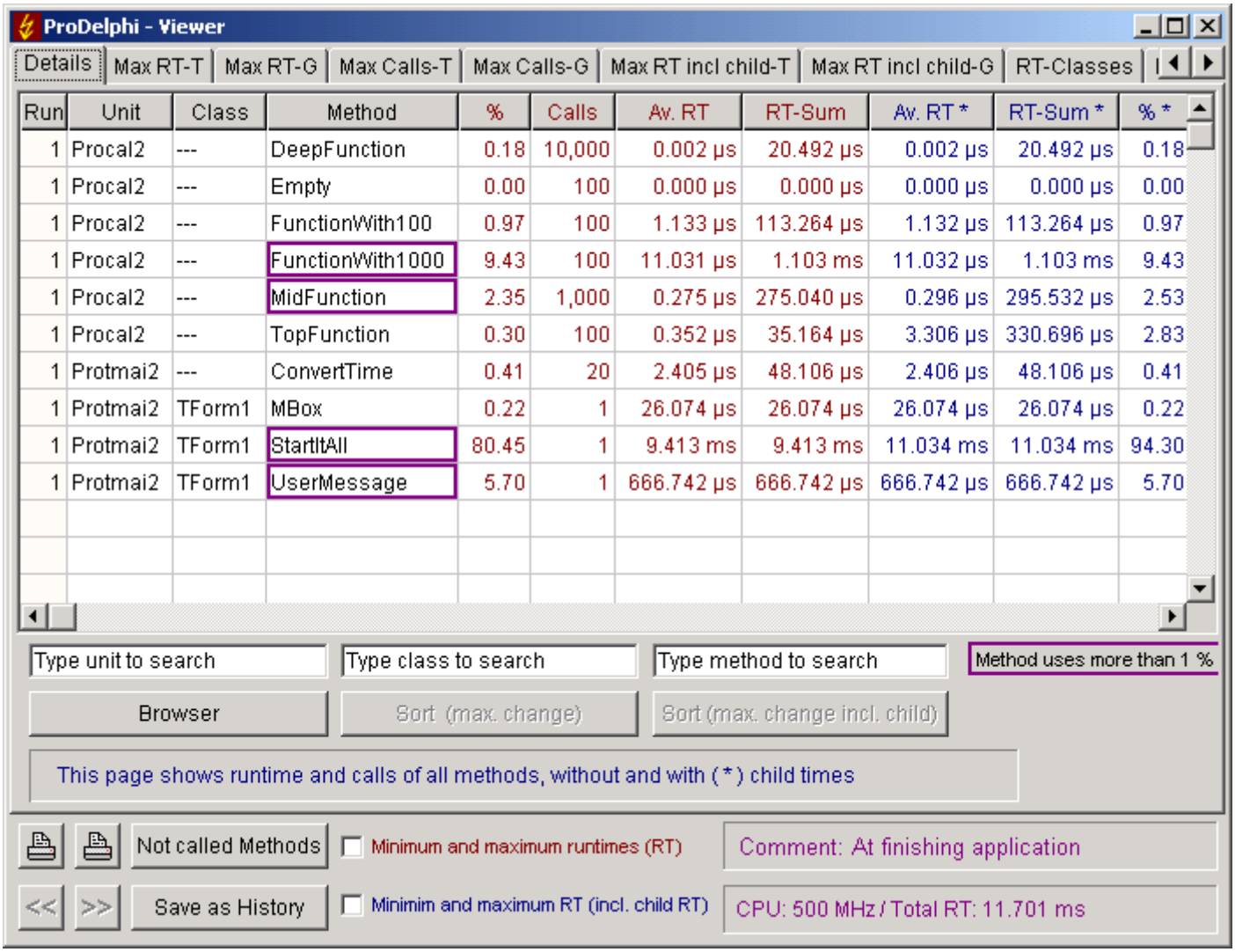
You can choose if you want to view the results in µs, ms ... or in CPU-Cycles.

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

Also you can emulate (recalculate) 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 1.3).

On clicking '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)
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 throgh 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 viewers 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. Available for Delphi 5 and above !!!

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 absolutely nescessary, color is used (color save mode). Using the second button, everything is printed as displayed on the screen (full color mode).

The Minimum/Maximum checkboxes (Professional Mode only):

Checking these options, minimum and maximum runtimes are displayed if they were collected in the measurement (see Chapter A 6.2. If these checkboxes are disabled, no minimum and maximum values were evaluated.

The History - button: see chapter A3

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

Meaning of the RED columns:

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

Meaning of the BLUE columns:

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

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 availlable pages show:

The 12 sorted methods that consumed the most of the runtime (**exclusive** child procedures) 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 procedures) 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:

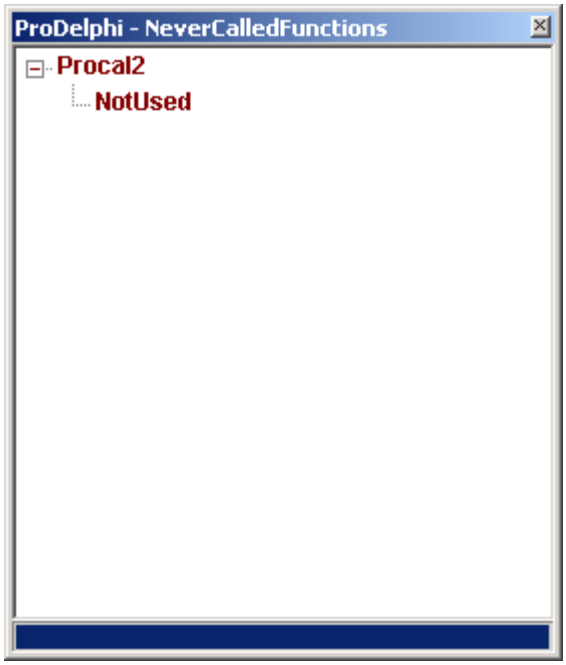
The runtime is greater as the time stored in the history file. The frame only then appears, if the change is greater than 1% of the total runtime of the application.

Meaning of runtimes inside a green frame:

The runtime is is less as 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.

