ProLaza and ProLaza64 User Guide

(Release 3.x for 32 and 64-bit-applications)

The Profilers for Lazarus
(for Pentium and compatible CPU's)

Profiling

The purpose of ProLaza is to find out which parts of a program consume the most CPU-time. ProLaza is based on ProDelphi (version 27.2), the profiler for Delphi 5 - XE7. ProLaza with its comfortable viewer, browser, history, and programmers API meanwhile is more than the legendary Borland Turbo Profiler. The viewer with it's sorted results enables the user to find the bottlenecks of his program very fast. The history function shows the user, if a preceding optimization was successful or not. ProLaza'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.

Post Mortem Review

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

32 and 64 bit versions

ProLaza is able to measure 32 bit applications developed with Lazarus 32 bit version.
ProLaza64 is able to measure 64 bit applications developed with Lazarus 64 bit version.

Differences between the freeware version and the professional version

With the freeware version up to 20 procedures can be measured or tracked, in the professional version up to 65500.
In the professional version additionally minimum and maximum run-times can be displayed in the viewer.
In the professional 32 bit version additionally assembler procedures can be measured and tracked.

Date: 4/18/2025

Contents of this description

Inhaltsverzeichnis

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

A. Profiling

A.1. Introduction

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

Any time a procedure / function / method (in the following 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 run time 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 ProLaza**. 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. This function is currently not supported.

The **fourth** file (programname.nev) contains the names of all methods which have never been called when measuring the run time 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.

The **fifth** file is also optional and only created, if the automatic switching off is activated (see A.5.1.).

A.2. Basic profiling

Using ProLaza is quite simple. It's Delphi-Version 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 (on which ProLaza bases). The program's run time could be decreased by 50 %.

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

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

If you want to measure procedures in a program and in DLL's simultaneously, take care of this:

Program and DLL's must have exactly the same units source path, their DPR/LPR-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 explicit path in the USES-statement in the DPR/LPR-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 A.9.).

If your files to profile are very large, and you have opened them in the IDE, you should close them. It was reported, that Lazarus does not properly actualize its 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 ProLaza still contains bugs. Just make a security copy of the program to be measured, e.g. by zipping all PAS-, DPR/LPR and INC-files.

For measuring the run time perform the following steps:

- Define the compiler-Symbol PROFILE.

- Deactivate the optimization.

- Optionally deactivate all run time checks.

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

- Start ProLaza from the Lazarus tools menu, from the Windows Start menu or somehow else.

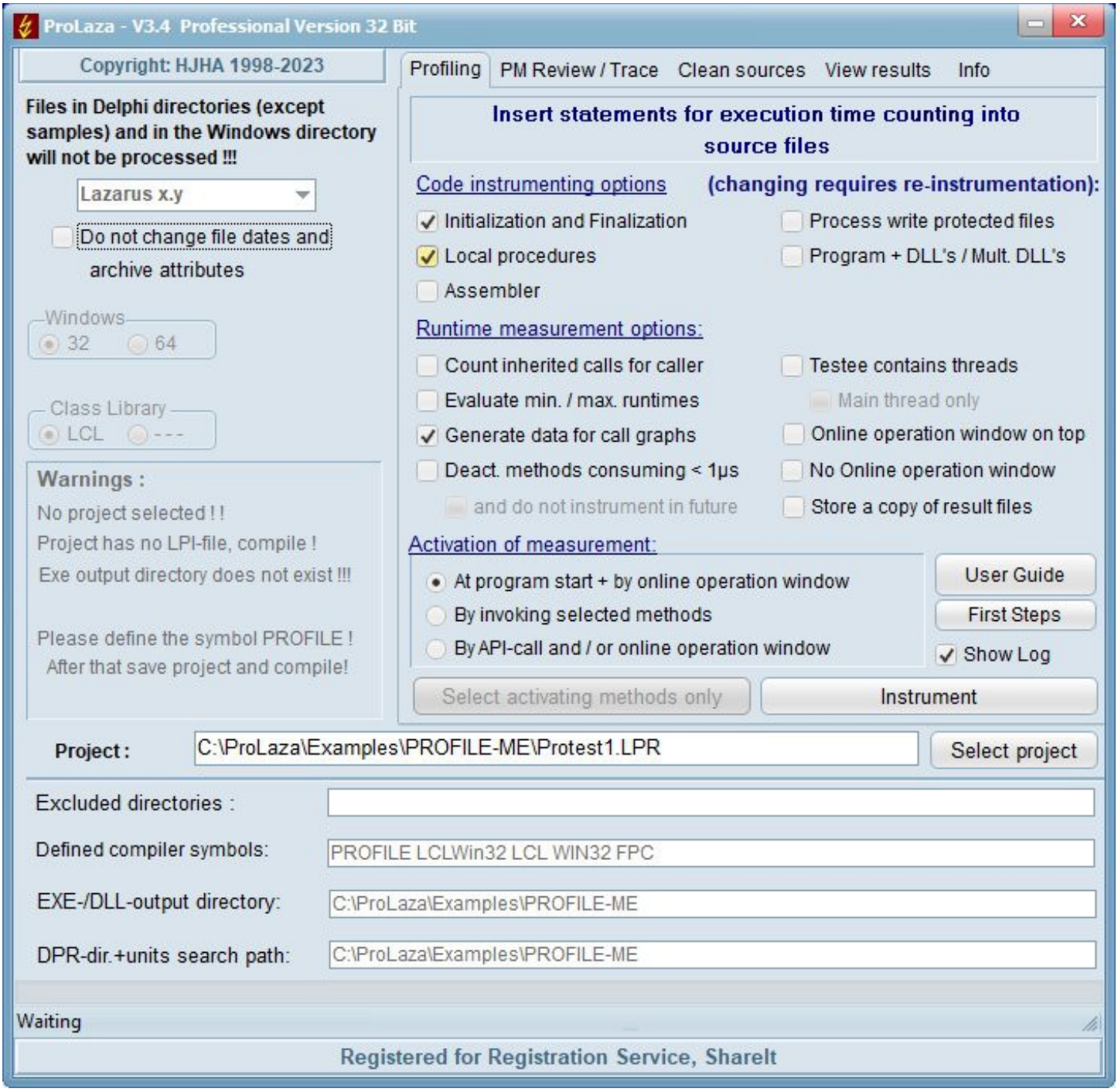
- With ProLaza select the project to profile (if it is not automatically selected).

- For the first example, only those options that are checked in the example of this guide are recommended.

Following options are available in professional mode only:

- Measuring units in the library path
- Assembler procedures (only 32 bit version)
- Evaluating minimum and maximum run-times (in the freeware mode only the much more important average run-times are available).
- Do not change file dates. Checking this option results in changing the file date/time by 2 seconds only when profiling, just enough to make Lazarus realize that a file has changed. The file date/time is set back to the value that was set when doing the first instrumentation by cleaning the sources.

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



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

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

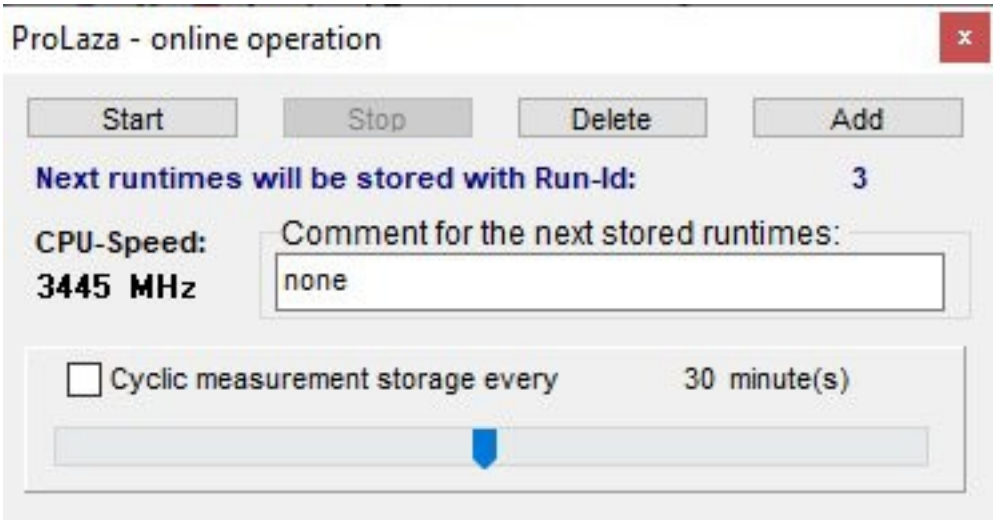
- Recompile the program (and / *or DLL's*).

