ProLazaX and ProLazaX64 User Guide

(Release 1.x for 32 and 64-bit-applications)

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

Profiling

The purpose of ProLazaX is to find out which parts of a program consume the most CPU-time. ProLazaX is based on ProLaza (version 2.4), the profiler for ProLaza for Linux. ProLazaX with it's 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 bottle necks of his program very fast. The history function shows the user, if a preceding optimization was successful or not. ProLazaX'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.

32 and 64 bit versions

ProLazaX is able to measure 32 bit applications developed with Lazarus 32 bit version.
ProLazaX64 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 64000.
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: 11/15/2017

Contents of this description

BEFORE using ProLazaX practically, please read Chapter A.15 carefully !!!

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 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 four 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 ProLazaX**. 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 is currently not used by ProLazaX(64).

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 be the viewer, it is displayed as a hierarchical tree when you press the button named 'Not called methods'. This button is not enabled if all methods have been called or if you display the measurement results of a former version of ProLazaX.

A.2 Basic profiling

Using ProLazaX 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 (on which ProLazaX bases). The programs run time could be decreased by 50 %.

Install ProLaza as described in chapter G.

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 and *executable*).**

Libraries are not supported.

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 it's window content if a file is very large and the file is changed on disk from outside the IDE.

If no compilation errors occur, you may profile your program.

Don't use the original units for profiling, maybe ProLazaX 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.

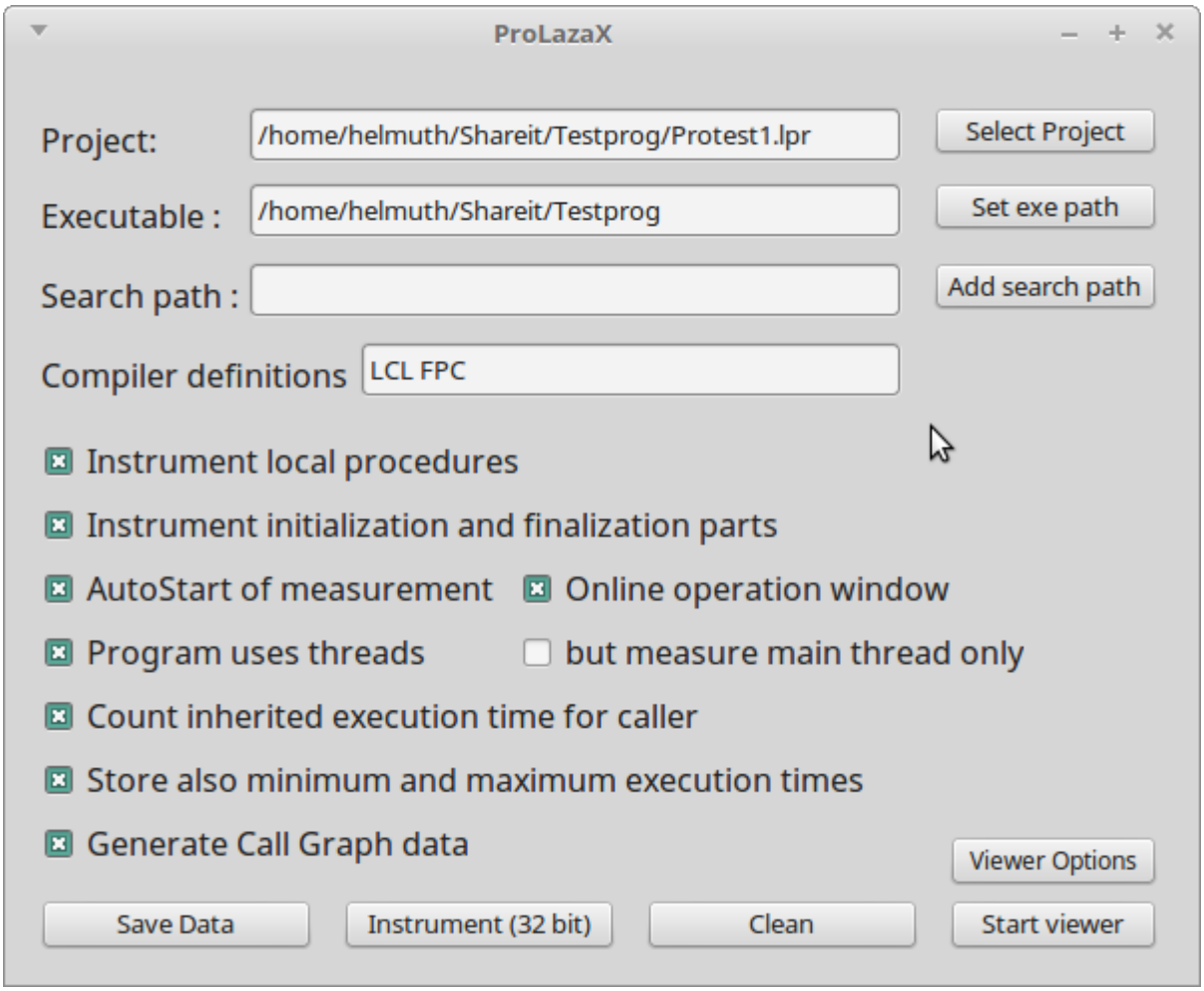
- Start ProLazaX from the Lazarus tools menu, from the Linux start menu or somehow else.