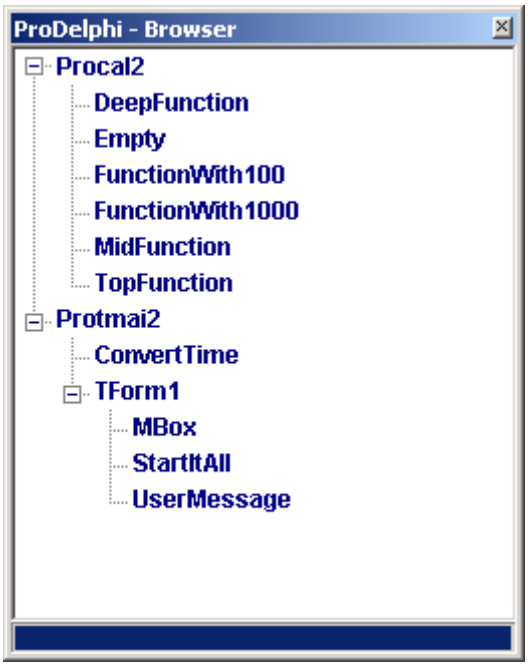
The Not called Methods - button:

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.

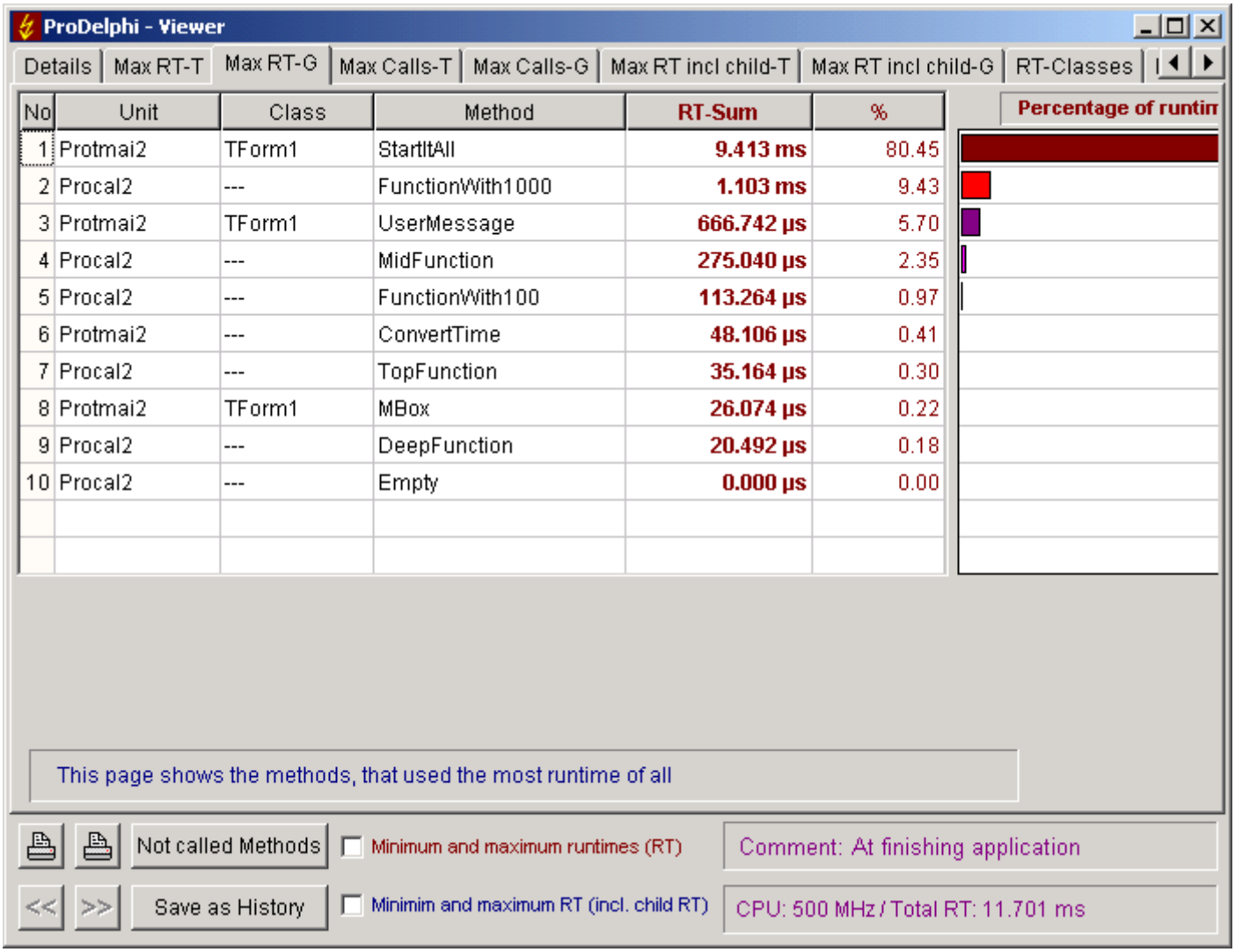


The Browser - button:

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



See next page please for another viewer window example



Example of: Maximum run time consuming methods (graphical)

