

# ProDelphi User Guide

( Release 9.2 )

Copyright Helmuth J. H. Adolph 1998/99/2000/2001

## The Profiler for Delphi 2, 3, 4, 5 and 6 (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 whoever) 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 Turbo 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 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.

### 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 30 procedures can be measured or tracked, in the professional mode 32000.

In the professional mode additionally assembler procedures can be measured and tracked.

**Date: 11/25/2001**

## 0. Contents of this description

- A. Principle of Profiling
  - A1. How to profile
    - A1.1 Files created by ProDelphi or the measured program
    - A1.2 Checking the results with the Built in viewer
    - A1.3 Emulation of a faster or slower PC
    - A1.4 Checking the results by viewing the ASCII-file
  - A2 Getting exact results
    - A2.1 Common causes of disturbing influences outside of your program
    - A2.2 Common causes of disturbing influences inside your program
    - A2.3 Common cause of disturbing influence is the PC's cache
    - A2.4 Summary
  - A3 Interactive optimization
    - A3.1 The history function
    - A3.2 Practical use of the history function
  - A4 Measuring only parts of the program
    - A4.1 Exclusion of Parts of the program
    - A4.2 Dynamic activation of measurement
  - A5 Programming API
    - A5.1 Measuring defined program actions through Activation and Deactivation
    - A5.2 Preventing to measure idle times
    - A5.3 Programmed storing of measurement results
  - A6. Options for profiling
    - A6.1 Code instrumenting options:
    - A6.2 Runtime measurement options
    - A6.3 Measurement activation options
  - A7. Online operation of the profiled program
  - A8. Profiling dynamic link libraries (DLL)
  - A9 Treatment of special Windows- and Delphi-API-functions
    - A9.1 Redefined Windows-API functions
    - A9.2 Redefined Delphi-API functions
    - A9.3 Replaced Delphi-API functions
    - A9.4 Not replaced or redefined Delphi functions
  - A10 Conditional compilation
    - A10.1 Delphi 2 .. 5
    - A10.2 Delphi 6
  - A11. Limitations of use
  - A12. Assembler code
  - A13. Modifying code vaccinated by ProDelphi
  - A14. Error messages

A15 Security aspects

Appendices:

- B. Post mortem review
- C. Cleaning the sources
- D. Compatibility
- E. Installation of ProDelphi
- F. Description of the result file (data base export)
- G. Updating / Upgrading ProDelphi
- H. How to order the registration key for unlocking the Professional mode
- I. Author
- J. History
- K. Literature

## **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 elapsed 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.nev) contains the names of all methods which have never been called when measuring the runtime of your program. It is used by 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 **fourth** file is optional and contains the measurement results in a format, that can be printed in landscape format or can be viewed by Delphi. It is described in chapter A1.3.

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 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. If you want to install ProDelphi manually, you need to perform the following steps:**

*Copy the files ProfCali.DLL and ProfMeas.DLL into the WINDOWS\SYSTEM - directory (for Windows 95/98) or into the WINDOWS\SYSTEM32 - directory (for Windows NT). Copy the files ProfInt.PAS, ProfOnli.PAS, ProfOnli.DFM into into the Delphi LIB-directory. The DFM-files are distributed as DF2-, DF3-, DF4- and DF5- files for Delphi 2/3/4/5. You need to rename the correct file.*

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 DLL, compile the DLL.*

***If you want to measure procedures in the program and in DLL's simultaneously, program and DLL's must have exactly the same units source path and the DPR-files must be in the same directory, in that case compile both: program and DLL's.***

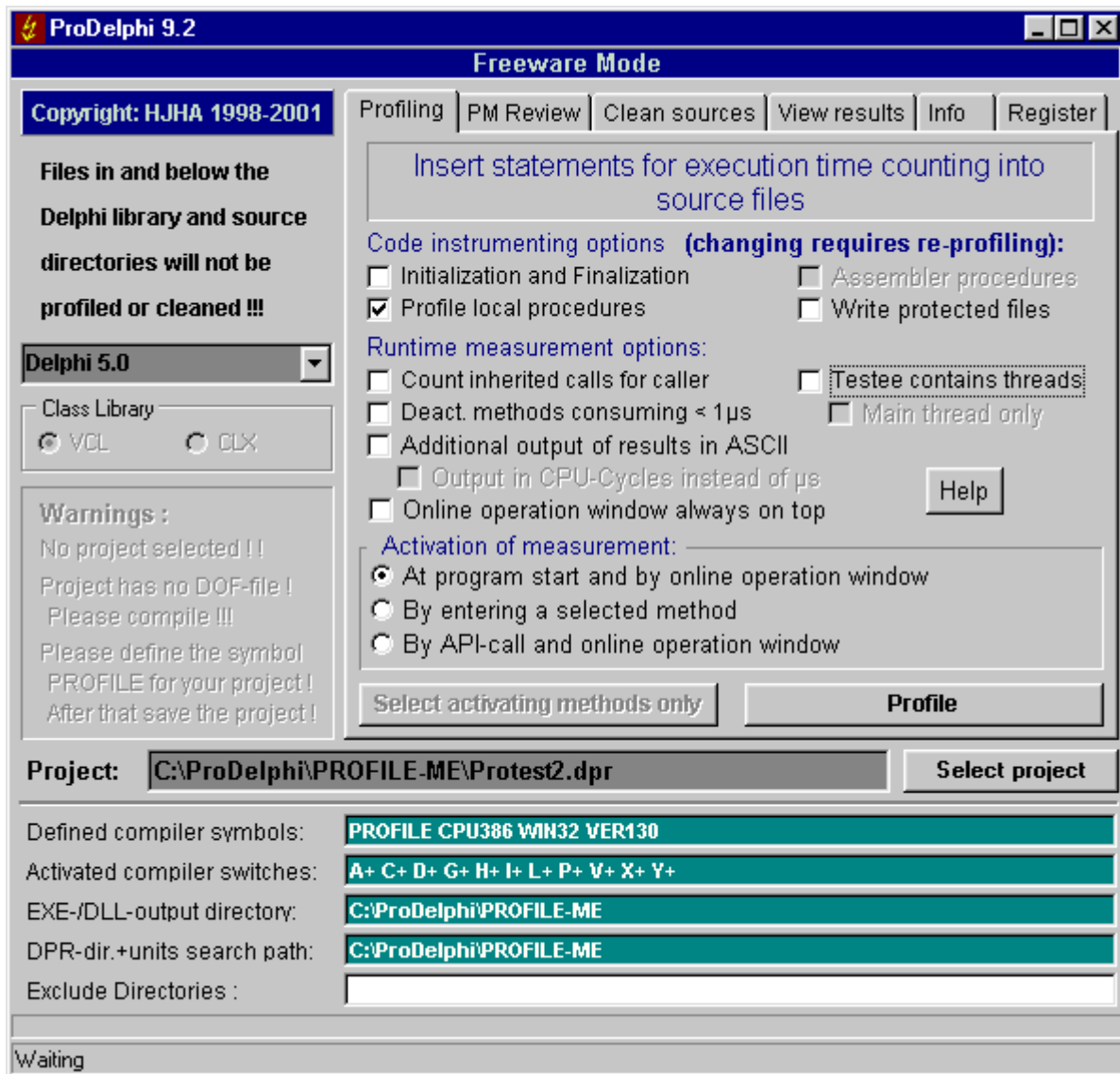
If no compilation errors occur, you may profile your program (and/or DLL).