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

- Enter the path for the executable (if it is not automatically selected).

- Enter the compiler definitions (if they are not entered yet).

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



- Save Data

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

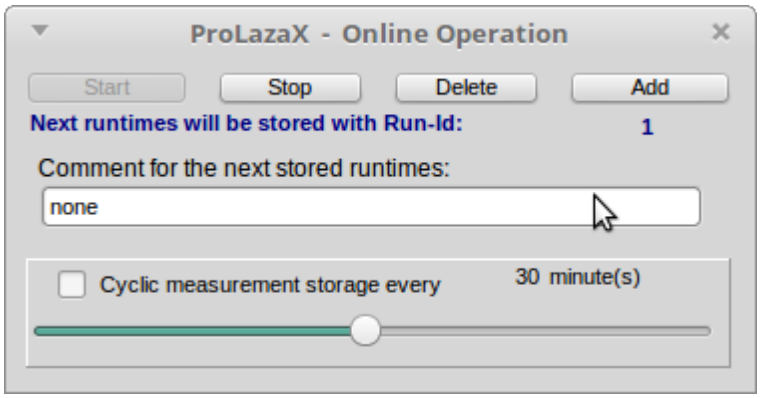
- Click the Instrument-buttons. After a very short time all units are instrumented. The instrumented files are listed in a log window.

- Recompile the program.

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



Depending on the profiling options the button 'Start' is enabled (No autostart option) or not (with autostart option). With autostart option the measurement starts with the start of the testee. Without the autostart option you have to press the start button in the online operation window when you want to start the measurement, define activating methods or insert calls into your sources for activation or deactivation. See chapter A.10 for the complete description. After the program has ended, you can

View the results of the measurement with the built-in viewer of ProLazaX,

For the Built-in viewer, just start ProLazaX again and click ‘Start viewer’. If the name of your project is not automatically displayed, select it. Then click the ‘Start viewer’ button.

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

A.3 Files created by ProLazaX or the measured program

ProLazaX creates the file ‘proflst.asc’, it contains information about the procedures to be measured. The file profile.ini contains options for the time measurement. The viewer can create a file named ‘*.hst’ if you use the history function (see A.8).

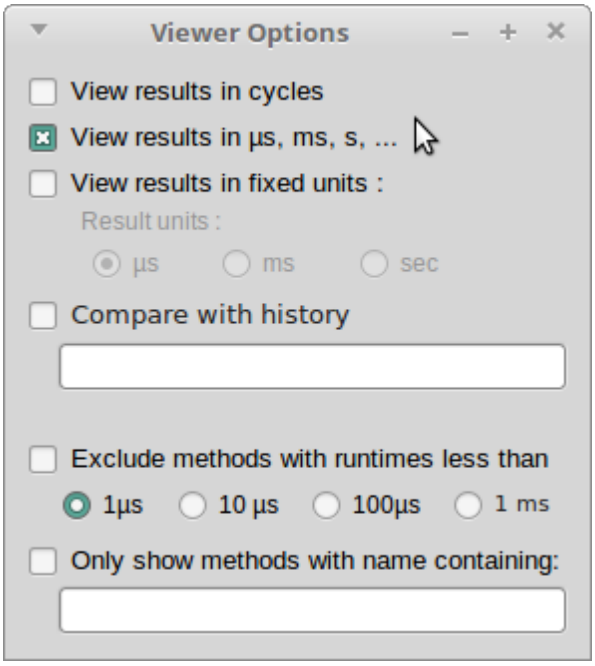
Your compiled program the file with the name ‘progname.txt’ contains the data in the ASCII-semicolon-delimited format for data base export and viewer and ‘progname.tx2’ for the headlines for the different intermediate results (for the built-in viewer). A file ‘programname.tx3’ is stored for the interface to the Lazarus-IDE. The file ‘progname.swo’ with the list of procedures that have to be deactivated for time measurement at next program start is stored optionally. Also a file with the name ‘progname.nev’ is created into which the names of the uncalled methods are stored. This file is also used by the viewer.