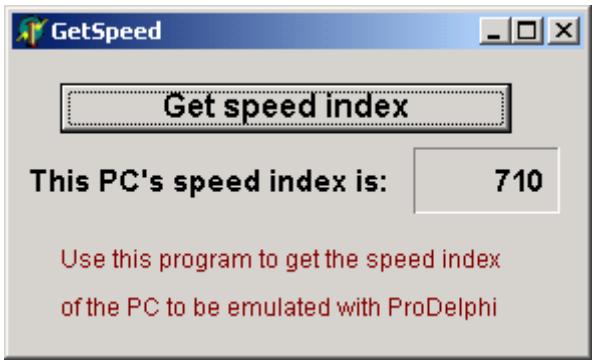
A1.3 Emulation of a faster or slower PC

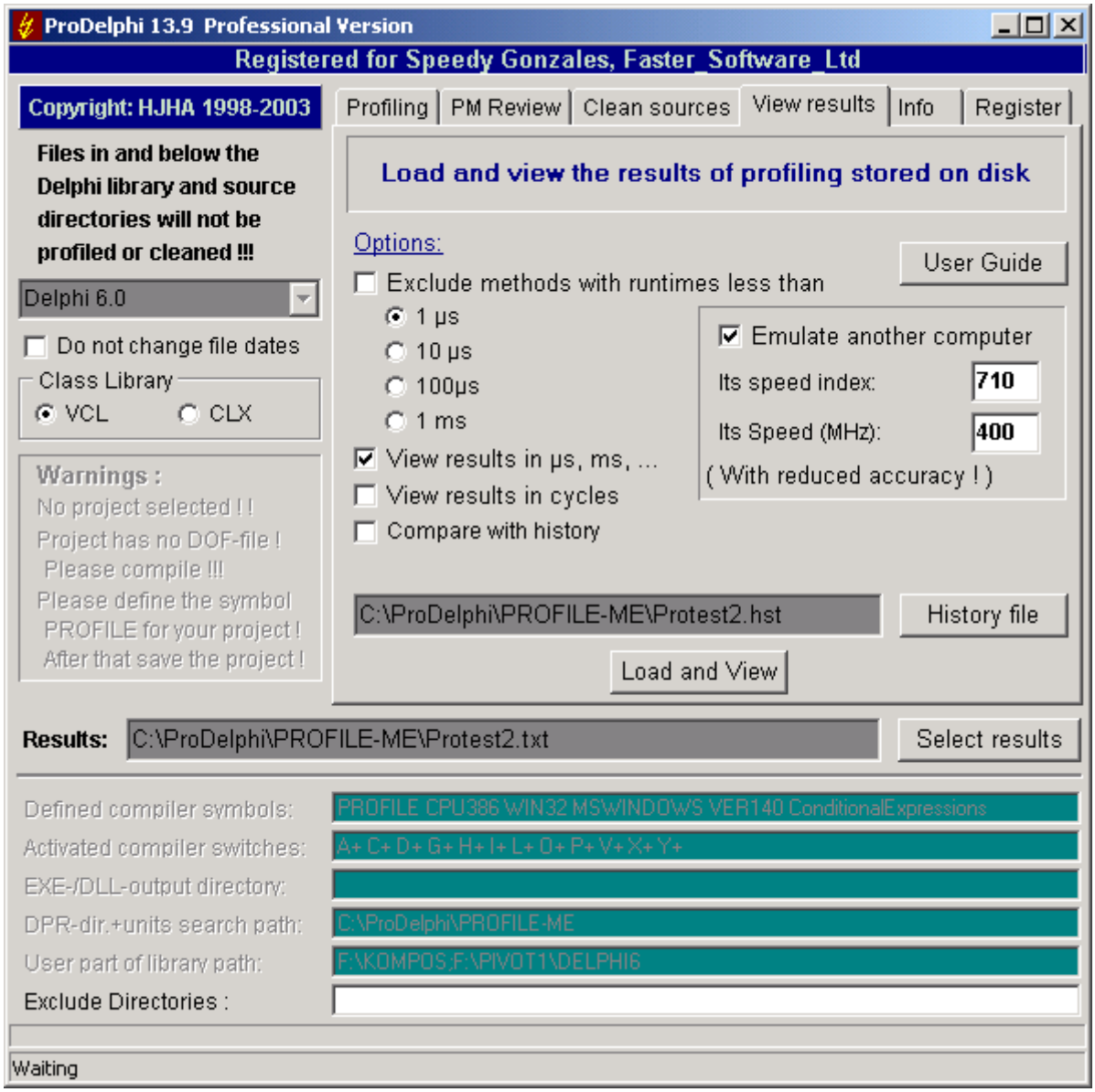
If you want to know, how fast (or slow) your program would perform on another PC, just use the program Getspeed.exe to get the other PC's speed index, enter it in ProDelphi, enter the speed in MHz of the other computer and start the viewer. Automatically all measurements are recalculated for the other 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 procedure that executes for a fixed time (e.g. for 1 sec), the emulation result for that procedure 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.