**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).

**See next page for the ProDelphi window, please.**



- Select the kind of activation for 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 .
- Recompile the program (or DLL's).

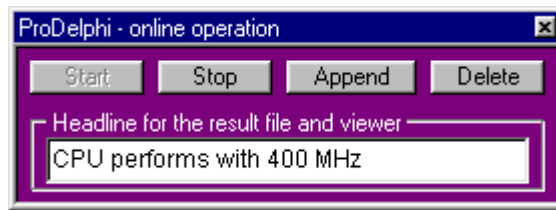
**To allow simultaneous measuring of DLL's and programs, all files in the units search path are profiled!!! (unless they are write protected !!!). The unit search path must be exactly the same for program and DLL, both DPR-files have to be in the same directory !!!**

**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 A5 for the complete description. After the program has ended, you can view the results of the measurement

**by the built-in viewer of ProDelphi,**

or if you have checked the option for ASCII-output, you can

**load the file 'program-name.ben' into Delphi**

and maybe print it with Delphi.

For the Built-in viewer, just start ProDelphi again, go to the view page. If the name of your project is not automatically displayed, select it. Then click the 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 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 will create the file 'progrname.hst' if you use the history function (see A3).

Your compiled program creates 'progrname.ben', it contains the results of the time measurement in a printable format if you have checked 'Additional output in ASCII' when profiling. The file with the name 'progrname.txt' contains the data in the ASCII-semicolon-delimited format for data base export and 'progrname.tx2' for the headlines for the different runs (for the built-in viewer). It creates 'progrname.swo' with the list of procedures that have to be deactivated for time measurement at next program start. It creates also a file with the name 'progrname.nev' 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 'progrname.pmr' in case you have selected post mortem review and an exception occurred and was trapped. It contents the call stack.

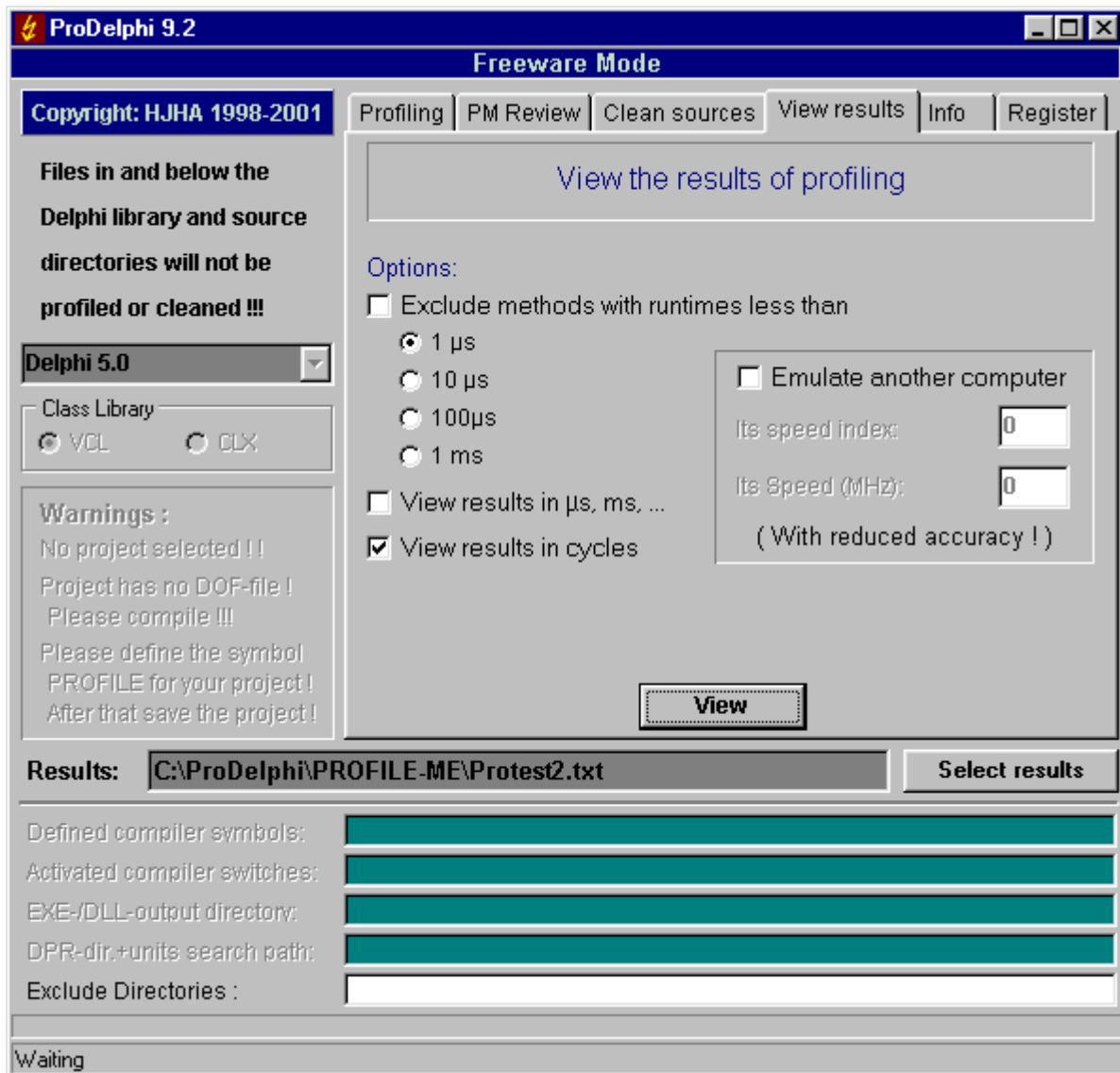
All files are stored in the output directory for the \*.exe (\*.dll) file.