All files are stored in the output directory for the executable.

A.4 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 ‘Start viewer’.

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 or by API-calls (see chapter A10.3).

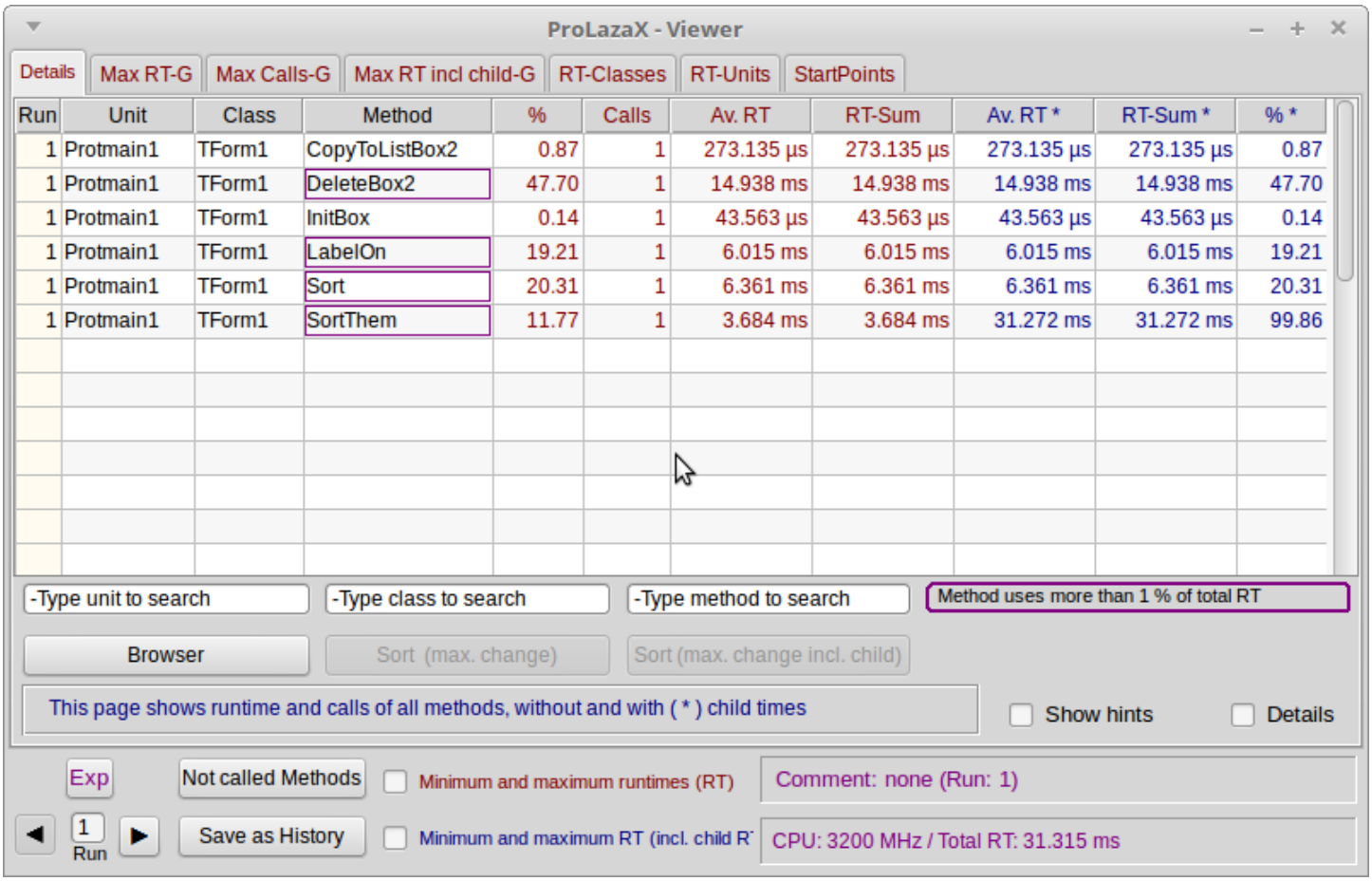


You can choose if you want to view the results in µs, ms... or in CPU-Cycles.

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

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

See next page please.



