# ProKylix User Guide

## ( ProDelphi for Linux, release 2.4 )



## The Profiler for Kylix  (for Pentium and compatible CPU's)

### Profiling

The purpose of ProKylix is to find out which parts of a program consume the most CPU-time. Because Borland (Inprise, Corel or whowever) gave up the profiler for 32-bit applications, a new tool had to be created. ProKylix with it's comfortable viewer, browser, history and programmers API meanwhile is more than the legendary Tubo Profiler. The viewer with it's sorted results enables the user to find the bottle necks of his program very fast. The history function shows the user, if a preceeding optimization was successful or not. ProKylix's outstanding granularity makes it possible even to optimize time critical procedures. The built-in calibration routine adapts the measurement routines to the used processor and guaranties results that do not include measurement overhead.

### Post Mortem Review

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

### Differences between the freeware mode and the professional mode

In the freeware mode up to 30 procedures can be measured or tracked, in the professional mode 32000.

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

**Date: 5/31/2002**

# 0. Contents of this description


A. Principle of Profiling

A1. How to profile
A1.1 Files created by ProKylix or the measured program
A1.2 Checking the results with the Built in viewer
A1.3 Emulation of a faster or slower PC
A1.4 Checking the results by viewing the ASCII-file

A2 Getting exact results
A2.1 Common causes of disturbing influences outside of your program
A2.2 Common causes of disturbing influences inside your program
A2.3 Common cause of disturbing influence is the PC's cache
A2.4 Summary

A3 Interactive optimization
A3.1 The history function
A3.2 Practical use of the history function

A4 Measuring only parts of the program
A4.1 Exclusion of Parts of the program
A4.2 Dynamic activation of measurement
A4.3 Measuring specified parts of procedures

A5 Programming API
A5.1 Measuring defined program actions through Activation and Deactivation
A5.2 Preventing to measure idle times
A5.3 Programmed storing of measurement results

A6. Options for profiling
A6.1 Code intrumenting options:
A6.2 Runtime measurement options
A6.3 Measurement activation options

A7. Online operation of the profiled program

A8. Profiling dynamic link libraries (DLL)

A9 Treatment of special Linux- and Kylix-API-functions
A9.1 Redefined Linux-API functions
A9.2 Redefined Kylix-API functions
A9.3 Replaced Kylix-API functions
A9.4 Not replaced or redefined Kylix functions

A10 Conditional compilation

A11. Limitations of use

A12. Assembler code

A13. Modifying code vaccinated by ProKylix

A14. Error messages

A15 Security aspects

Appendices:

B. Post mortem review

C. Cleaning the sources

D. Compatibility

E. Installation of ProKylix

F. Description of the result file (data base export)

G. Updating / Upgrading ProKylix

H. How to order the registration key for unlocking the Professional mode

I. Author

J. History

K. Literature

## BEFORE using ProKylix practically, please read Chapter 15 carefully !!!

# A. Principle of Profiling

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

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

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

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

The **third** file (programname.nev) contains the names of all methods which have never been called when measuring the runtime of your program. It is used 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 ProKylix.

The **fourth** file is optional and contains the measurement results in a format, that can be printed in landscape format or can be viewed by Kylix. It is described in chapter A1.3.

The **fifth** file is also optional and only created, if the automatic switching off is activated (see A5).

# A1 How to profile

Using ProKylix is quite simple. The windows version (ProDelphi) has been used in a project with a large program, which now already contains more than 420 000 lines of code written by 14 programmers. After more than two years of developping the program has been optimized with the help of ProKylix. The programs runtime for processing process messages could be decreased by 50 %.

**Use the Setup-program to install ProKylix. It will do all the necessary things that you can start ProKylix by the Kylix tools menu. It also installs two programs to test the accuracy of ProKylix and installs a start script for ProKylix. This startscript (startprofiler) is only then necessary, when you want to start ProKylix without the tools menu. To start the Setup program you need to use the starting script 'startsetup'. It needs the installation path of Kylix as parameter.**

**Example: /home/user/startsetup /home/carol/Kylix**

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

*Copy the files Profadjx.dcu, Profcbrx.dcu, Profini.dcu, Proftimx.dcu, Profonlix.dcu and Profonlix.xfm into the Kylix LIB-directory. Install ProKylix by 'Tools / Configure Tools' into the Kylix-Tools-Menu. You need to enter following items:*

| | |
|---|---|
| *Title:* | *ProKylix* |
| *Program:* | */installation-directory/Profiler* |
| *Working directory:* | */installation-directory/* |
| *Parameters:* | *$SAVEALL $EXENAME /D1* |

*Together with the distribution files you get two additional programs. One is a program that measures the runtimes of a few procedures. The other is in principle the same program without the measuring instructions, it's purpose is to show the measurement accuracy of ProKylix. You should move the files into special directories. E.g., you could create a directory 'DONT-PROFILE' and copy the files Ptcalx.pas, Ptmainx.pas, Ptest.dpr and Ptmain.xfm 'PROFILE-ME' into that directory. The other directory you could e.g. name PROFILE-ME and copy the files Ptcalx2.pas, Ptmainx2.pas, Ptest2.dpr and Ptmain2.xfm into that directory.*