**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 !!!**

## 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.**



**ProDelphi - Viewer**

Details | Max RT-T | Max RT-G | Max Calls-T | Max Calls-G | Max RT incl child-T | Max RT incl child-G | RT-Classes

Run	Unit	Class	Method	%	Calls	RT	RT-Sum	RT	RT-Sum	%
1	Procal2	---	DeepFunction	0.39	20,000	1	20,000	1	20,000	0.39
1	Procal2	---	Empty	0.00	200	0	0	0	0	0.00
1	Procal2	---	FunctionWith100	2.38	200	609	121,857	609	121,857	2.38
1	Procal2	---	FunctionWith1000	23.48	200	6,012	1,202,323	6,012	1,202,323	23.48
1	Procal2	---	MidFunction	4.93	2,000	126	252,412	136	272,412	5.32
1	Procal2	---	TopFunction	0.63	200	162	32,315	1,524	304,727	5.95
1	Protmai2	---	ConvertTime	0.42	20	1,077	21,536	1,077	21,536	0.42
1	Protmai2	TForm1	MBox	0.16	1	8,143	8,143	8,143	8,143	0.16
1	Protmai2	TForm1	StartItAll	61.09	1	3,127,727	3,127,727	4,786,313	4,786,313	93.49
1	Protmai2	TForm1	UserMessage	6.51	1	333,371	333,371	333,371	333,371	6.51

Browser | Sort Table | Search Unit | Search Class | Search Method

Red: Runtime (RT) or calls of the nan  
Blue: Runtime contents also called c  
Method uses more than

This page shows the runtime of all methods, first without and second with childtimes

Comment: At finishing application

<< >> Save as History Not called Methods CPU: 400 MHz / Total RT: 5,119,684 CPU-cycles

Alphabetically sorted results, first Units, second classes and third 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)

### The Sort Table - button:

The displayed table can be sorted after different criteria, just try it! You can also click on the headline of the columns to get the table sorted. E.g. clicking on calls makes the whole table be sorted after the number of calls.

### The Print - button:

Prints the actually displayed table or table + graphic. The table is automatically adjusted so that it fits on the paper.

**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
RT	Runtime of the procedure in CPU-cycles or in µs, ms, sec or hour units
RT-sum	RT * Calls

Meaning of the **BLUE** columns:

RT	Runtime of the procedure inclusive its child procedures in CPU-cycles or in µs, ms, ...
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 available 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

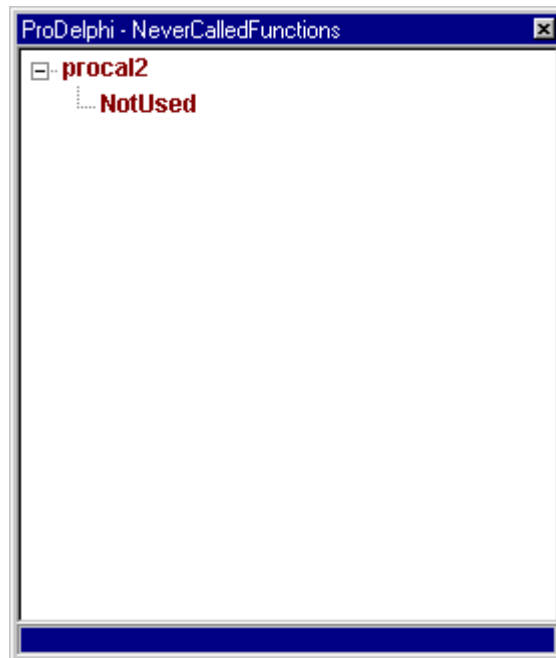
The 12 sorted lasses that consumed the most runtime