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/LPR-files have to be in the same directory !!!

Files in and below the Lazarus 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:



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 tested program. Without the autostart option, you have to press the start button in the online operation window when you want to start the measurement, define activating methods or insert calls into your sources for activation or deactivation. See chapter A.5.2. for the complete description. After the program has ended, you can

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

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

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

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

ProLaza creates the file ‘proflst.asc’, it contains information about the procedures to be measured for profiling or traced for postmortem 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 A.4.1.).

Your compiled program the file with the name ‘progname.txt’ contains the data in the ASCII-semicolon-delimited format for database export and viewer and ‘progname.tx2’ for the headlines for the different intermediate results (for the built-in viewer). The file ‘programname.tx3’ is currently not used. 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. The viewer also uses this file.

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

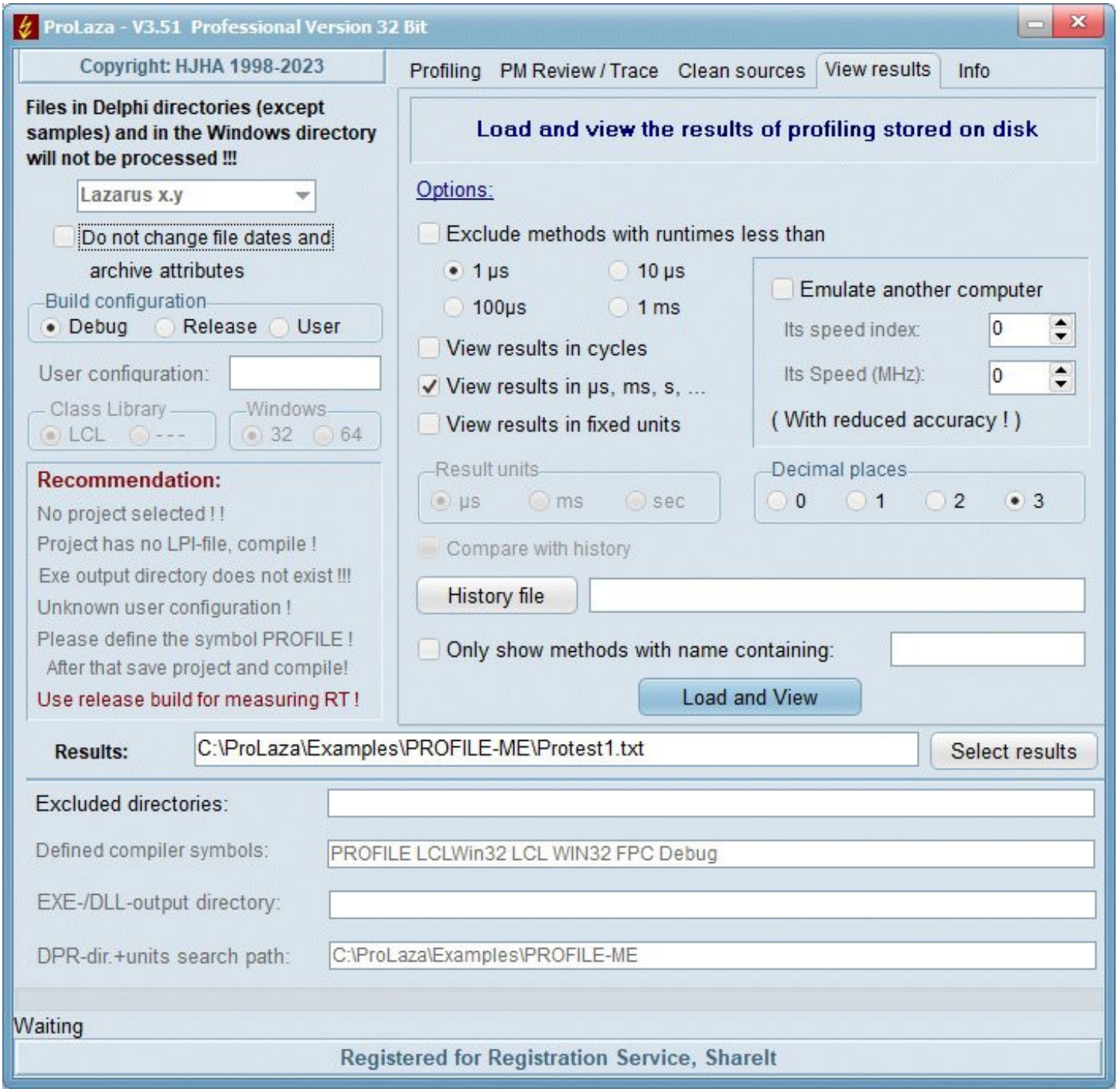
All files are stored in the output directory for the *.exe (*.dll) file.

To allow simultaneous measuring of DLL's and programs, all files in the units search path (except the Lazarus LIB and SOURCE directories and below them) are profiled if they are not write protected !!! Search path for program and DLL need to be identical in that case.

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

The most comfortable way to view the run times of your 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.



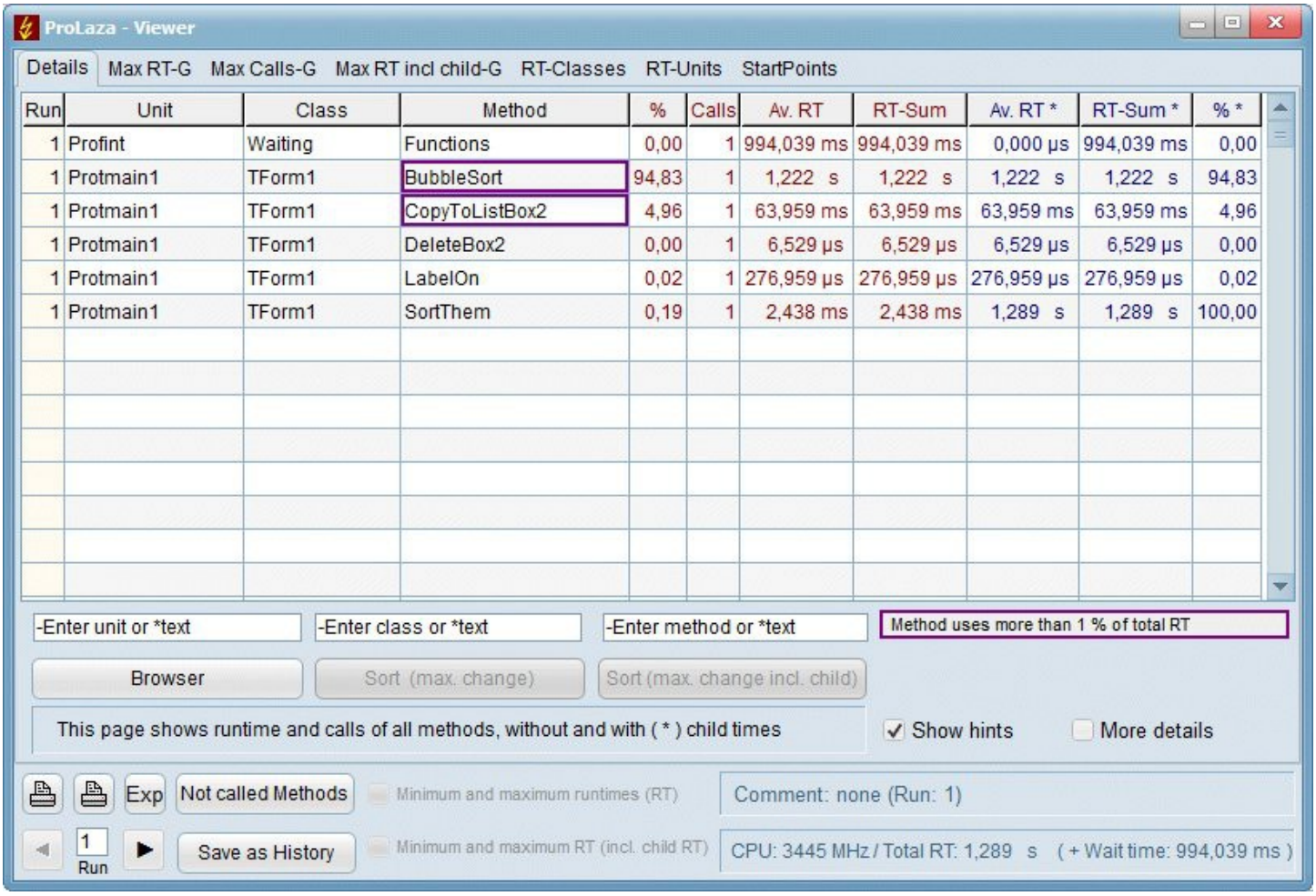
You can choose if you want to view the results in µs, ms ... or in CPU-Cycles.