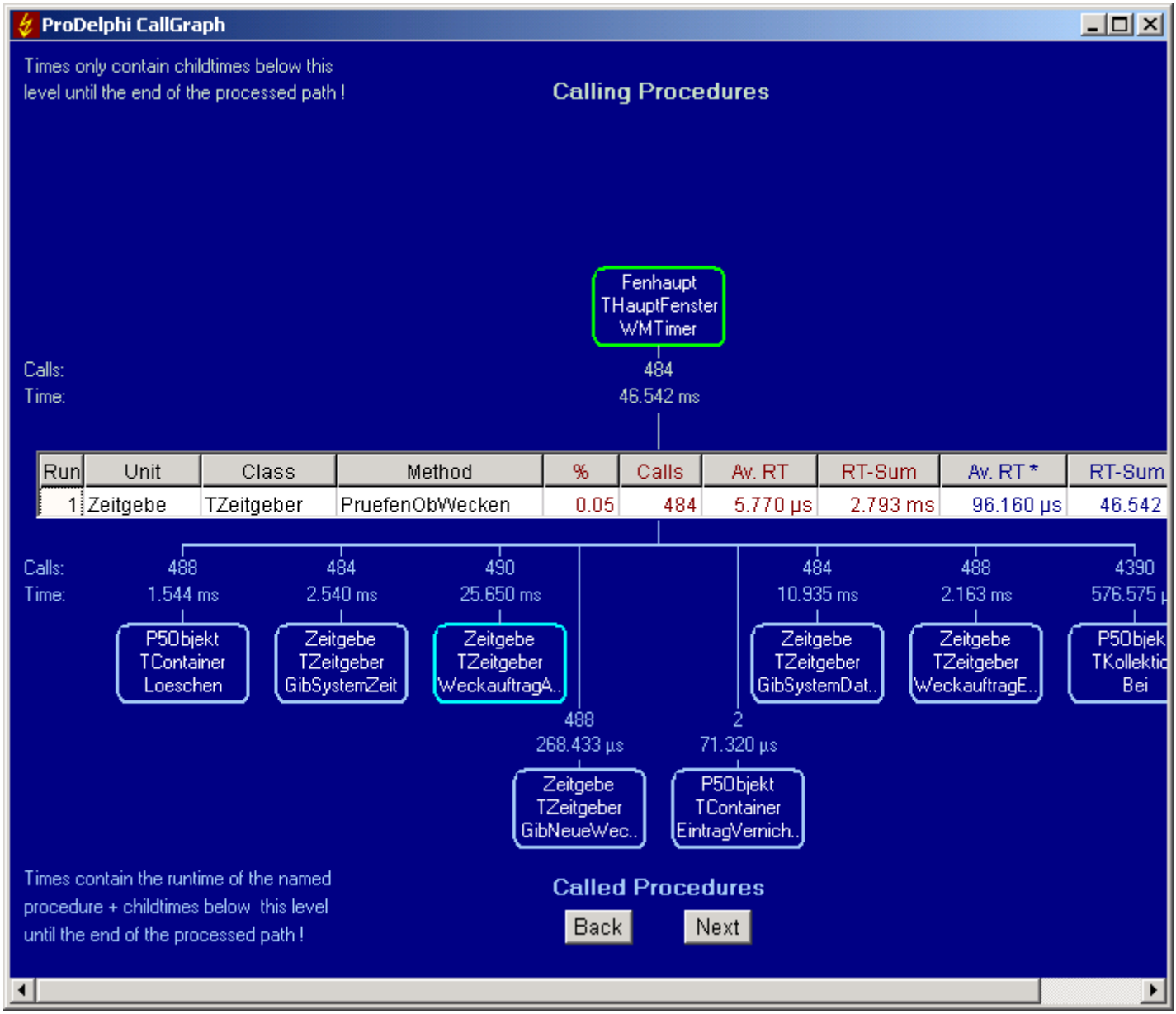

GetSpeed
Get speed index
This PC's speed index is:
710
Use this program to get the speed index
of the PC to be emulated with ProDelphi


ProDelphi 13.9 Professional Version
Registered for Speedy Gonzales, Faster_Software_Ltd
Copyright: HJHA 1998-2003
Profiling
PM Review
Clean sources
View results
Info
Register
Files in and below the Delphi library and source directories will not be profiled or cleaned !!!
Load and view the results of profiling stored on disk
Options:
User Guide
Delphi 6.0
Exclude methods with runtimes less than
1 µs
10 µs
100µs
1 ms
Emulate another computer
Its speed index:
710
Its Speed (MHz):
400
(With reduced accuracy !)
Do not change file dates
Class Library
VCL
CLX
View results in µs, ms, ...
View results in cycles
Compare with history
Warnings :
No project selected ! !
Project has no DOF-file !
Please compile !!!
Please define the symbol
PROFILE for your project !
After that save the project !
C:\ProDelphi\PROFILE-ME\Protest2.hst
History file
Load and View
Results:
C:\ProDelphi\PROFILE-ME\Protest2.txt
Select results
Defined compiler symbols:
PROFILE CPU386 WIN32 MSWINDOWS VER140 ConditionalExpressions
Activated compiler switches:
A+ C+ D+ G+ H+ I+ L+ O+ P+ V+ X+ Y+
EXE-/DLL-output directory:
DPR-dir.+units search path:
C:\ProDelphi\PROFILE-ME
User part of library path:
F:\KOMPOS;F:\PIVOT1\DELPHI6
Exclude Directories :
Waiting

A1.4 Using the caller / called graph (call graph)

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 procedure in a grid in the middle of the form. Above that up to 15 procedures are displayed that have called this procedure. If more procedures called the selected procedure, this is displayed at the right top of the form. Always those procedures are displayed, that consumed the most runtime. For each procedure the number of calls for the selected procedure and the runtime inclusive all child procedures is displayed.

Below the procedure shown in the grid, up to 15 procedures called by the selected procedure are displayed. Again here those procedures 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)**:



Left clicking on the symbol for a called or calling procedure makes this procedure appear in the grid with it's complete measurement results. Clicking on the procedure in the grid opens the concerned unit in the editor. The editor displays the procedure on top of the window. A yellow '**R**' on the left side of the grid in the middle of the window mean that the shown procedure was called recursively. Also a yellow procedure name in one of the ymbols has this meaning.

A2 Getting exact results

If you measure program runtimes a few times, you will see that the measurement results differ from measurement to measurement with out that you have changed your sources. Two kind of results will offen 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 procedures which perform disk transfer, the data can be transferred to disk or to disk cache.

Every profiler has this 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 some hints:

A2.1 Common causes of disturbing influences outside of 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 swapfile 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.

A2.2 Common causes of disturbing influences inside your program

Some disturbances you might have inside your measured program itself, these occure 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 everytime 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 A4, how to measure defined actions only is described in chapter A5.

A2.3 Common cause of disturbing influence is the PC's processor cache

The influence of the cache can't be easyly 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 eyery 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.

A2.4 Profiling 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 use 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 to 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 dispayed. It depends on the CPU speed that was set when the CPU speed was measured. Different processors use

different algorithyms to change the speed. The only way to get 100% correct results is to switch off the power safe mode.

A2.5 Summary

If you eliminate the disturbances mentioned in A2.1 / A2.2 and measure defined actions, you will see the differences between two measurements is very low, most times only a few CPU-cycles. Larger differences appear only when neasuring procedures 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.

A3 Interactive optimization

Interactive optimization means that you optimize something, check if it has brought you significant decreasement 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.

A3.1 The history function

The history function of the viewer enables you to compare your measurement results with a preceeding run. So you can see, if an optimization has brought an increasement or a decreasement 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 viewers 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 occured.

To get the cell colored, the methods change of runtime must be essential. Essential means, it must have changed so much, that it influenced the programs runtime by 1 % or more.

To display the runtime of a method from the stored history, just right-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 masurement on and off. I switched id 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 occure any more, I repeated this and compared the measurement results with the history function. When there 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.

A3.2 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 colored green now.
    If your method is slower now, it is colored red.

If there is no significant difference, it is colored 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 procedure stored in the history file. If '---' is displayed, the method is not present in the history file.

A4 Measuring parts of the program

A4.1 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 procedure 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 procedure.

What I described above, is the default setting of ProDelphi: all procedures 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 optmized (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. Procedures 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

   Enter the directories in the field ‘Excludedirectories’ 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 profiled) 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 profiling insert statements before and after the procedures that
   have to be excluded to switch off the vaccination by ProDelphi:

```
//PROFILE-NO              |
Excluded procedure(s)     |      These statements are not removed by ProDelphi.
//PROFILE-YES             |
```

8. Automatic exclusion

You can exclude procedures automatically by checking the option 'Deactivate functions consuming < 1 µs'. Checking this option means that those procedures, which are at least called 10 times during the measurement period and consume an average of less then 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 procedures which have to be deactivated. When you start your program next time the file will be read and all named procedures are deactivated. It might be that after the next run of your program again some lines will be appended with procedures to be deactivated.

The procedures 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.

A4.2 Dynamic activation of measurement

This is the best way of profiling. Normally one optimizes a certain function of a program, mostly that which takes too long. 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 dialog

In the main window of ProDelphi under the option ‘Activation of measurement’ select:
‘By entering a selected method’. After profiling 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 procedure 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.