The 12 sorted units that consumed the most runtime

**The Not called Methods - button:**

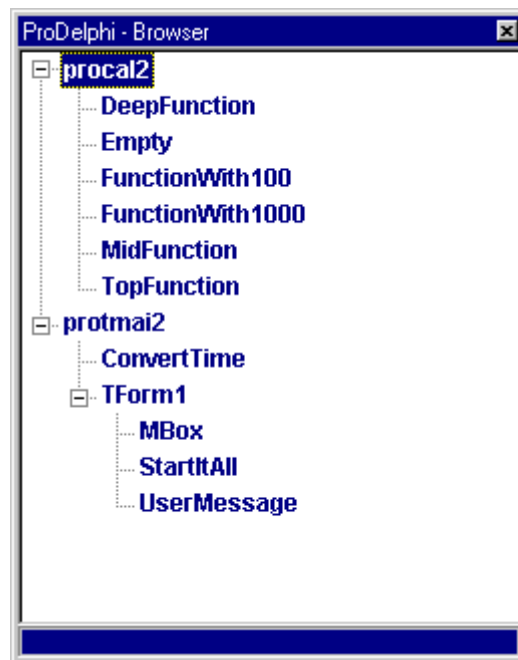
At the end of runtime the testee creates a file with the names of all uncalled methods. Using this button, these methods are displayed in hierarchical order: Unit - Class - Method.

**See next page, please.**

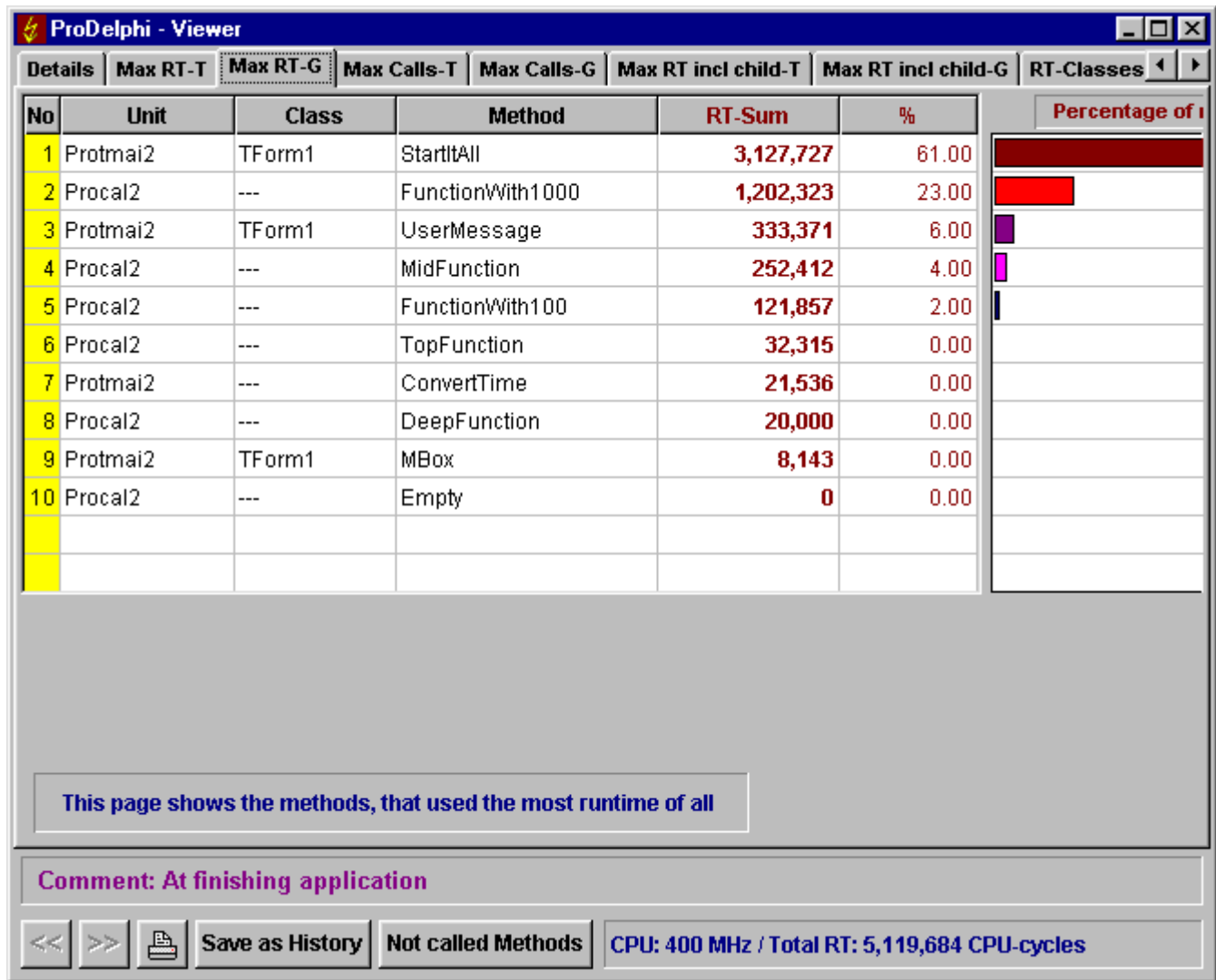


### The Browse - button:

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)