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

Explanation of this window:

CPU: nnn MHZ giving the CPU - speed
Total RT: ttt giving the run time of all measured methods (alternatively in CPU-cycles)
Comment: ccccc Text set as comment in the online operation window for intermediate results, ‘At finishing application’ when the results were automatically stored when the testee ended or date and time when the online operation window cyclically stored results.

Sorting the table:

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

Navigating through the results:

Navigating 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 LibreOffice or OpenOffice.

The Minimum/Maximum check boxes:

Checking these options, minimum and maximum run times are displayed if they were collected in the measurement (see Chapter A 2. If these checkboxes are disabled, no minimum and maximum values were evaluated.

The History - button: see chapter A.8

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, min or hour units (in the professional mode also minimum and maximum run times can be displayed)
RT-sum	RT * Calls

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	RT * Calls
%	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 ProLazaX-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 graphical representation

The 12 sorted classes that consumed the most run time

The 12 sorted units that consumed the most run time

Meaning of run times inside a red frame:

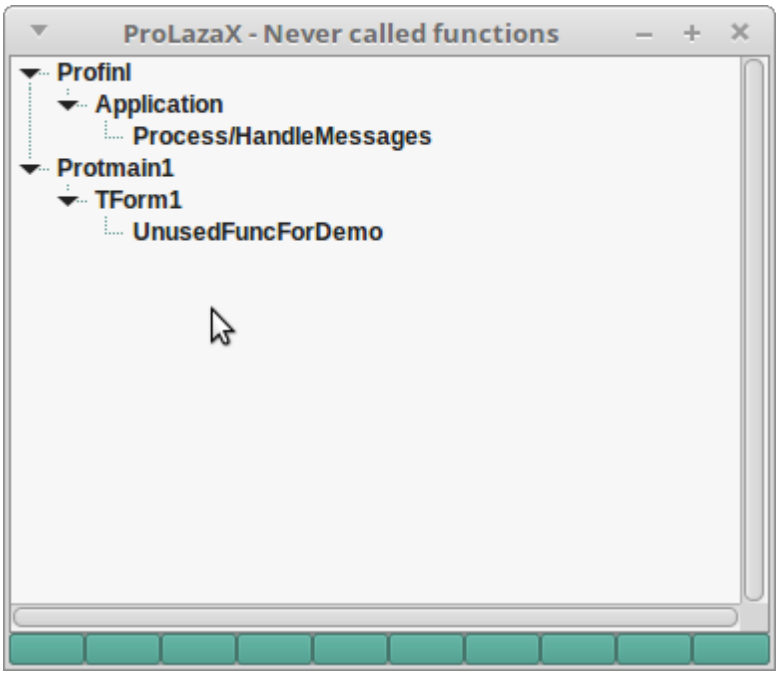
The run time is greater than the time stored in the history file. The frame only then appears, if the change is greater than 1% of the total run time of the application.

Meaning of run times inside a green frame:

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

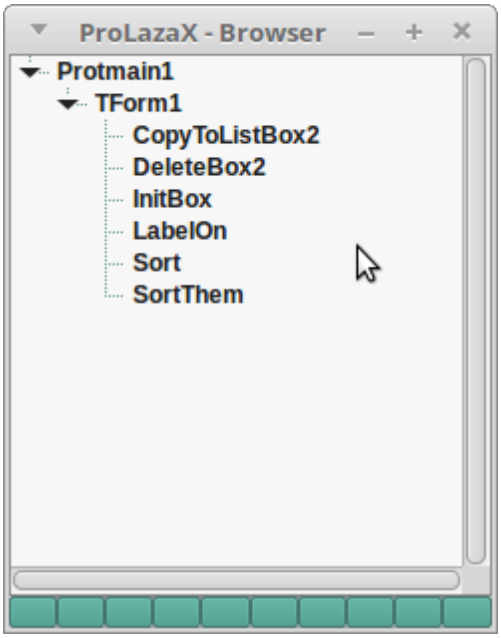
The Not called Methods - button:

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

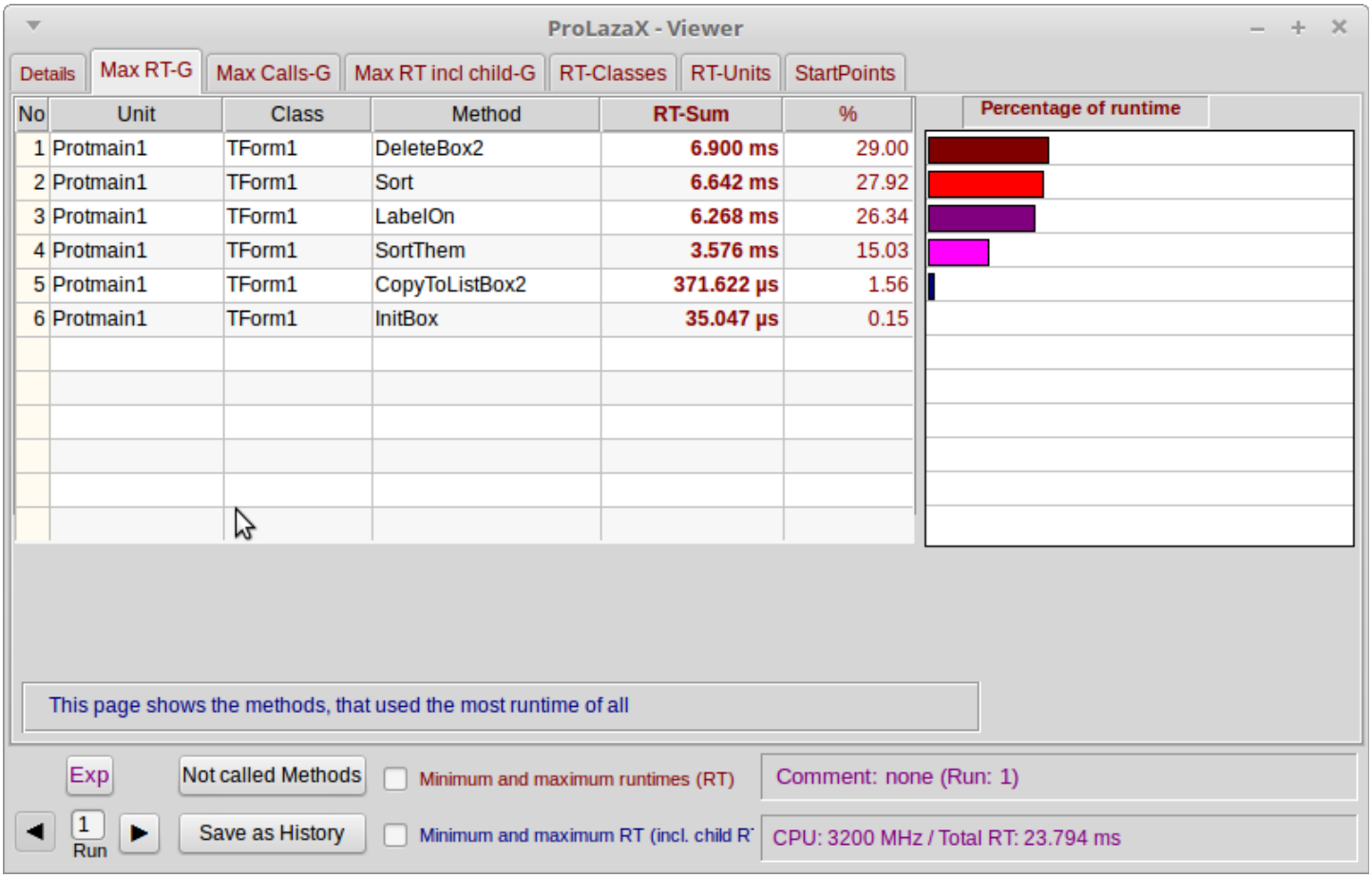


The Browser - button:

It opens a small browser window (similar to the explorer) that shows units, classes and methods in a 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