A4.3 Finding points for dynamic activation

If you need to profile 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 procedures reacting on these events or messages?

To make it a little easier to find out this, all procedures thart start an action are entered in a list of starting points. Just perform a measurement run wich measures all procedures and starts the measurement automatically with the start of the application. After performing the action to profile, end the application, start the profiler and view the results. Under the last tab of the viewer all procedures are listed, that were not called by other measured procedures, 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 procedure to display the call graph for a procedure.

A4.4 Measuring specified parts of procedures

For the case of very large procedures sometimes it might be interesting to know which *part* of it consumed the most run time. One way to find this out is to restructure the procedure into neat parts or to devide it up by means of local procedures. 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 timecritical applications stop working.
For the case that both solutions given is too much work or to risky, 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 can 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
```

Profiling the sources after this causes ProDelphi to insert measurement statements right after the comments. The runtime measured in this so defined block will be found in the viewer because the comment is set behind the procedure name.

Using this feature is only possible when taking care to insert these statements so, 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 procedure but 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 procedure would be 8 ms (displayed in the line for procedure DoSomething), run time inclusive child time would be 18 ms.
In the line for procedure 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 in a different way, especially processors with a small cache have the problem, that not the whole procedure inclusive measurement parts of ProDelphi fit into the cache, so additional wait states occure.

Remark:

It is possible to define more than one measurement block in a procedure 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 examle the runtime for part b is displayed separatly AND also included as child time of part a (and, of course, also in the child time of DoSomething).

A5 Programming API

A5.1 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 profiling of your source code and insert activation statements at the relevant places.

Example1:

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

```
PROCEDURE TForm1.ButtonClick;
BEGIN
{$IFDEF PROFILE}asm...end; Try; asm... call ProfInt.ProfEnter;...end; {$ENDIF}
   SortAll;  // the procedure of which you want to know the runtime
{$IFDEF PROFILE}finally; asm...; mov cx,number; call ProfExit; end; end; {$ENDIF}
END;
                                    // @self if used inside classes otherwise NIL
```

You can modify the code in three different ways:

```
{ possibillity 1 }
PROCEDURE TForm1.ButtonClick;
BEGIN
{$IFDEF PROFILE}asm...end; Try; asm... call ProfInt.ProfEnter;...end; {$ENDIF}
{$IFDEF      PROFILE}try; ProfInt.ProfActivate;{$ENDIF}
   SortAll;  // the procedure which you want to know the runtime of
{$IFDEF      PROFILE}finally; ProfInt.ProfDeactivate; end; {$ENDIF}
{$IFDEF PROFILE}finally; asm...; mov cx,number; call ProfExit; end; end; {$ENDIF}
END;

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

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

You should use possibillity 1 or 3 because a new profiling does not change your code, Possibillity 2 is changed by the next profiling 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 vaccinated by ProDelphi. Alternatively you also can use lower case letters.

Example 2:

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 procedure call is measured until activation.

A5.2 Preventing to measure idle times

Some Windows-API functions and Delphi functions interrupt the calling procedure and set the program into an idle mode. A well-known example is the Windows-call MessageBox. This call returns to the calling procedure after the a button click. Between call and return to the calling procedure, 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 redefiniton. It automatically interrupts the counting of CPU-cycles for the calling procedure only and reactivates it after returning from windows.

If other procedures 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 A9 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;

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

Important:

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

A5.3 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 chapter A5 also).

Important:

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

A6 Options for profiling

Profiling options are divided into three groups:

- Code instrumenting options (or vaccination options): How and what to vaccinate.
- Runtime measurement options: How to measure and what to do with the results.
- Activation of measurement: Where or when to start measuring runtimes.
- General options: Which Delphi version / file date.

A6.1 Code instrumenting options:

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

Profile Assembler procedures (Professional Mode only)

Assembler code is normally not profiled (often assembler is a result of an optimization process already). In the professional mode 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 INITIALZATION and FINALIZATION in your units.

Profile local procedures

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

Profile Library path (Professional Mode only)

Normally only the files belonging to a project are profiled. These files are all the files in the unit search path of the actual project. Files which are belonging to components e.g. which are linked to the program by the linker are not profiled. 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, carefullness is necessary. If you include these files, the sources are vaccinated with profiler statements. If you, after profiling one project, change to another project, the files are still vaccinated. 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, the need to be excluded in all projects, otherwise in the results you might find undefined procedures.

Process write protected files

checking this option means, that all write protections for your source files are deleted and the files are profiled. 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 a DLL or a program + the used DLL('s). See chapter 8 for details.

A6.2 Runtime measurement options

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

Count Inherited calls for parent

This option is only valid for methods (procedures and functions belonging to objects or classes).

Normally times are measured separate for each procedure. Use this method if you want, that, if a method calls 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 procedures that are called at least ten times and consume less then 1 µS are deactivated for the future. The deactivated functions are stored in the file 'ProgramName.SWO' for the next run.

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.

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 absolutely 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 runtimes (Professional Mode only)

If this option is checked, the measurement routines of ProDelphi additionally estimates minimum and maximum runtimes of every procedure. Normally only average times ar estimated. Minimum an 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.

A6.3 Measurement activation options

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

At program start (default)