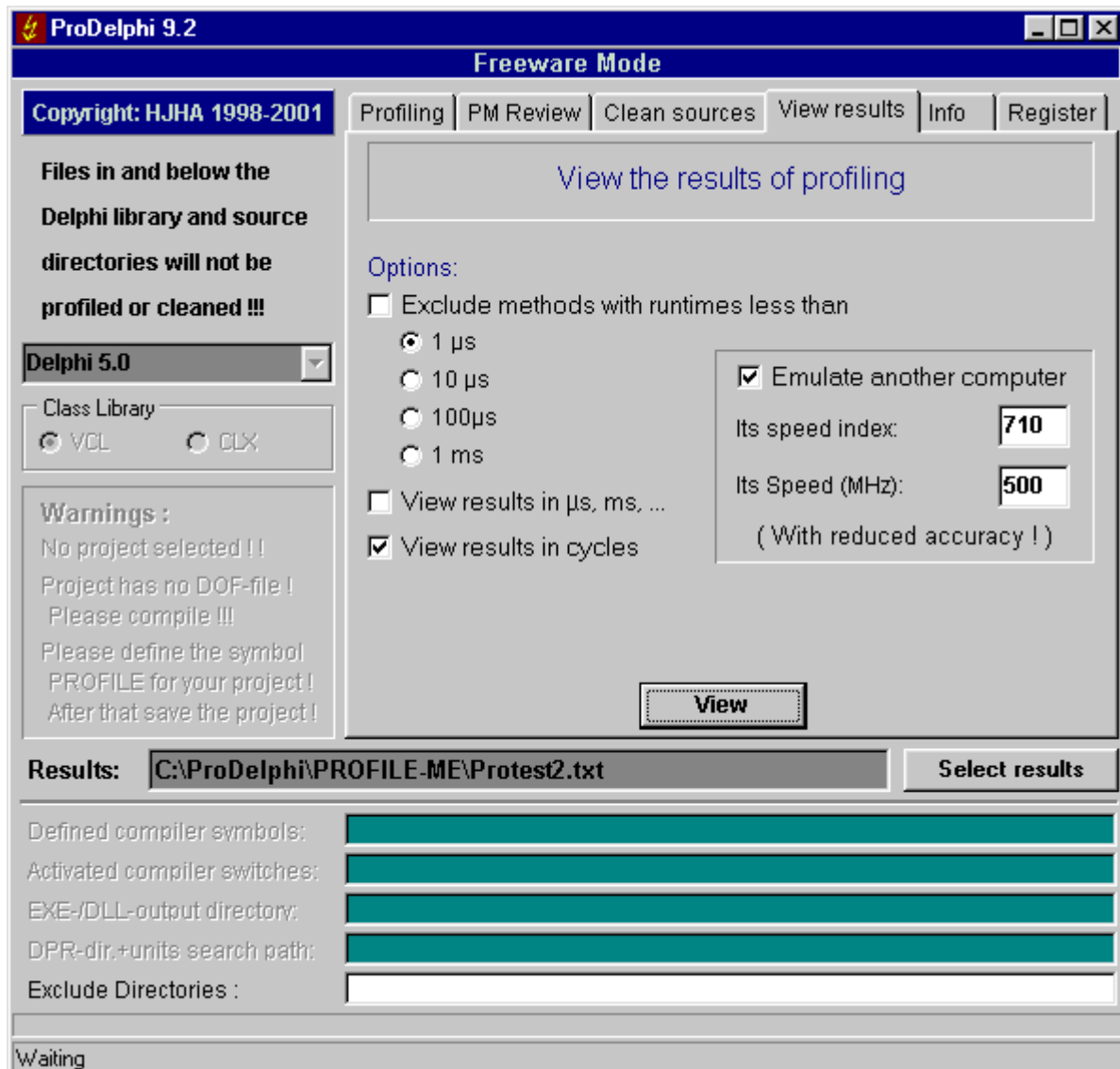
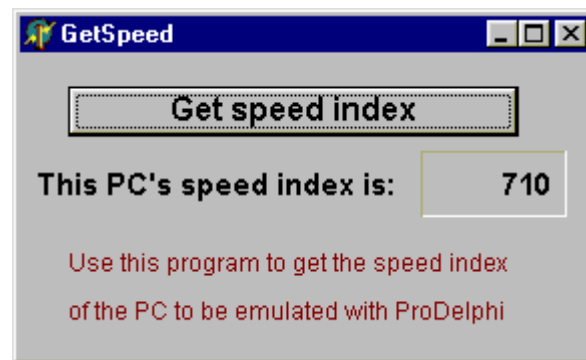
### A1.3 Emulation of a faster or slower PC

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).

See next page for example, please.



## A1.4 Checking the results with the optional ASCII-file

IF you have checked the option 'Additional output of results in ASCII', a file with the name 'programname.ben' is created. For each procedure one record is stored in that file. Below the content of this file is described:

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

### **a. Runtime of the procedure bodys exclusive called child procedures**

- runtime of the procedure in percentage of all measured procedures
- number of calls
- runtime in  $\mu$ s, ms, sec, min, h (or alternatively in CPU-Cycles) given for a single call
- runtime sum ( = runtime \* number of calls )

### **b. Runtime of the procedure bodys inclusive called child procedures**

- runtime in  $\mu$ s, ms, sec, min, h (or alternatively in CPU-Cycles) given for a single call
- runtime sum ( = runtime \* number of calls )

### **c. Summary**

- the most often called procedures
- the most CPU-time consuming procedures

### **d. The class wich consumed the most runtime (= the sum of all method runtimes)**

### **e. The total runtime of the tested program (= the sum of all measured procedures)**

For a quick test, points a. and b. can be disabled.

For every procedure such line is given in the result file:

classname-MethodName	consumed time as described above	or
ProcedureName	consumed time as described above.	

The sorting order of the listing is Unit-alphabetical, inside the units the order depends on the order of the procedures.

***All times given are exclusive the time used for measurement !!!***

## 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 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 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 time,
- 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 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.