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 ProLaza tell you how fast or how slow your program would perform on that PC

(see A.2.3.).

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



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 run time of all measured methods (alternatively in CPU-cycles)
Wait time: ttt	appears when time is spent by waiting for something (e.g. Sleep, MeassageBox etc.)
Comment: ccccc	Text set as comment in the online operation window for intermediate results, ‘At finishing application’ when the results were automatically stored when the testee ended or date and time when the online operation window cyclically stored results.

Sorting the table:

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

Navigating through the results:

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

The Exp - button:

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

The Minimum/Maximum check-boxes (Professional Mode only):

Checking these options, minimum and maximum runtimes are displayed if they were collected in the measurement (see Chapter A.2.. If these check-boxes are disabled, no minimum and maximum values were evaluated.

The History - button: see chapter A.4.1.

Meaning of Run:

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

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

Meaning of the columns with the RED text:

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

Meaning of the columns with the BLUE text:

Av. RT *	Average run time of the procedure inclusive its child procedures in CPU-cycles or in µs, ms, ... (in the professional mode also minimum and maximum run-times can be displayed)
RT-sum *	Runtime sum inclusive child times.
% *	Percentage of the total run time 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 ProLaza-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 run time (**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 run time (**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 run-times inside a red frame:

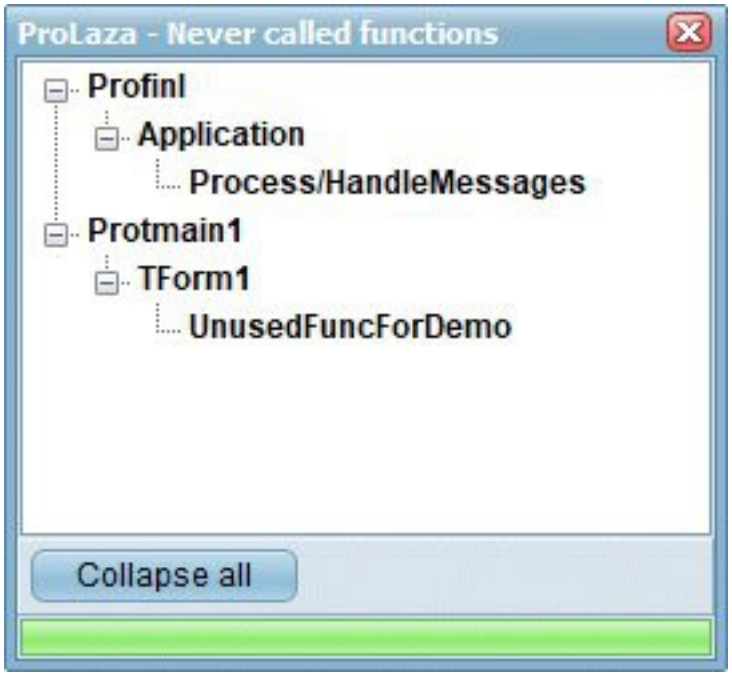
The run time is greater than the time stored in the history file. The frame only then appears, if the change is greater than 1% of the total run time of the application.

Meaning of run-times inside a green frame:

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

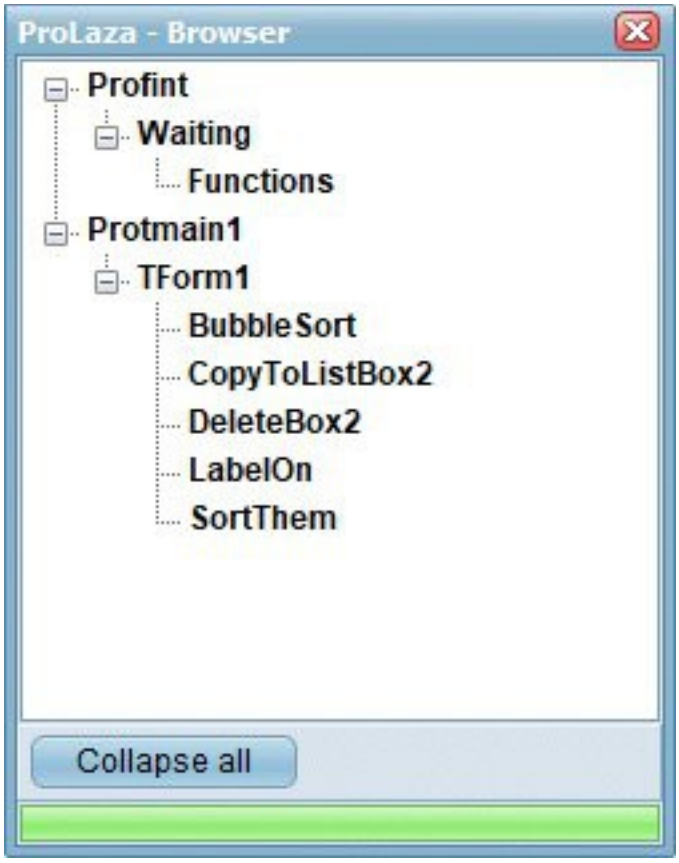
The Not called Methods - button:

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

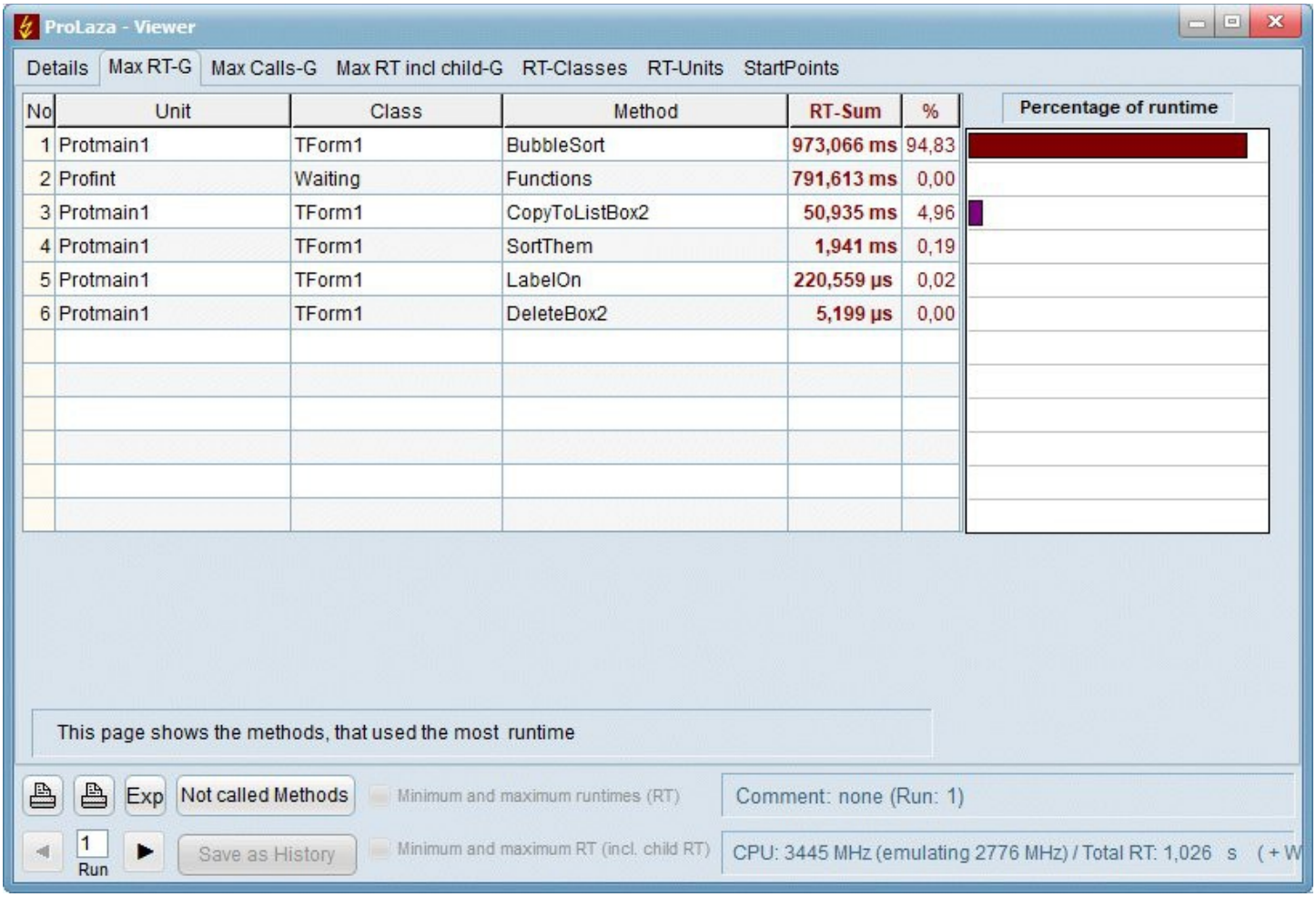


The Browser - button:

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



See next page please for another viewer window example



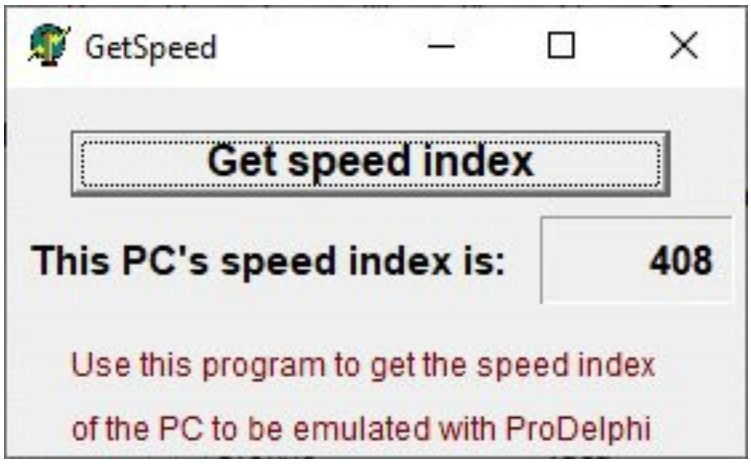
Example of: Maximum run time consuming methods (graphical)

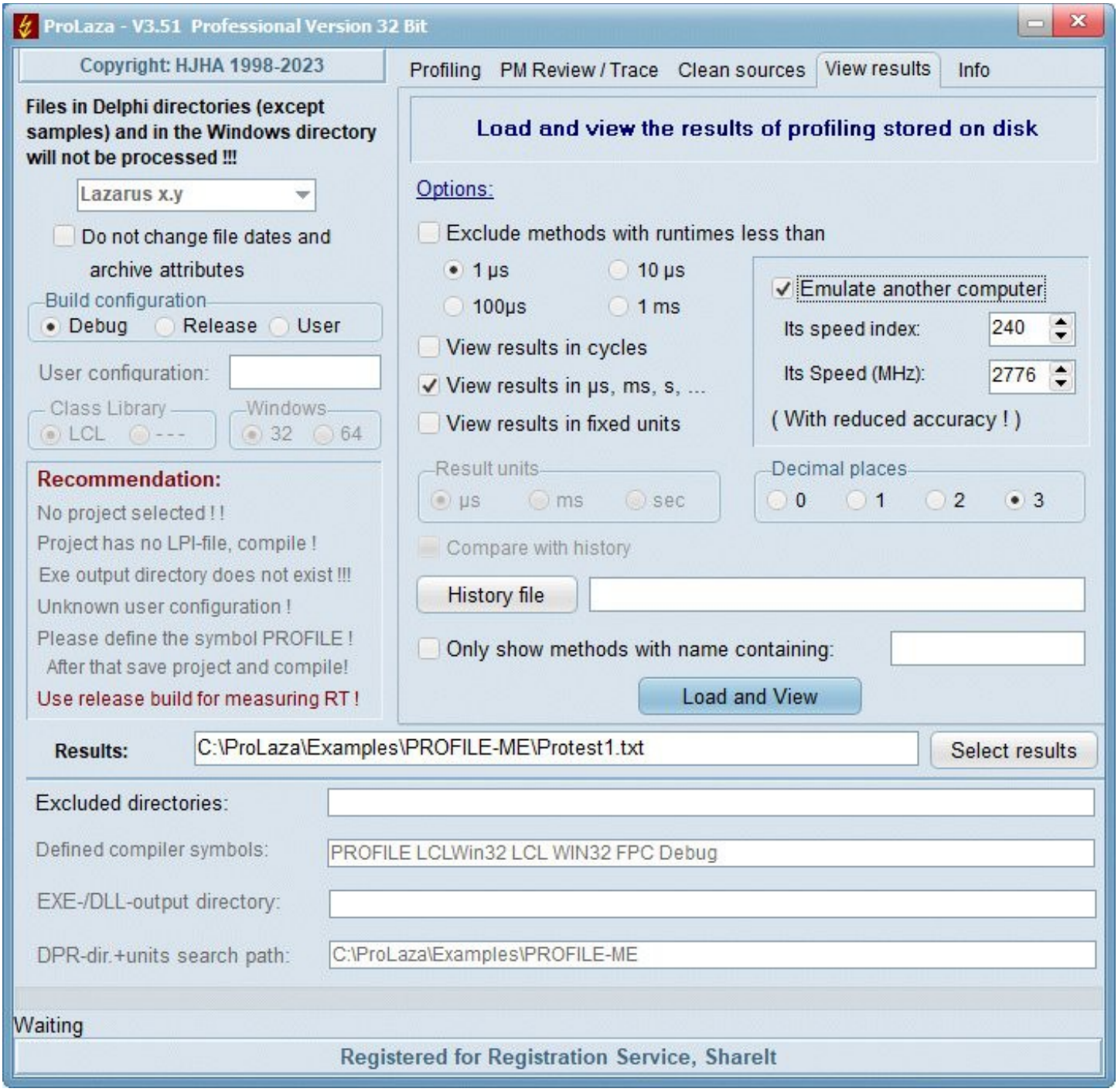
A.2.3. Emulation of a faster or slower PC

If you want to know, how fast (or slow) your program would perform on another PC, just use the program Getspeed.exe to get the other PC's speed index, enter it in ProLaza. Then 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 ProLaza is executed, is calculated automatically. So do not delete Getspeed.exe after installing ProLaza, it is used for this purpose also on the PC on which ProLaza is installed).