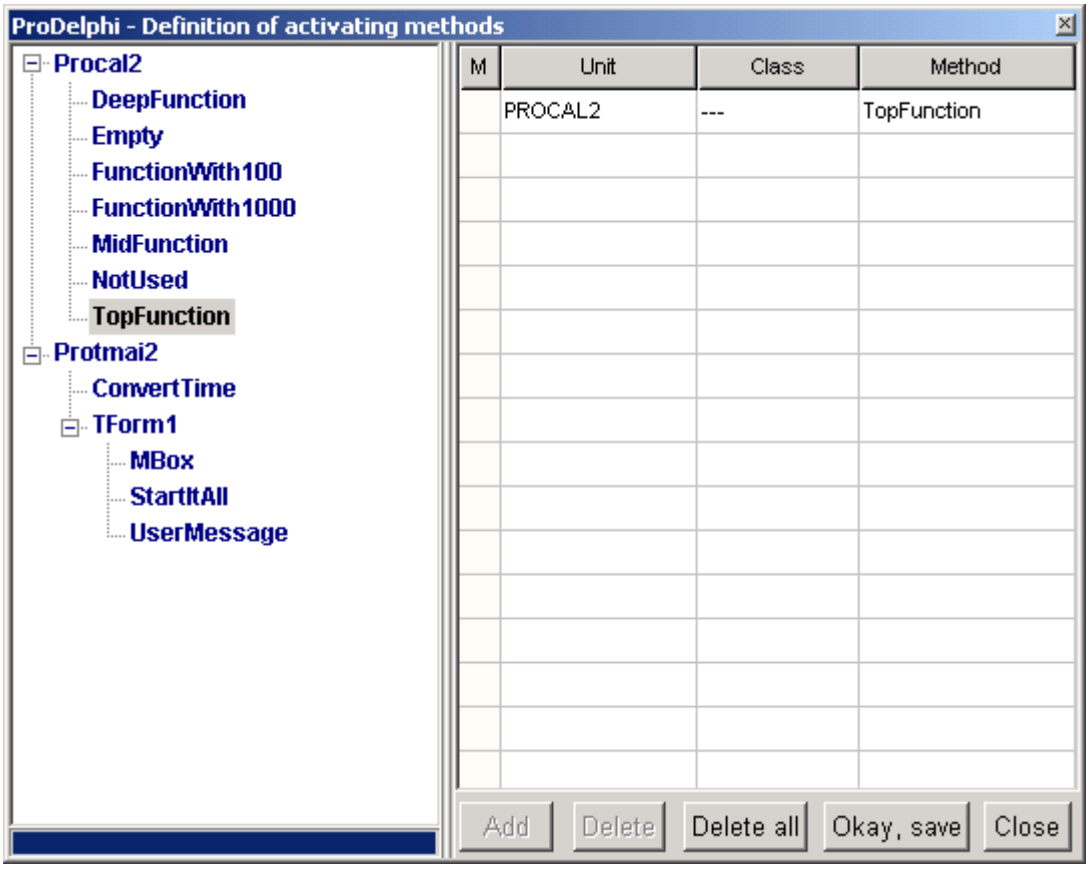
If this option is checked, the time measurement will start as soon as your program is started. In that case the 'Start'-button in the online operation window is disabled and the stop button is enabled. If the option is not checked the 'Start'-Button is enabled and the 'Stop'-button is disabled.

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 4.2). If you use this option, you should not use the Online-operation window.

Example:

see next page, please.



By API-Calls or online operation window

(see chapter A5.1 and A7 for details)

A6.4 General options

Delphi version

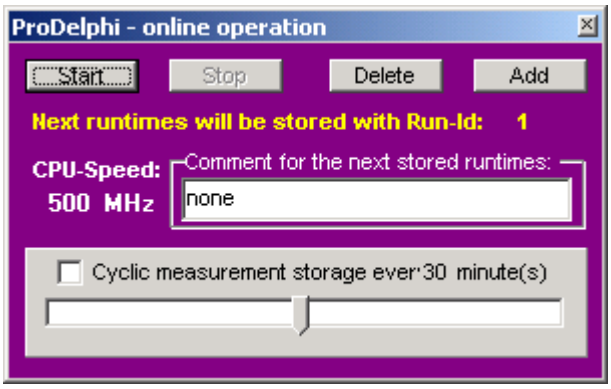
You should check the version to that Delphi verion you are going to compile the program with. This assures that ProDelphi uses the correct compiler switches.

File Date

The check box ‘Do not change file dates’ is available in professional mode only. Checking this results in changing the file dates by 5 seconds when profiling, enough to make Delphi realize that a file has changed. Unchecking means that the actual date and time is used.
When cleaning the surces the filedate is decreased by 5 sec. This makes possile that the file date keeps the same between checking out and in from a source code control system.

A7 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 results in the ASCII-File. 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 measurment 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 !

A8 Dynamic Link Librarys (DLL's) and packages

A8.1 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 profiled 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 problemless 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. Also the directory for storing EXE- and DLL-file have to be identical. ***ProDelphi reads the search path and the compiler switches from the DOF-file of the selected project. No matter which of the projects is profiled, you always have the profiling information and the measurement results in the correct directory and all necessary code is profiled.***

3. 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 an additional manual step is necessary:

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

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

A8.1 Packages

Profiling designtime 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 procedures in a package one needs to profile the program that uses the package. In order to profile the units belonging to the package, all PAS- and INC-files of the package need to be stored in directories that are named in the search path of the using program. Until here it is quite similar like profiling a program and DLL's simultaneously. The big difference comes into the game whith the DPK-files.

DPK-files are not evaluated. In the CONTAINS section of the DPK-file PAS-files belonging to the package can be named. If these files are named together with a path, a problem could occur. When these pathes are not in the units search path of the program, they won't be profiled. (Units named in DPR-files are profiled even the when they are not stored in a directory of a search path if their path is explicitly named in the uses section).

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. Profile the program. This then includes to profile 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.

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-profile again, step 4 and 5 also have to be executed again.
Any time now you change the package by inserting or deleting functions and re-profile again, step 6 also has to be executed again.

A9 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 procedure is not counted, but other procedures 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 Profint-unit 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.

A9.1 Redefined Windows-API functions

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

A9.2 Redefined Delphi-API functions

- ShowMessage,
- ShowMessageFmt and
- MessageDlg.

A9.3 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.

A10 Conditional compilation

A10.1 Delphi 2, 3, 4 and 5

Conditional compilation is fully supported.

A10.2 Delphi 6 and above

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

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

The directives $IF, $IF, $ELSEIF, $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 = 4;
{$IF xxx > 5 }
  PROCEDURE AddIt(VAR first, second, sum : Int64);
  BEGIN
{$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 = 4;
{$IF xxx > 5 }
  PROCEDURE AddIt(VAR first, second, sum : Int64);
{$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;
```

A11 Limitations of use

Console applications have no online operation window.

Procedures in a DPR-file can not be measured.

The measured times are always differ about +-5 % (max) from those of an unprofiled program. The reason is that the program code is not so often replaced in the cache than without measuring.

For the purpose of vaccinating 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, a user stack is used by the profiler unit. The maximum stack depth is 16000 calls.

In the freeware mode of ProDelphi only 20 procedures can be measured, in the professional mode 32000.

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 let the same routine again and again and each time I've got slightly differing results.

Don't forget the influence of the processor cache also. 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 procedure needs some CPU-cycles for getting the code into the cache. **The larger the cache size, the better the results ! The profiling procedures use the cache too !**

Then there is the CPU itself. The modern CPU's like Intels Pentium or AMD's K6 are able to execute instructions parallel. When the profiler inserts instruction, the parallelity 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 procedure entry.

Be aware that, if you measure procedures 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.

A12 Assembler Code

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

If it is absolutely necessary to measure such procedures in the Freeware mode, just put an additional BEGIN before the asm statement and an additional END after the last statement (e.g. FUNCTION Assi : Integer; BEGIN asm mov eax,2; end; END;)

A13 Modifying code vaccinated by ProDelphi

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

A14 Hidden performance losses / Tips for optimization

ProDelphi measures runtimes of procedure bodies. This means that the entry part of a procedure which e.g. writes variables to the stack, is measured in the calling procedure! 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 anymore. Anyway, changing of the number of parameters of a procedure changes always the runtime of the calling procedure (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;          // increasement 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;
```