### A2.4 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 measuring 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 decrease 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 10 % to decrease the total program runtime by 1 % !!!***

There are different ways of comparing the measurement results:

- to use the ASCII-output and print it,
- to use the viewer and make screen dumps 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 ONE preceding run. So you can see, if an optimization has brought an increase or a decrease of runtimes.

Having made a measurement, you can decide, if you want to store the results being displayed in the viewer's table on disk or not.

If you have stored the results on disk, the next time you open the viewer window, automatically the history stored before is read and compared with the actual measurement. By coloring the cells of the viewer's table, you have a quick overview about all changes of runtime: Red means method got slower, green means method got faster and white means that no essential change occurred.

To get the cell colored, 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.

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.

### A3.2 Practical use of the history function

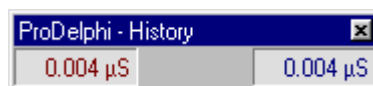
- Make a measurement for the defined action you want to optimize.
- Load the results into the viewer.
- Click on the history button to store these results into the history file.
- Optimize a method that is worth to be optimized.
- Repeat your measurement.
- Load the new data into memory.

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 only 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 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. 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:

<code>//PROFILE-NO</code>		
<code>Excluded procedure(s)</code>		<b>These statements are not removed by ProDelphi.</b>
<code>//PROFILE-YES</code>		

8. Automatic exclusion

You can exclude procedures automatically by checking the option 'Deactivate functions consuming < 1  $\mu$ s'. Checking this option means that those procedures, which are at least called 10 times during the measurement period and consume an average of less than 1  $\mu$ s 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.