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

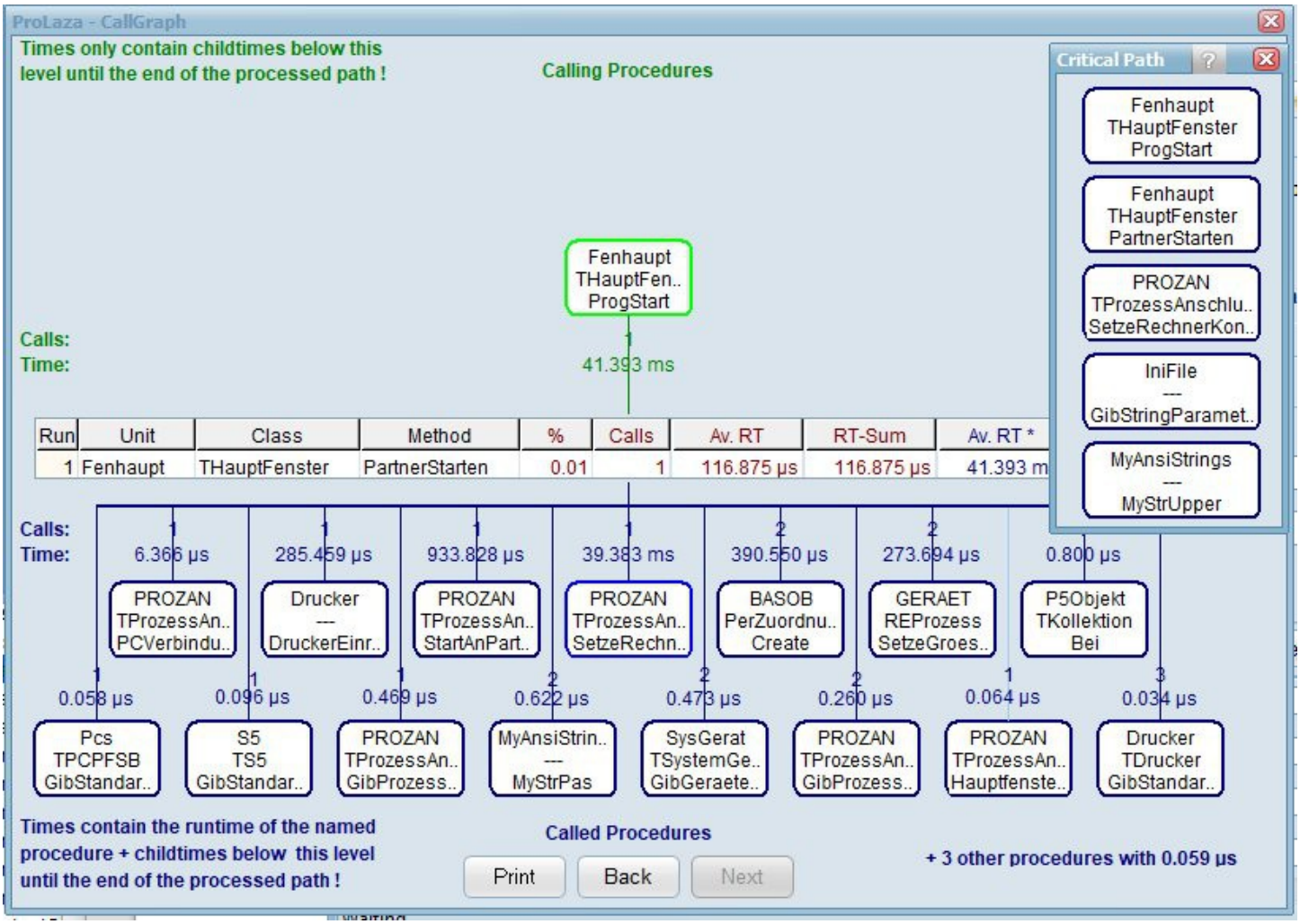

ProLaza - V3.51 Professional Version 32 Bit
Copyright: HJHA 1998-2023
Profiling
PM Review / Trace
Clean sources
View results
Info
Files in Delphi directories (except samples) and in the Windows directory will not be processed !!!
Lazarus x.y
Do not change file dates and archive attributes
Build configuration
Debug
Release
User
User configuration:
Class Library
LCL

Windows
32
64
Recommendation:
No project selected ! !
Project has no LPI-file, compile !
Exe output directory does not exist !!!
Unknown user configuration !
Please define the symbol PROFILE !
After that save project and compile!
Use release build for measuring RT !
Load and view the results of profiling stored on disk
Options:
Exclude methods with runtimes less than
1 µs
10 µs
100µs
1 ms
View results in cycles
View results in µs, ms, s, ...
View results in fixed units
Emulate another computer
Its speed index:
240
Its Speed (MHz):
2776
(With reduced accuracy !)
Result units
µs
ms
sec
Decimal places
0
1
2
3
Compare with history
History file
Only show methods with name containing:
Load and View
Results:
C:\ProLaza\Examples\PROFILE-ME\Protest1.txt
Select results
Excluded directories:
Defined compiler symbols:
PROFILE LCLWin32 LCL WIN32 FPC Debug
EXE-/DLL-output directory:
DPR-dir.+units search path:
C:\ProLaza\Examples\PROFILE-ME
Waiting
Registered for Registration Service, ShareIt

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

If the call graph data has been collected when measuring the run time (checkbox in main window) the call graph can be displayed. When clicking with the left mouse button on a measured result in the viewer grid, a new form opens which displays the runtimes of the selected 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 top of the form. Always those procedures are displayed, that consumed the most run time. For each procedure, the number of calls for the selected procedure and the run time 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 run time are displayed with the number of calls and the run time consumed inclusive child times **(Screen shot is not from the example program in this manual)**:



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

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

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

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

A.3. Getting exact results

If you measure program run-times a few times, you will see that the measurement results differ from measurement to measurement, without that you have changed your sources. Two kind of results will often differ: the run time of a method and the percentage of their run time 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 these problems. Because of the highest possible granularity of ProLaza (1 CPU-cycle), you see these differences.

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

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

Some disturbers everybody might be aware of:

- activated screen saver,
- Windows power management,
- background schedulers,
- online virus protection,
- automatic recognition of CD changing,
- temporary windows swap file causes memory transfers of different duration,

These disturbing influences are easy to eliminate.

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

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

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

The three influences are also easy to eliminate. You only need to exclude these procedures from measurement. Another way is not to use the autostart function of ProLaza but start measurement at the starting point of a certain action. How to exclude methods is described in Chapter A.5.1., how to measure defined actions only is described in chapter A.5.2..

A.3.3. Intel SpeedStep Technology / Turbo Boost mode

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

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

The influence of the cache cannot simply be ruled out. The only way to compare measurements is to ensure that the measurement situation is always the same. To do this, you should start an action twice in succession and only use the second measurement for later comparisons (save the second measurement as a history). The easiest way to do this is via the online operating window by saving the results after the first and second execution. The second measurement is then saved as a history in the viewer.

A.3.5. 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 uses the full CPU speed, if working with battery power, the CPU speed changes dynamically. This does not directly affect the measurement: ProLaza measures CPU cycles. If we look at the CPU - cycles displayed in the viewer, the measurement is correct. If times are displayed, it could be that too long or too short times are displayed. It depends on the CPU speed that was set when the CPU speed was measured. Different processors use different algorithms to change the speed. The only way to get 100% correct results is to switch off the power safe mode.

A.3.6. Summary

If you eliminate the disturbances mentioned in / A.3.1 / A.3.3 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.

A.4. Interactive optimization

Interactive optimization means that you optimize something, check if it has brought you significant decreasing of run time 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 % run time must be optimized by 50 % to decrease the total program run time by 5 % !!!

There is a way to compare the measurement results:

> Use the viewer, save the results as history, make the optimization make a new measurement and compare both with ProLaza's history function.

A.4.1. The history function

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

Having made a measurement, you can store the results being displayed in the viewer's 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 color the cells of the viewer's table, by this you have a quick overview about all changes of run time: Red means method got slower, green means method got faster and white mean that no essential change occurred.

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

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

If you succeed in excluding disturbing effects as mentioned before, you can use the history very well.

A.4.2. Practical use of the history function

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

A.5. Measuring parts of the program

A.5.1. Exclusion of parts of the program

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

What I described above, is the default setting of ProLaza: 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 Lazarus 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 'Exclude directories' of the ProLaza 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 ProLaza:

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

8. Automatic exclusion

You can automatically exclude methods by enabling the 'Deactivate methods with runtime < 1 µs' option.
If you enable this option,

- the runtime of a method that has been called at least 10 times,
- has not called any other measured method, and
- has taken an average of less than 1 µs during the measurement period

will no longer be explicitly recorded and displayed beginning with the next program start. For this purpose, a file is created when the program terminates. It contains all methods whose runtimes should not be explicitly recorded. This file is read the next time your program is started. The runtimes of the identified methods are added to the runtimes of their calling methods.

It is possible that other methods with a runtime of less than 1 µs will be found after the next program execution. These methods will then also no longer be recorded explicity.

Optionally, no measurement calls will be inserted into these methods during the next instrumentation. Their runtime is then automatically included in the runtime of their calling method. This means that a program can contain more than 65,500 methods.

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

Note! If this exclusion should be permanent, insert a //PROFILE-NO and a //PROFILE-YES statement around such a method in the source code.

A.5.2. 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 ProLaza (1. and 2. can be used simultaneously):

1. By dialog

In the main window of ProLaza 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 ProLaza. 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 there due to compatibility with old versions of ProDelphi. It is not very comfortable. Using this method, you always need to insert two calls, one for activation and one for deactivation.

A.5.3. 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 that start an action are entered in a list of starting points. Just perform a measurement run which 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.

A.5.4. 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 out this is to restructure the procedure into neat parts, or to divide it up by means of local procedures. Another idea would be that ProLaza would measure each block of a structure and not the whole procedure. The last solution would cost a lot of measurement overhead and would make time critical applications stop working. For the case that both solutions given is too much work or is too risky, ProLaza 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
```

After that, re-profiling with the option 'Profile local procedures' is necessary !!!

Profiling the sources after this change causes ProLaza to insert measurement statements right after the comments. The run time measured in this so defined block will be found in the viewer by searching for 'procedure name(comment)'.

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

The time measured in this part is not included in the run time of the procedure but it is included in the child time.

Example:

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

The total run time 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 run time 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 a different way. Especially processors with a small cache have the problem, that not the whole procedure inclusive measurement parts of ProLaza fit into the cache, so additional wait states occur.

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.

Nesting example:

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

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

A.6. Programming API

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

A good way to make different result files comparable, is to measure only those actions of your program you want to optimize. In that case, do not check the button for 'automatic start' of measurement. Do the 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
{$IFDEF PROFILE}ProfStop; Try; ProfEnter; end; {$ENDIF}
   Sort All;  // the method of which you want to know the run time
{$IFDEF PROFILE}finally; call ProfExit; end; {$ENDIF}
END;
```

You can modify the code in three different ways:

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

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

{ possibility 3 }
//PROFILE-NO
PROCEDURE TForm1.ButtonClick;
BEGIN
{$IFDEF      PROFILE}try; ProfInt.ProfActivate;{$ENDIF}
   SortAll;  // the method which you want to know the run time 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 instrumented by ProLaza. 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.