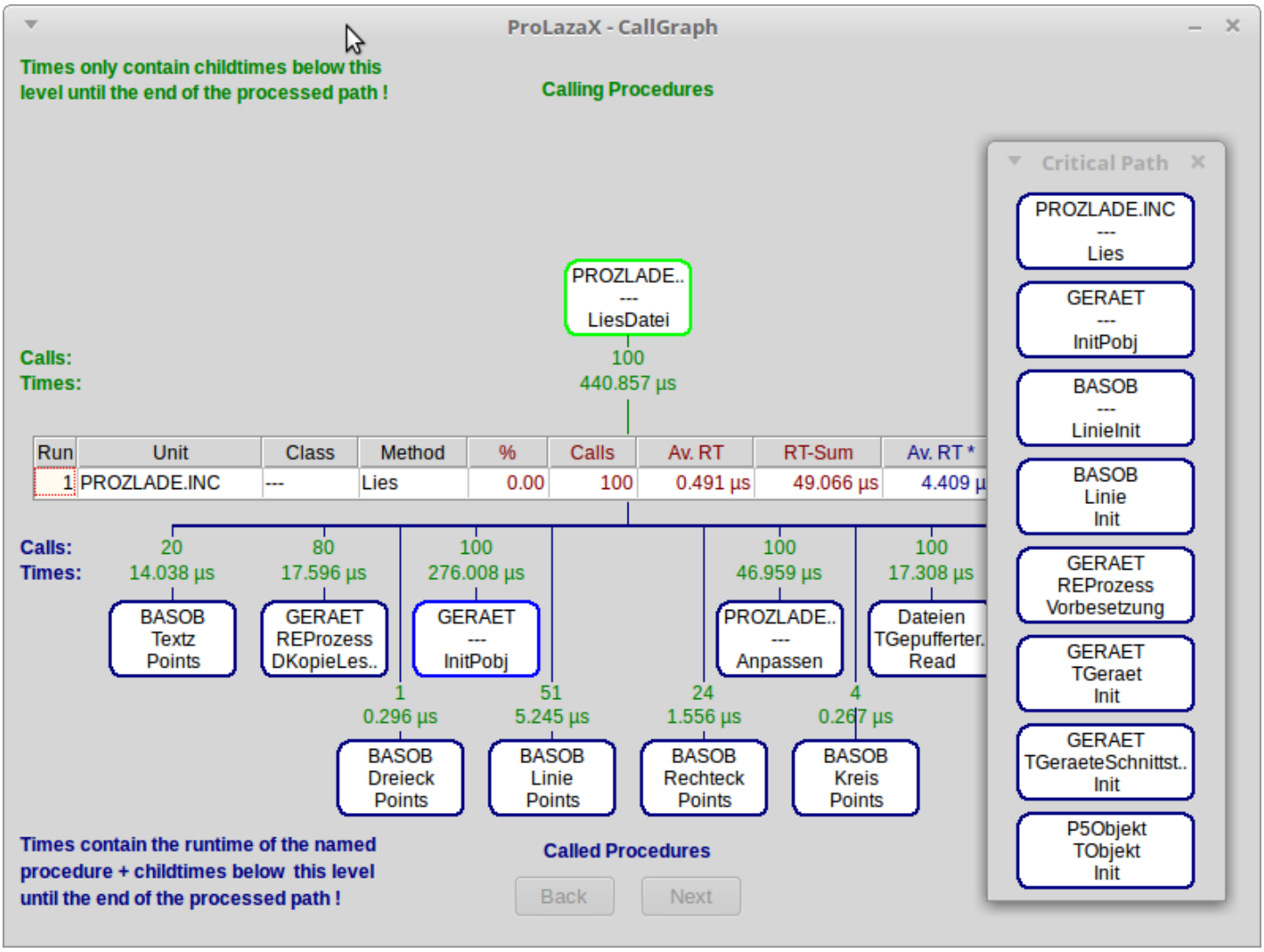


Example of: Maximum run time consuming methods (graphical)

A.5 Using the caller / called graph (call graph)

If the call graph data have been collected when measuring the run time (check box 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 run times 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 are 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 it’s complete measurement results.

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

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

A.6 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 Linux 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 ProLazaX (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:

A.6.1 Common causes of disturbing influences outside of your program

Some disturbers everybody might be aware of:

- activated screen saver,
- Turbo Boost mode activated (Intel Core i3, i5, i7), can be deactivated in some BIOS-versions.
- Intel Speed Step technology activated, can be deactivated in some BIOS-versions.
- Linux power management,
- background schedulers,
- online virus protection,
- automatic recognition of CD changing,
- Linux swap file causes memory transfers of different duration.

These disturbing influences are easy to eliminate.

A.6.2 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 ProLazaX:

- 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 ProLazaX but start measurement at the starting point of a certain action. How to exclude methods is described in Chapter A9.1, how to measure defined actions only is described in chapter A9.2.

A.6.3 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, even when exactly the same code under the same conditions is executed. When measurements have to be compared, these features should be deactivated. Some PC’s have this possibility in their BIOS.

A.6.4 Common cause of disturbing influence is the PC's processor cache

The influence of the cache can't be easily excluded. The only way is to produce exactly the same sequence of events two times every measurement and to start measurement with starting the second sequence by the programming API, switch it off at the end of the second sequence and store the measured data to disk (also by the ProLazaX 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 ProLazaX), if an optimization has decreased the runtime or not.

A.6.5 Mobile PC's

Mobile PC's cause a 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: ProLazaX measures CPU cycles. If we look to the CPU - cycles displayed in the viewer, the measurement is correct. If times are displayed, it could be that too long or too short times are displayed. It depends on the CPU speed that was set when the CPU speed was measured. Different processors use different algorism to change the speed. The only way to get 100% correct results is to switch off the power safe mode.