A15 Error messages

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

Vaccinating a file is done in this way:

- the original file *.pas is renamed into *.pay (or *.dpr into *.dpy and *.inc into *.iny),

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

- 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.

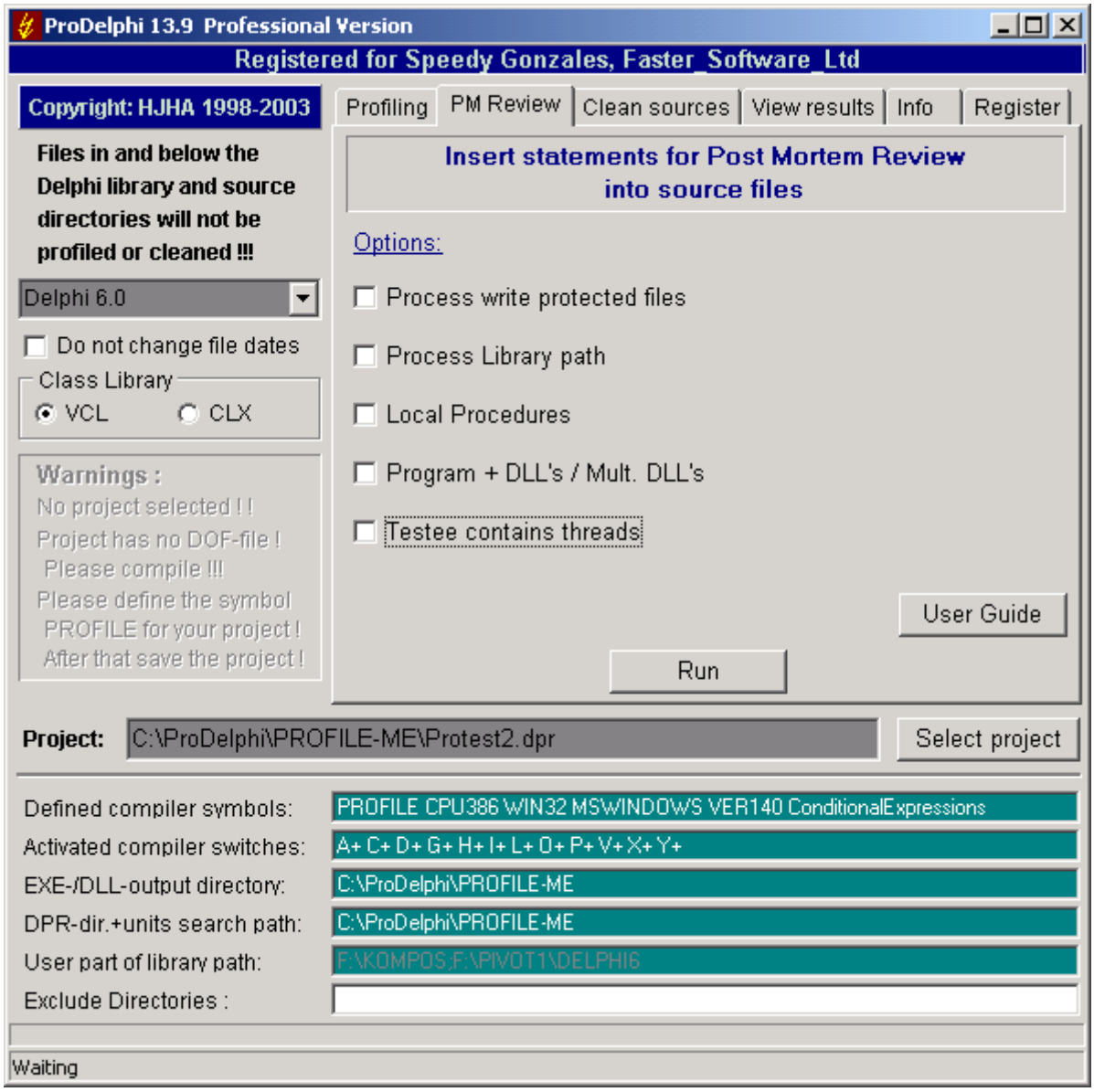
A16 Security aspects

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

- ProDelphi checks, if you have enough space on disk to store a profiled file before profiling it. ProDelphi assumes that the output file uses 3 times the space of the original file (normally it uses less). If there is not sufficient space, it will stop profiling.

B Post mortem review

As mentioned above, ProDelphi can vaccinate your sources with statements for post mortem review. It also interpretes the sources and inserts statements at the begin and at the end of a procedure.



In case of an aborting 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.

For the availlable options see chapter A4.