A.6.2. Preventing to measure idle times

Some Windows-API functions and Lazarus 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 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 Lazarus-calls are replaced automatically by the Unit ‘Profinl.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 ProLaza-API-calls StopCounting and ContinueCounting. You can find the calls, which are re-defined, in the unit ‘Profint.pas’ and chapter A10. They automatically call these functions before using the original Windows- or Lazarus calls. Some functions are replaced by the profiler (e.g. Application.HandleMessage).

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

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

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

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

Important:

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

A.6.3. Programmed storing of measurement results

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

Just insert the statement

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

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

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

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

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

into your source.

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

Important:

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

More important:

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

A.7. Options for profiling

Profiling options are divided into three groups:

- Code instrumenting options (or vaccination options): How and what to instrument.
- 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 Lazarus version / file date.

A.7.1. Code instrumenting options:

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

Assembler procedures (32 bit Professional Mode only)

Assembler code is normally not profiled (often assembler is a result of an optimization process already). In the professional mode of the 32 bit version, this option can be used.

Initialization and finalization

Normally, the initialization and finalization parts of the units are not measured. In case you want to do this, check the

appropriate option if you use the keywords INITIALIZATION and FINALIZATION in your units.

Profile local procedures

Normally local procedures are not instrumented and 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 ignored, 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 A.9. for details.

A.7.2. 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 are deactivated, which

- are called at least ten times and
- have no child procedures except deactivated ones and
- consume less than 1 µS.

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

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

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

Generate data for call graph

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

Online operation window on top

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

No Online operation window

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

Store a copy of result files

The measurement result files have the names

‘application name.xxx’

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

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

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

Testee contains threads

If this option is checked, the measurement is enhanced for handling threads. It is not useful to check this option if your program does not create threads, the program only runs slower. But it is 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 run-times (Professional Mode only)

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

A.7.3. Measurement activation options

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

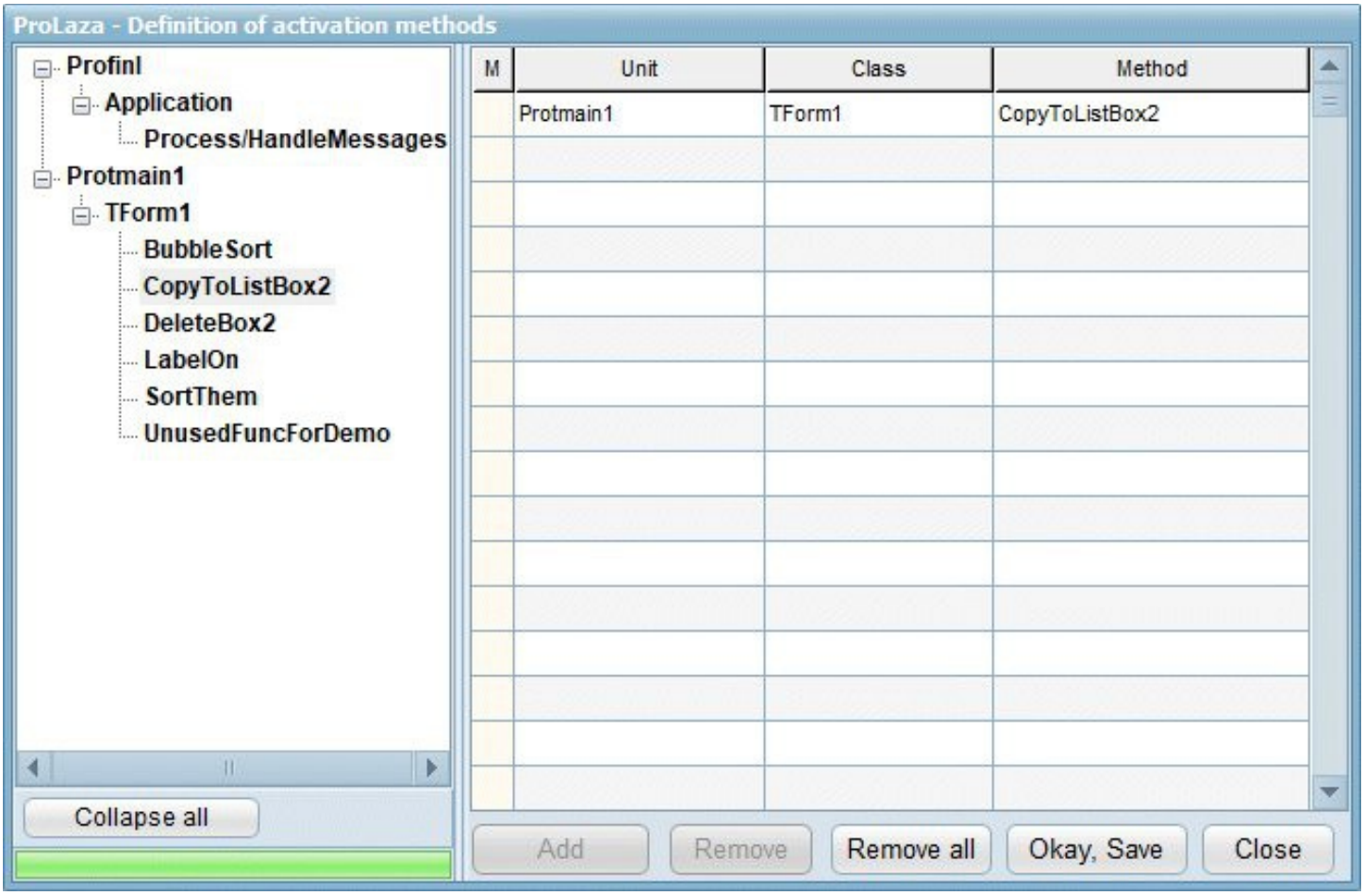
At program start (default)

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 chapterA.5.2.). If you use this option, you should not use the Online-operation window.

Example:



By API-Calls or online operation window

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

A6.4 General options

Lazarus version

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

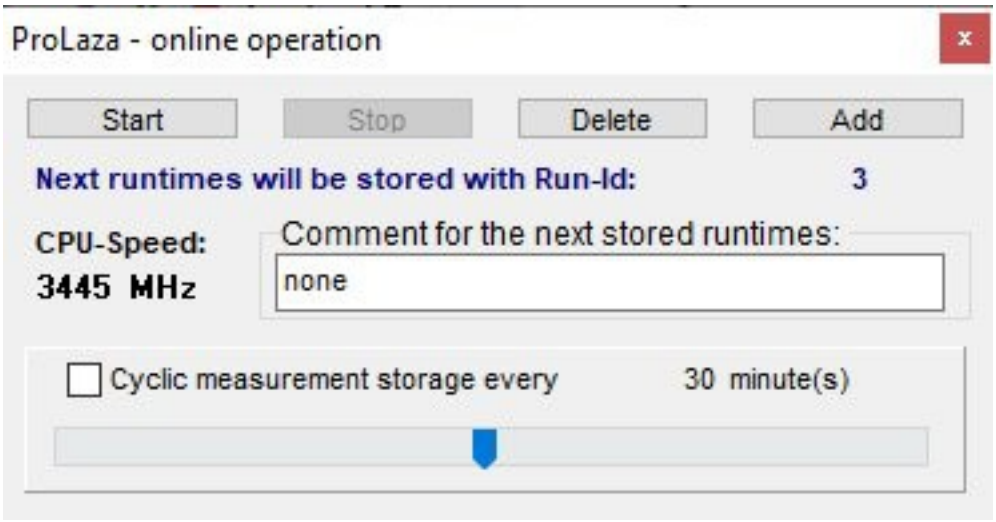
File Date

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

When cleaning the sources the file date/time is decreased by 2 secs for each profiling process resulting in a file date which is identical to that one when starting profiling (unless the file is changed by the editor). This makes possible that the file date keeps the same between checking out and in from a source code control system.
As some source code control systems also check the archive attribute, ProLaza keeps its status as it is before instrumentation.

A.8. Online operation window

With the online-operation window



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

You can edit the text which is the headline for the 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 in the viewer.

The default value for the headline for intermediate results is:

'none’.

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

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

A.9.1. DLL’s

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

Some precautions are needed to avoid problems:

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

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

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

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

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

> ***ProLaza reads the search path and the compiler switches, depending on the Lazarus version, from the project - 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.***

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

If you measure the DLL without the program and need the online operation, 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}

A.9.2. Packages

Packages are not supported.