## 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 algorithm 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:

```
{ possibility 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;

{ possibility 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;

{ possibility 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 possibility 1 or 3 because a new profiling does not change your code, Possibility 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 redefinition. 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 } Proftime.ProfAppendResults; { $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 } Proftime.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 the results.
- Activation of measurement: Where or when to start measuring runtimes.

### A6.1 Code instrumenting options:

*Changing these options after profiling DO afford a new profiling !!!*

#### ***Profile Assembler procedures***

Assembler code is normally not profiled (often assembler is a result of an optimization process). In the professional mode this feature can be enabled.

#### ***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.

### ***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.

## **A6.2 Runtime measurement options**

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

### ***Additional output of results in ASCII (printable or viewable with Delphi)***

with the sub-option:

- Output in CPU-Cycles instead of  $\mu$ Seconds

This option enables the creation of a printable file. You can open this file with Delphi and print it (landscape format should be preferred).

### ***Deactivate functions consuming < 1 $\mu$ S***

Any time the measurement results are stored in the result file, those procedures that are called at least ten times and consume less than 1  $\mu$ S are deactivated for the future. The deactivated functions are stored in the file 'ProgramName.SWO' for the next run.

### ***Inherited 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.

### ***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.

## **A6.3 Measurement activation options**

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

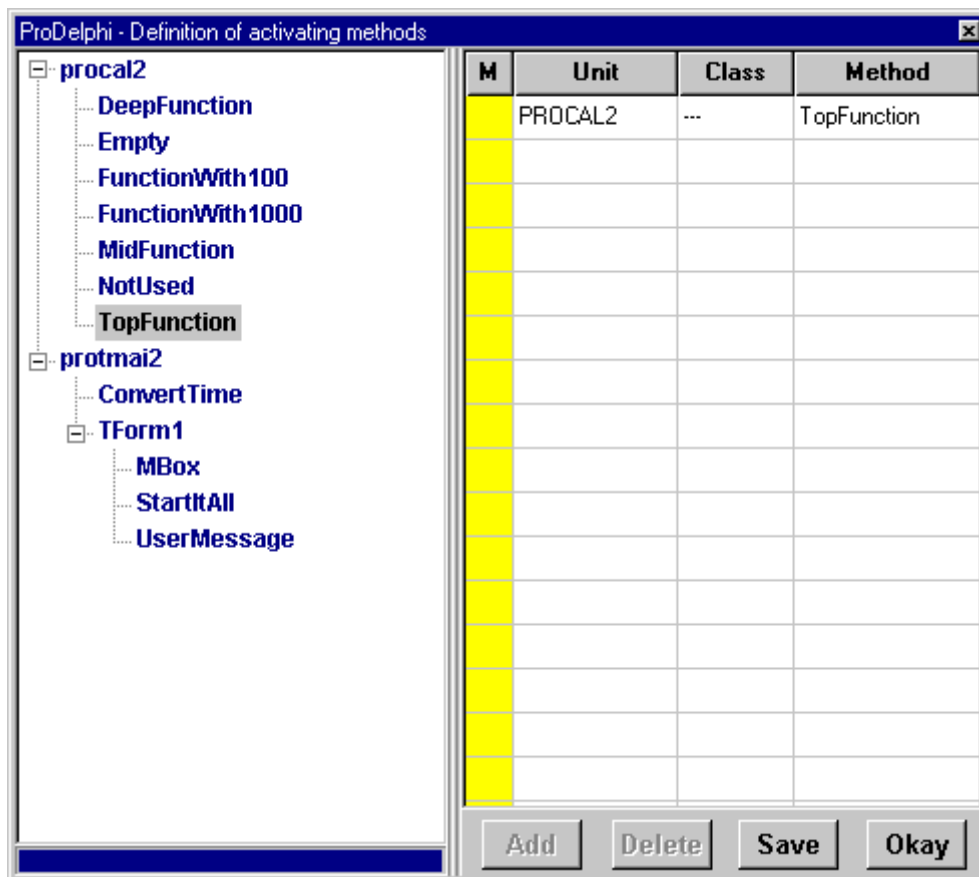
### ***At program start (default)***

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.

***See next page for an example***

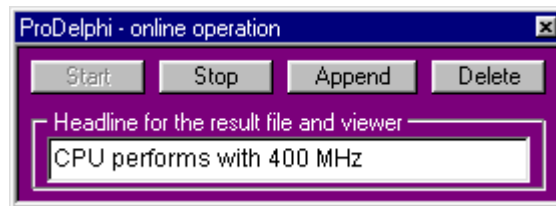


***By API-Calls or online operation window***

(see chapter A5.1 and A7 for details)

## A7 Online operation of the profiled program

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 'Append' - 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-Number' is incremented and you can switch between different runs with the viewer.

The default value for the headline for intermediate results is:

'CPU performs with XXX MHz' where XXX depends on your PC.

## A8 Profiling Dynamic Link Librarys (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.

*If you need the online-operation window, just insert two items into the DPR-file of your main program manually:*

*In the USES-clause you need to insert:           {\$IFDEF PROFILE } Unit Profonli, {\$ENDIF }*

*before Application.Run; you need to insert:     {\$IFDEF PROFILE } Profonli.ProfOnlineOperation; {\$ENDIF }*