A.6.6 Summary

If you eliminate the disturbances mentioned in A.3.1 / A.3.2 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.

A.7 Interactive optimization

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

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

There are different ways of comparing the measurement results:
- to use the viewer and print the measurement results or
- to use ProLazaX's history function.

A.8 The history function

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

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

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

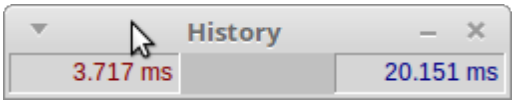
To get the cell coloured, the methods change of runtime must be essential. Essential means, it must have changed so much, that it influenced the programs runtime by 1 % or more.

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

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

A.8.1 Practical use of the history function

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



A.9 Measuring parts of the program

A.9.1 Exclusion of parts of the program

All Linux programs are message driven. So, if you define a function, that, for instance, handles mouse moves, ProLazaX 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 ProLazaX: 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. 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 !!!**

2. Exclusion of complete units

   - Enable write protection for the units not to compile or
   - insert the following statement before the first line of the unit:

   //PROFILE-NO

3. Exclusion of functions

   Before profiling insert statements before and after the procedures that have to be excluded to switch off the vaccination by ProLazaX:

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

A.9.2 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 is much easier to see, which function should be optimized.

There is one way for dynamical activation of measurement in ProLazaX:

1. By inserting a special comment into the source code

    Insert the comment: //PROFILE-ACTIVATE right before the procedure which shall start the measurement. When the procedure is entered, the measurement starts, when the procedure exits, the measurement stops. This comment is not deleted, when the sources are cleaned.

2. By using API-calls.

    This method is described in the next chapter.

For both methods automatic start should not be activated.

A.9.3 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 Linux 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 start 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, Linux 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.9.4 Measuring specified parts of procedures

For the case of very large procedures sometimes it might be interesting to know which *part* of it consumed the most run time. One way to find this out is to restructure the procedure into neat parts or to divide it up by means of local procedures. Another idea would be that ProLazaX 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 to risky, ProLazaX 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 ProLazaX to insert measurement statements right after the comments. The runtime measured in this so defined block will be found in the viewer by searching for ‘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 runtime 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 runtime displayed by the viewer would be 18 ms (displayed in the line for the procedure DoSomething).

The same example with measuring part-b separately:

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

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

It might be that the results are not exactly the same because the processor cache is used in a different way, especially processors with a small cache have the problem, that not the whole procedure inclusive measurement parts of ProLazaX 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.

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

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

A.10 Programming API

A.10.1 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}
   SortAll;  // the method of which you want to know the runtime
{$IFDEF PROFILE}finally; call ProfExit; end; {$ENDIF}
END;
```

You can modify the code in three different ways:

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

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

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

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

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

A.10.2 Preventing to measure idle times

Some Linux-API functions and Lazarus functions interrupt the calling procedure and set the program into an idle mode. A well-known example is the Linux-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 Linux-API calls and some Lazarus-calls are replaced automatically by the Unit ‘Profinl.pas’. For the above named example MessageBox, there is a re-definition. It automatically interrupts the counting of CPU-cycles for the calling procedure only and reactivates it after returning from Linux.

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 ProLazaX-API-calls StopCounting and ContinueCounting. In chapter A.14 you can find the list of calls, which are redefined in the unit ‘Profinl.pas’. They automatically call these functions before using the original Linux- 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:

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

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

{$IFDEF      PROFILE}ProfInl.ContinueCounting; {$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.10.3 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

```
{$IFDEF     PROFILE}ProfInl.ProfAppendResults(FALSE); {$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.12).

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 runtime of the current call will be zero.

A.11 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.11.1 Code instrumenting options:

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

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.

A.11.2 Runtime measurement options

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

Count Inherited execution time for caller

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.

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

The online operation window will be displayed. So intermediate measurement results can be stored.

Program uses 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.

But measure main thread only

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

Store also minimum and maximum runtimes

If this option is checked, the measurement routines of ProLazaX 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.11.3 Measurement activation options

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

Auto start of measurement

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 API-Calls or online operation window

If Auto start is not checked, this needs to start measurement by API-Call, by online-operation window or by entering defined methods (which is activated by inserting comments //PROFILE-ACTIVATE before the method implementation).