A.10. Treatment of special Windows- and Lazarus – 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 ‘Profinl.pas):

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

Because of this redefinition, the Profinl-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. Lazarus itself places them into the interface part.

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

A.10.1. Redefined Windows-API functions

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

A.10.2. Redefined Lazarus-API functions

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

A.10.3. Replaced Lazarus-API functions

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

There are some LCL-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.

A.11. Conditional compilation

Conditional compilation is fully supported.

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

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

The directives $IF, $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;
```

A.12. Measuring on a customer PC

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

Beside the files belonging to the application, copy also following files to the customer's PC:

For ProLaza:

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

For ProLaza64:

- Profmeas64.dll
- ProfOnFo64.dll
- ProDVer64.dll
- ProDVer32.dll
- Profcali64.dll
- Proflst.asc
- Profile.ini

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

A.13. Limitations of use

A.13.1. General

Console applications have no online operation window.

The measured times always differ about 10 % (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. On a multiprocessor machine, the results might differ even more.

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

While measuring, the profiler unit uses a user stack. The maximum stack depth is 16000 calls.
The maximum number of threads that can be measured simultaneously by ProLaza is 32.
In the freeware version of ProLaza only 20 procedures can be measured, in the professional version 65500.

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

A problem for measurement is Windows itself. Because it is a multitasking system, it may let other tasks run besides the one you are just measuring. Maybe only for a few microseconds. So your program can be interrupted by a task switch to another application. I've made tests and 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 Intel's Pentium or AMD's Athlon are able to execute instructions in parallel. When the profiler inserts instruction, the parallelism is different from without these instructions. That's another reason, why the run time 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 run time and the measured run time is.

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.

A.13.2. Aborted procedures

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

A.13.3. Measuring multiple applications

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

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

A.13.4. Excluding instrumentation of directories for all projects

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

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

A.14. Assembler Code

Pure Assembler procedures and functions (e.g. FUNCTION Assi : Integer; asm mov eax,2; end;) are profiled only in the 32-Bit Professional version.

A.15. Modifying code instrumented by ProLaza

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

A.16. Hidden performance losses / Tips for optimization

ProLaza measures run-times 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 always changes the run time 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 run time
BEGIN
  FOR i := 1 TO 10 DO
    Result := LastFunction;
  IF Result > 0 THEN
    GoTo final                          // in connection with this GoTo
  ELSE
    Result := -1;

  Final:

END;
```

A.17. Error messages

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

Vaccinating a file is done in this way:

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

- after that the renamed file is parsed and instrumented, the output is stored into a *.pas-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 take a look into the output file. In case of a parsing error, you can send the original file + the incomplete output file to the author for the purpose of analysis.

A.18. Security aspects

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

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

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

ProLaza can be started from command line (or batch file). Depending on the command line parameter, ProLaza can perform profiling (instrumenting) of the source files, clean the sources or start the viewer. The first parameter is always the LPR-file.

A.19.1. Automatic profiling

If as parameters the Lazarus version and the parameter /PROFILE are set, ProLaza automatically instruments the named program and terminates after that.

Syntax:

Profiler path\application.lpr /D2 /PROFILE

Precautions:

- Profiling should have been done before interactively to be sure that all necessary data exists.
- Profiling should not cause any warning in the profile log.

Example:

```
cd ProLaza
Profiler F:\AppDir\Testprogram.lpr /D2 /PROFILE
```

A.19.2. Automatic cleaning

If as arguments the Lazarus version and the parameter /CLEAN are set, ProLaza automatically cleans the named program and terminates after that.

Syntax:

Profiler path\application.lpr /Dlazarus-version /CLEAN

Example:

```
cd ProLaza
Profiler F:\AppDir\Testprogram.lpr /D2 /CLEAN
```

A.19.3. Automatic opening of the viewer

If as arguments the Lazarus version and the parameter /VIEW are set, ProLaza automatically starts the viewer and shows the measurement results.

Syntax:

Profiler path\application.lpr /D2 /VIEW

Example:

```
cd ProLaza
Profiler F:\AppDir\Testprogram.lpr /D2 /VIEW
```

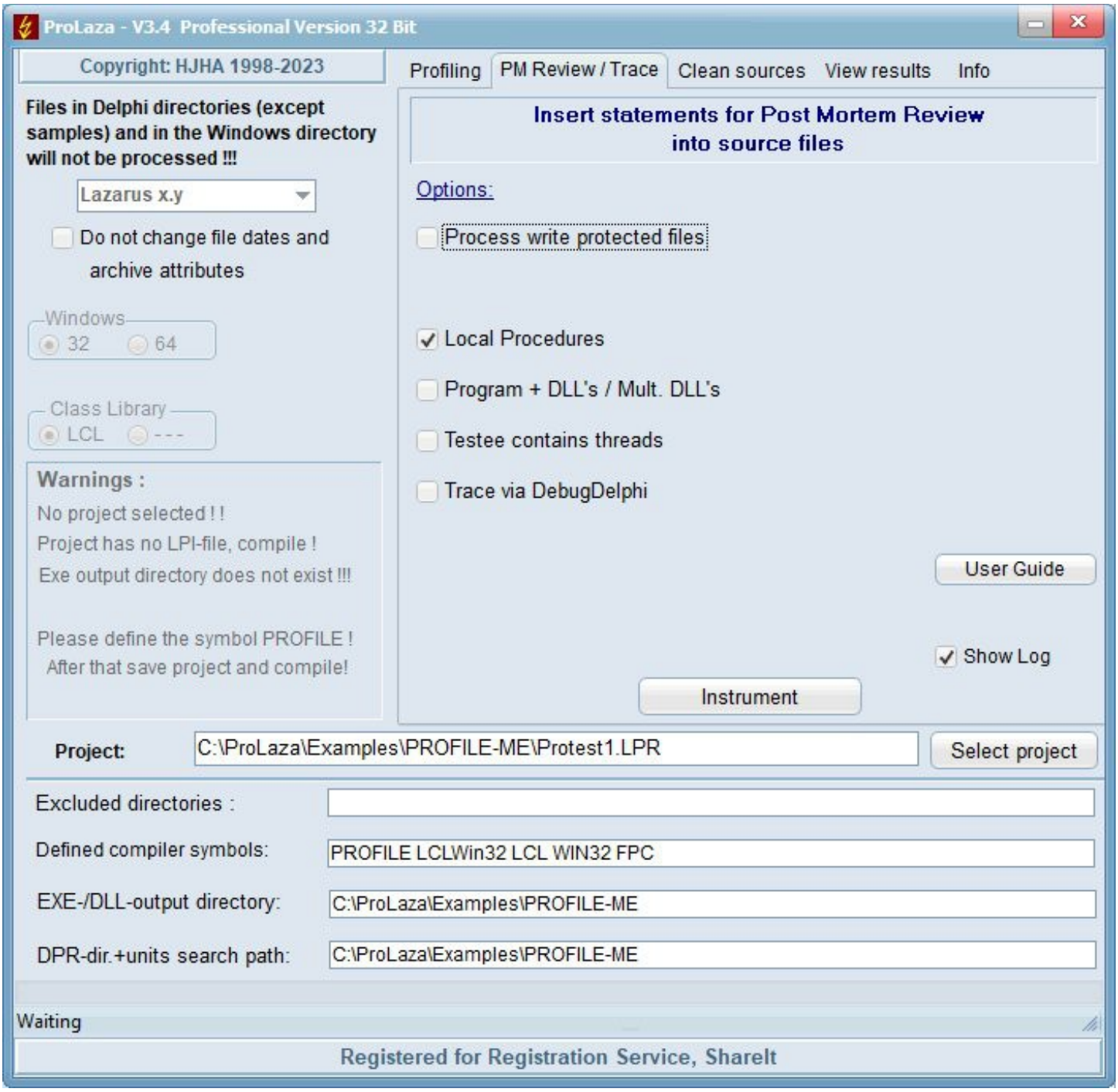
A.20. National language support

Currently, an English and a German user interface are supported. Which to use is specified when installing ProLaza with the setup program. But also later, ProLaza can switch between these two languages. This can be done by the system menu. It has two additional entries: ‘English’ and ‘Deutsch’ (= German).

If another language should be used: There is a file installed with the name ‘TranslateMe.LAN’. It contains the English text strings. By translating these and renaming the file into ‘Deutsch.LAN’ one can produce a user interface for his own language (and loose the German translation).