All other necessary ProDelphi-units are linked into the DLL as well as into the program. There are only two small Units to be included: ProfInt (the interface to the DLL with the ProDelphi measurement functions) and the calibration unit ProfCali with is used by ProfInt. These two are included into program and DLL. But they are really small.

## 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 by 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 too 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 Prepared for Delphi 6

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.

As ProDelphi does not make a complete syntax analysis, it might occur, that not all places to insert the time measurement statements could be found. Maybe that some strange code constructs have been forgotten by me. As mentioned before, the large project, which was optimized with ProDelphi, was written by 12 different programmers, all their code was recognized correctly. Also the VCL could be compiled after profilation (all units of Delphi 3 have been profiled, except those, which used OBJ-files which were missing). In case ProDelphi does not recognize code correctly, you would get a compiler error by Delphi. In such a case, try to structure your source more simple. If that doesn't help, send me an E-Mail with the code.

While measuring, a user stack is used by the profiler unit. The maximum stack depth is 2400 calls. If a pocedure calls itself (recursive procedure), it only needs one entry in the profiler units stack.

In the freeware mode of ProDelphi only 30 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 process your code another time. It is NOT necessary to delete the old statements inserted by ProDelphi before.

## **A14 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 that directory. In case that an error occurs you can rename the saved file to \*.pas / \*.dpr / \*.inc.

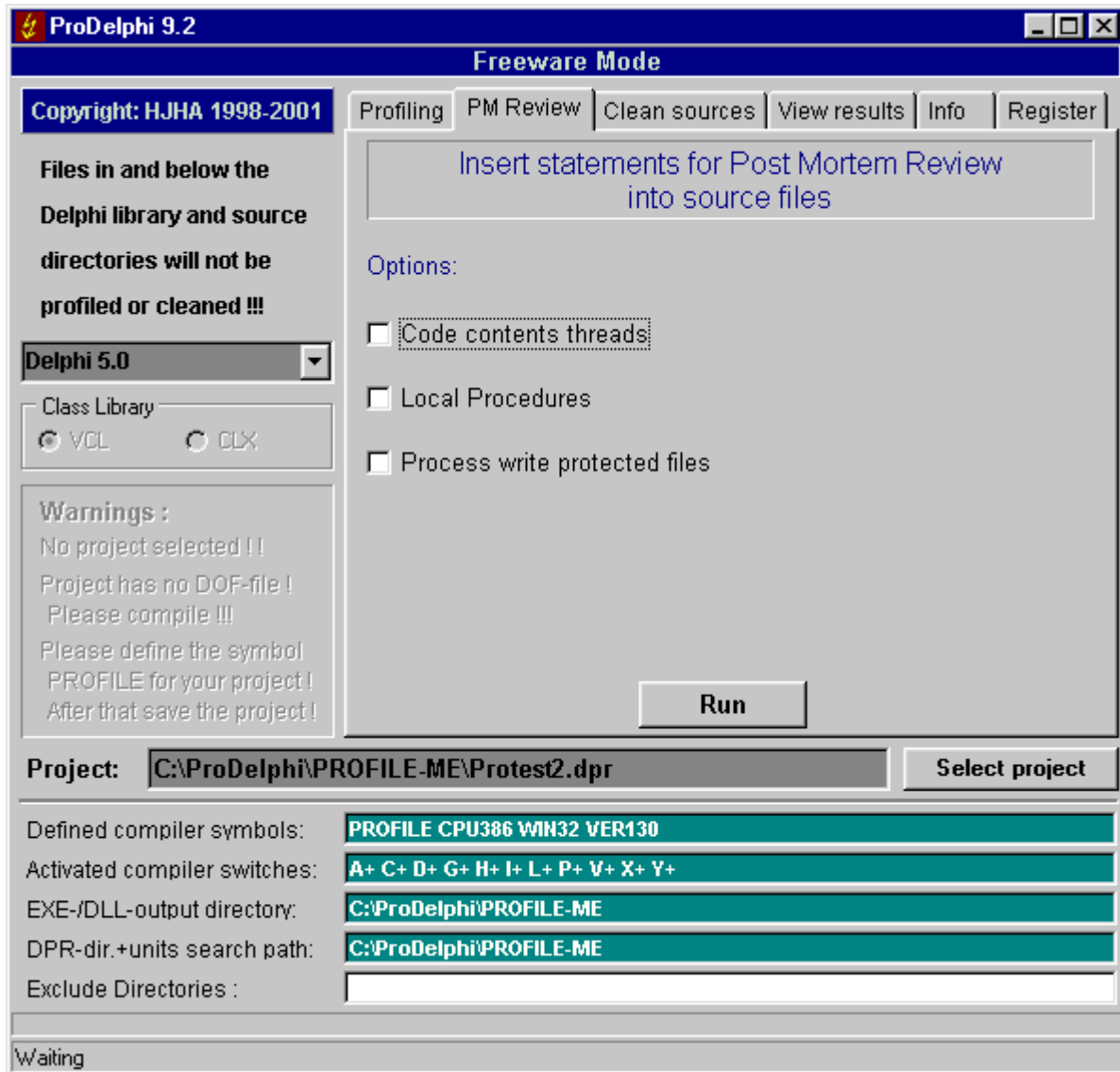
Before doing so, maybe it's worth to have a look into the output file. In case of a parsing error, you can send the original file + the incomplete output file to the author for the purpose of analysis.

## **A15 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 available options see chapter A4.

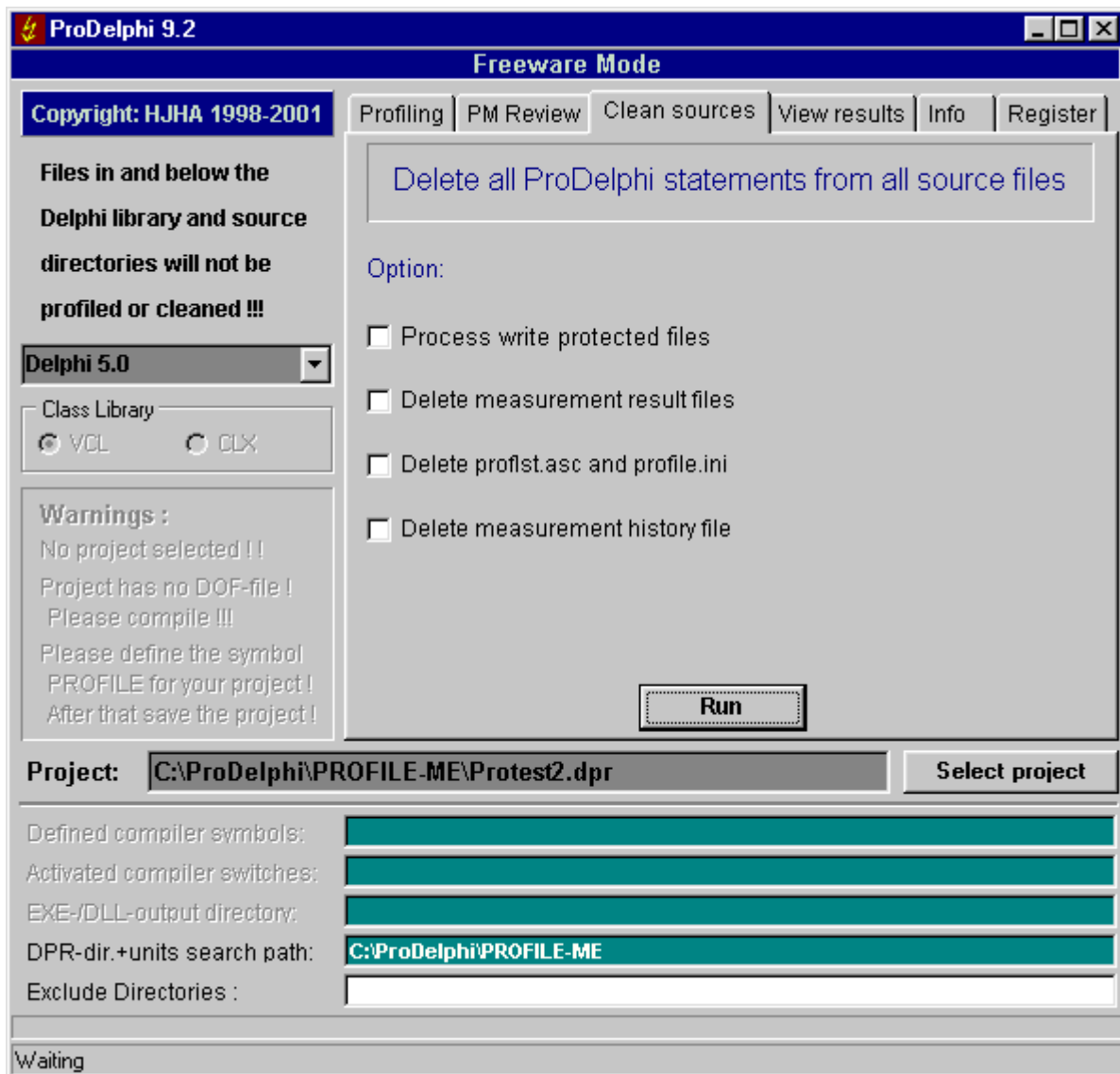
The handling of Prodelphi is the same as for profiling. You also have to define the compiler symbol `PROFILE`:

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).

## D Compatibility

ProDelphi was testet under

- Windows 95, 98, Windows NT 4.0 and Windows 2000,
- AMD K6 166 / 233 MHz, AMD K6-2 266 / 300 MHz, AMD K6-3 400 MHz, AMD Athlon 600 MHz,
- Pentium Overdrive 120 MHz, Pentium II / III 400 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. 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 'progrname.txt' (one line for each procedure):

```
run; unitname; classname; procedurename; % of RT; calls; RT excl. child; RT-sum excl. child; RT incl. child; RT-sum incl. child; % incl. child
```

File content of 'progrname.tx2' (one line for each run):

```
run; CPU-clock-rate; headline for that run
```

## **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 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. Info)  
Am Gruener Park 17  
90766 Fuerth  
Germany

E-Mail: [helmuth.adolph@prodelphi.de](mailto:helmuth.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	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.

## 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).