(see chapter A.10, A.12 and A.9.2 for details)

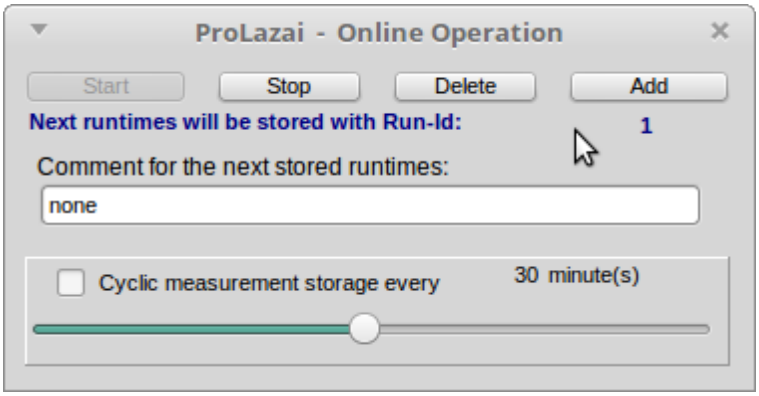
A.11.4 General options

Lazarus version

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

A.12 Online operation window

With the online-operation window



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

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

The default value for the headline for intermediate results is:

'none'.

Also an automatic and cyclic storing of 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.13 Libraries and packages

Libraries and packages are not supported.

A.14 Treatment of special Linux- 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 ‘Profinl.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 Linux Sleep function (defined in ‘Profinl.pas):

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

Because of this redefinition, the Profinl-unit must be named after the unit Dialogs. This is normally done. The only exception is, if you name this unit in the implementation part of the unit, Lazarus itself places them into the interface part.

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

A.14.1 Redefined Lazarus-API functions

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

A.14.2 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.15 Conditional compilation

Conditional compilation is fully supported.

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:

{$IF const > x }	always evaluated as TRUE	comparison with a constant
{$IF SizeOf(Integer) > 10}	always 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.16 Measuring on a customer PC

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

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

For ProLazaX:

- libprofmeas.so
- Proflst.asc
- Profile.ini

For ProLazaX64:

- libprofmeas64.so
- Proflst.asc
- Profile.ini

All files are stored in the exe-file directory of the application to be measured on the development PC. The libraries have to be copied into the /usr/lib directory.

B. Limitations of use

B.1 General

Console applications have no online operation window.

Procedures in a LPR-file can not be measured.

The measured times are 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 multi processor machine the results might differ even more.

If your PC-processor has the feature 'Turbo Boost Mode' or 'Intel Speed Step Technology', you will see that every measurement has strong differing results from the previous one without having changed the measured app. If your BIOS allows to deactivate these features, you should do it while optimizing

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

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

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

A problem for measurement is Linux 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 Athlon 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.

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 Linux. Sometimes Linux writes into the cache sometimes directly to the disk.

B.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.

B.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 !!!

B.4 Assembler Code

Pure Assembler procedures and functions are not supported.

B.5 Modifying code instrumented by ProLazaX

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 ProLazaX profile your code again. It is NOT necessary to delete the old statements inserted by ProLazaX before.

B.6 Hidden performance losses / Tips for optimization

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

Below three examples for this.

- Passing Parameters:

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

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

- Local variables:

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

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

- GoTo statements

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

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

 Final:

END;
```

C. Error messages

In case of errors an error message is displayed by ProLazaX at the bottom line of its window (e.g. file-I/O-errors) and in the profiling log. 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 *.pas_sav (or *.lpr into *.lpr_sav and *.inc into *.inc_sav),

- after that the renamed file is parsed and instrumented, the output is stored into a *.pas-file (or *.lpr / *.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 / *.lpr / *.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.

D. Security aspects

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

E. Cleaning the sources

If you want to delete all lines that ProLazaX 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 ProLazaX 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).

F. Compatibility

ProLazaX is tested with 32 and 64 bit UBUNTU versions (Linux Mint). There is no guaranty that it runs with other Linux versions. A Pentium compatible processor is necessary.

G. Installation of ProLazaX

The installation of ProLaza is simply done by copying all files of the delivery to an appropriate place of the disk without changing the directory structure. The libraries need to be copied to /usr/lib: libprofmeas.so for 32 Bit Linux, libprofmeas64.so for 64 Bit Linux.

If ProLaza has to be started with the tools menu, the file ProLazaX or ProLazaX64 has to be entered as file name, the working directory should be the directory where ProLazaX is stored. As parameters $ProjFile()$SaveAll() should be entered.

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

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

I. Updating / Upgrading of ProLazaX

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

J. Author

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

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

K. History

Version 1.0 : 9/97	First release of ProDelphi
Version 1.0 : 11/12	Adaption of ProDelphi to Lazarus and first release of ProLaza for Linux.
Version 1.0 : 11/17	Porting of ProLaza as ProLazaX for Linux

L. 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).