B. Post mortem review

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

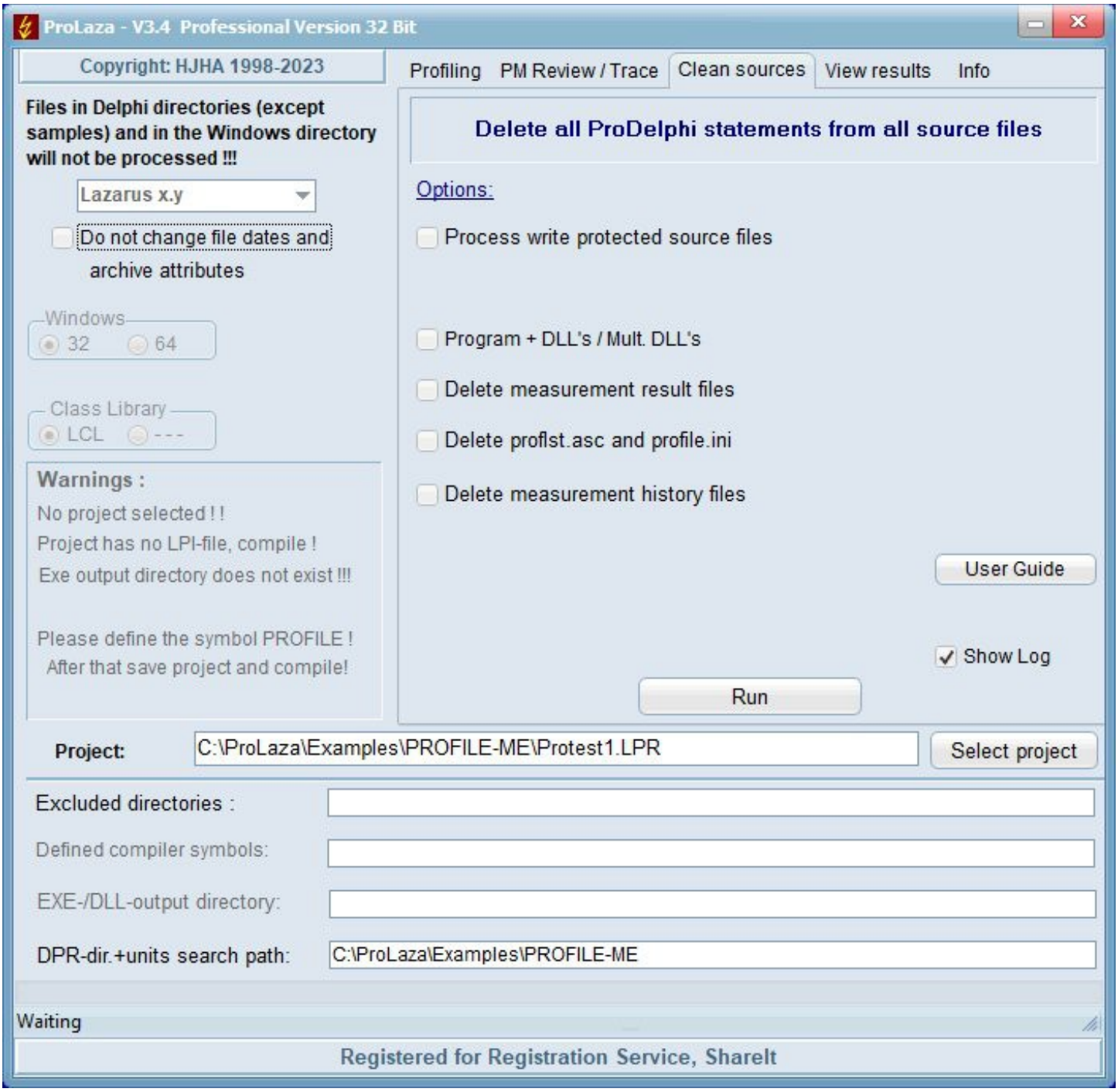
The handling of ProLaza is the same as for profiling. You also have to define the compiler symbol PROFILE:

If you have instrumented ProLaza with statements for postmortem review and work with the IDE of Lazarus 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 ProLaza itself needs stack space. And if there is no stack anymore, ProLaza can not work properly. The overflow might as well appear in the ProLaza stack tracing routines. ProLaza can not handle this.
If the **Trace option** is checked, additional WriteLn calls are inserted into the source file. These WriteLn calls produce trace information which can be viewed with DebugDelphi. For this purpose, DebugDelphi needs to be installed and started. If this option is checked, every entering and leaving a method is listed in the DebugDelphi log. To use this option, DebugDelphi must be installed and started.

C. Cleaning the sources

If you want to delete all lines that ProLaza 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 ProLaza 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 2 sec and decreased at cleaning by 2 sec. This makes possible that the file date keeps the same between checking out and in from a SCCS.

D. Compatibility

ProLaza runs on all Windows versions since Windows 95 on all computers with a Pentium compatible processor.

E. Installation of ProLaza

ProLaza is most comfortably installed with the included setup program (Setup.Exe). This program copies all necessary DLL's into the installation directory. Also, it creates an entry in the list of programs (Windows Start menu / Programs) and shows how to integrate ProLaza into the Lazarus tools menu.

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

The result file can also be used for export to a database (e.g. Paradox) or a spreadsheet program like Open Office Calc.

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

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

Description of call graph data:

0;0; if no call graph data exists

or

Called-From-Information Calling-To-Information

Description of Called-From-Information:

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

Description of Calling-To-Information:

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

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

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

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

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

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

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

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

```
run; CPU-clock-rate; keyword; headline for that run    //keyword is either MINIMAXON or MINIMAXOFF
```

G. Updating / Upgrading of ProLaza

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

H. How to order the professional version

Customers who want to use the professional version, can order it via Digistore24 registration service. A special download link for getting the professional version and a registration key will be sent via e-mail. Just start the professional version of ProLaza (Profiler.exe), select the page for registration and enter the registration number. At the next start of ProLaza, the Professional mode is unlocked. This key is also valid for upgrading to later versions.

I. Author

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

E-Mail: helado@prodelphi.de
Home page: https://www.prodelphi.de

J. History

Version 1.0 : 9/97	First release of ProDelphi
Version 27.0 2/12	Adaption to Delphi XE3.
Version 27.1 11/12	Problem with excluding directories solved.
Version 27.2 11/12	Viewer improved. Result can be exported to CSV-files (';' - separated)
Version 1.0 : 11/12	Adaption of ProDelphi to Lazarus and first release of ProLaza.
Version 1.1 : 3/13	Setup improved, interface fixed.
Version 1.2 : 4/13	UI beautified, German UI
Version 1.3 : 6/13	Bugfix, warning added
Version 1.3 : 9/13	64 bit version
Version 1.4 : 5/14	Exclusion of instrumentation of directories for all projects.
Version 2.0 : 11/16	UI beautified
Version 2.1 : 8/16	Command line parameters for batch profiling improved
Version 2.2 : 4/17	Viewer problem solved, Log window can be hidden
Version 2.6 : 3/19	Issue with very long units search path solved
Version 2.7 : 4/19	Exclude-directory window made sizable
Version 2.8 : 5/19	Missing headlines for exception window added
Version 2.9 : 9/19	Bugfix: Measurement started by selecteg methods did not produce results when HandleMessage or ProcessMessages was called
Version 2.10:10/20	Bugfix for measuring threaded programs
Version 2.11 9/21	Problem with Windows scaling text by 150 % solved Displaying of excluded directories fixed.
Version 2.12 (3/22)	Internal optimizations
Version 2.13 (6/22)	UI Beautyfied
Version 3.0 (10/22)	Measurement of waiting times.
Version 3.1 (4/23)	Improvement: Viewer displays values with localized decimal separator (. ,)
Version 3.2 (4/23)	Wrong warning message deleted
Version 3.4 (5/23)	One click expanding for browsers added

	Callgraph window enlargement added
Version 3.5 (7/23)	Cosmetical improvement of the viewer Log improvement Some wrong hints corrected Progress bar fixed Switching off hints in the viewer deactivates hints in the call graph too English corrected Precision in the viewer selectable: 0, 1, 2 or 3 decimal places
Version 3.6 (11/23)	Viewer font changed to improve readability Paper printout font also changed to improve readability
Version 3.61 (4/24)	Fonts in callgraph improved
Version 3.62 (4/25)	Settings read from project file and visualized by ProLaza not changeable anymore. Programs with more than 65500 procedures can be measured. - A maximum of 65500 methods can be instrumented. - So not all methods will be measured. - The times of the unmeasured methods will be included in the measured ones.

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).