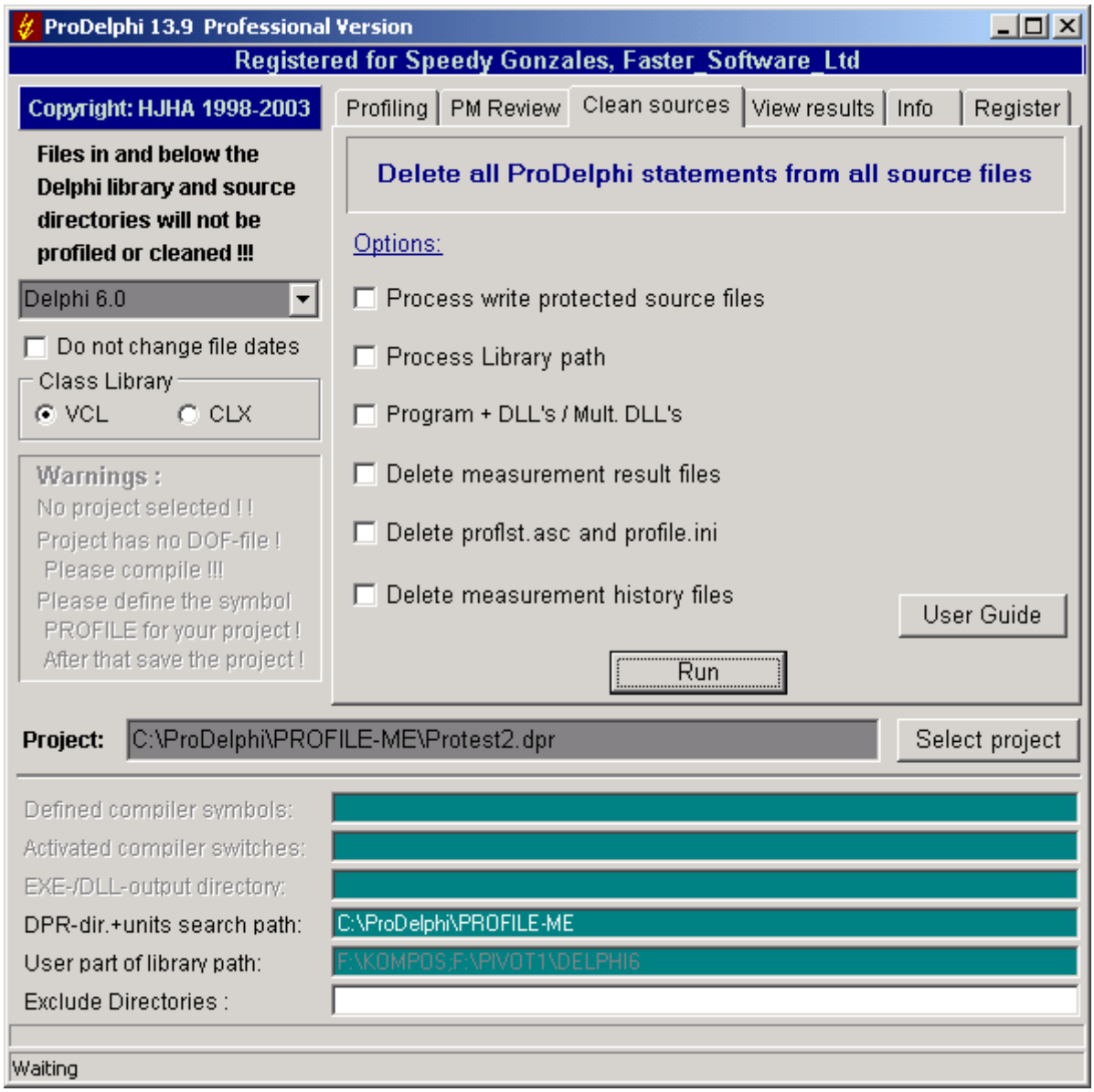
The handling of Prodelphi is the same as for profiling. You also have to define the compiler symbol PROFILE:

If you have vaccinated 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.

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 'Profile' command another time. Each profiling 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 profiling by 5 sec and decreased at cleaning by 5 sec. This makes possile that the file date keeps the same between checking out and in from a source code control system.

D Compatibility

ProDelphi was testet under
- Windows 95, 98, Windows NT 4.0, Windows 2000 and Windows XP.
- AMD K6 166 / 233 MHz, AMD K6-2 266 / 300 / 500 MHz, AMD K6-3 400 MHz, AMD Athlon 1800 MHz,
- AMD Duron 1100 MHz
- Pentium Overdrive 120 MHz, Pentium II / 400 MHz, Pentium III 750 MHz, Pentium Celeron 400 MHz.

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 Windows system directory and all needed units into the Delphi-LIB-directory and 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 file (for data base export and viewer)

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

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

> run; unitname; classname; procedurename; % of RT; calls; minimum RT excl. child; average RT excl. child; maximum RT excl. child; RT-sum excl. child; minimum RT incl. child; average RT incl. child; maximum RT incl. child; RT-sum incl. child; % incl. child

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 can be loaded via authors home page. Every new release will automatically be stored there. Just click on 'Additional information' to see which version is actual.

H How to order the registration key for unlocking the professional mode

Customers who who want to use the professional mode, can order a registration key to unlock the Professional mode. Just start the program ProDelphi (Profiler.exe), select the page for registration and enter the information you have got by e-mail. At the next start of ProDelphi, the Professional mode is unlocked. This key is also valid for upgrading following versions. If a bugfix is made or an upgrade is done, it will be stored on my homepage. Just download from there and you can continue to use the new program in Professional mode. The registration key is stored in the file ‘profiler.rky’.

Customers who ordered the registration key can have a link to their company in my customers reference list, just send me an e-mail.

The key to unlock the professional mode can be ordered by ShareIt shareware registration service (see the files REGISTER.ENG and REGISTER.GER).

I Author

Helmuth J. H. Adolph (Dipl. Inform)
Am Gruener Park 17
90766 Fuerth
Germany

E-Mail: hjh.adolph@prodelphi.de
Home page: http://www.prodelphi.de
http://www.windkraft.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	Profiling 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	Profiling 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 dialog and by special comments in the source files, emulation of other PC's, main form arranged nicer, profiling 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 color or in color 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

	profiling or cleaning. Overloaded functions were partly not recognized, same with some class definitions (this occured 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 a profiling error, Improvement: Cyclic automatic storage of measurement results
Version 10.0:05/02	Evaluation of minimum and maximum runtimes, profiling of the library path, measuring of specified parts of procedures, 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. Profiling of files named in the USES-statement in the DPR-file of a program added. Simultaneous measuring of a program and DLL's or mulltiple 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 profiled 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 nonger 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 colums 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 dialog bugfixed.
Version 12.1 03/03	Correction of algorithym 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 emuluate 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. Inprovement: 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 ecxeption. Bugfix: Check for profiling more than 32000 methods was missing, which caused partly wrong measurement results when this case occured.
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 profiled if Uses was in the same line with implementation statement Feature: Number of activating procedures 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 profiling by 5 sec and decreased at cleaning by 5 sec. This makes possile 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 procedures (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 profiling made easier and more safe. Trying now to analyze if the application contains threads and displaying a warning if so and the option ‘Testee contains threads’ is not checked.

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).