*The startscript for starting ProKylix has to be modified manually. You can do this by using a text editor. The necessary entries are named in the comment inside the script.*

After installation, try to compile your program to create the Kylix project files (the kof-file is needed by ProKylix). **If no kof-file exists, all files have to be in the same directory (*.pas, *.inc, *.xfm and *.dpr).**

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

**Don' t use the original units for profiling, maybe ProKylix still contains bugs. Just make a security copy of the program to be measured, e.g. by archiving all pas-, dpr- and inc-files.**

For profiling your sources perform the following steps:

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

- Deactivate the Optimization option.

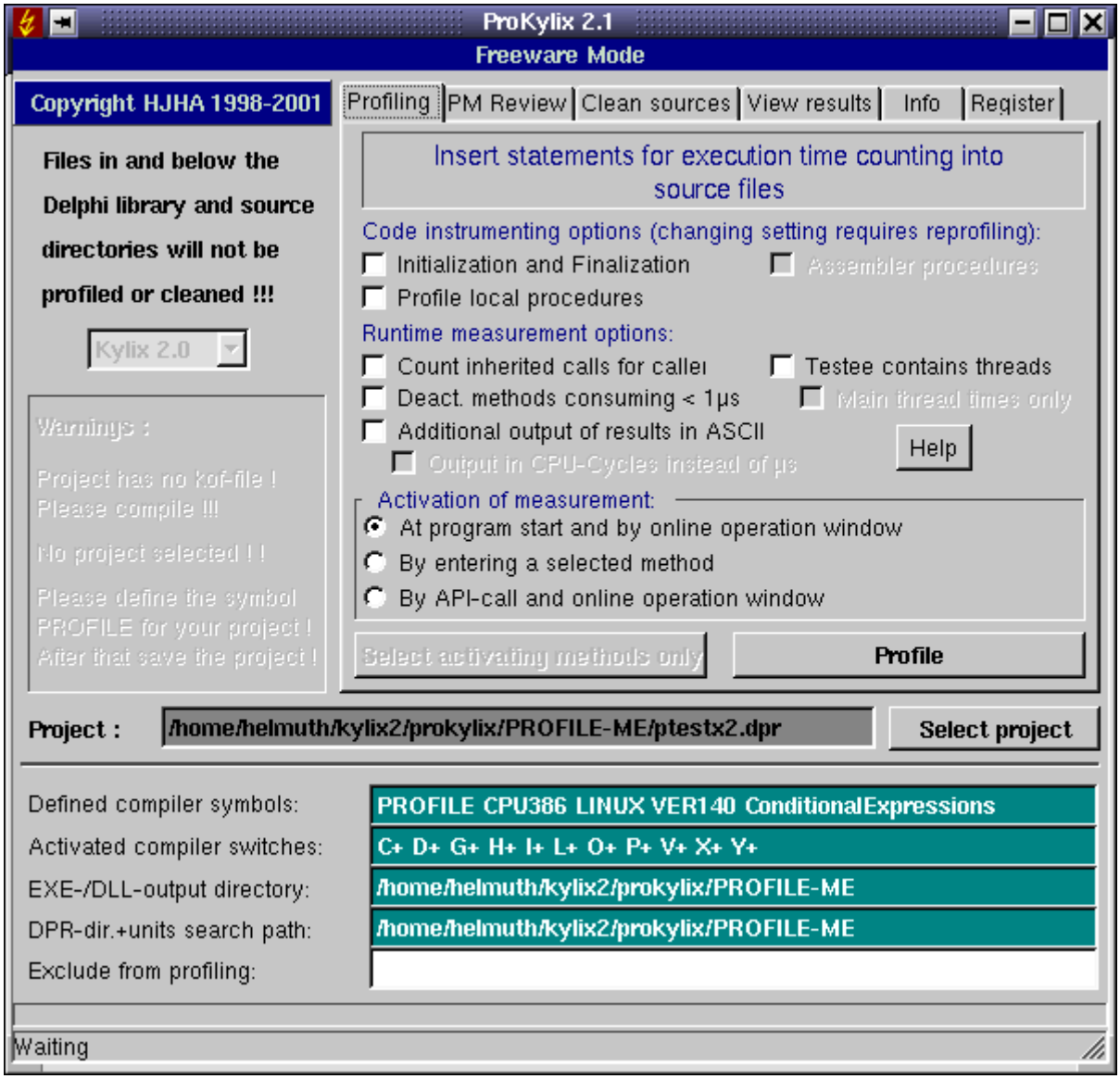
- Optionally deactivate all runtime checks.

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

- Start ProKylix from the Kylix tools menu or somehow else.

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

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




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

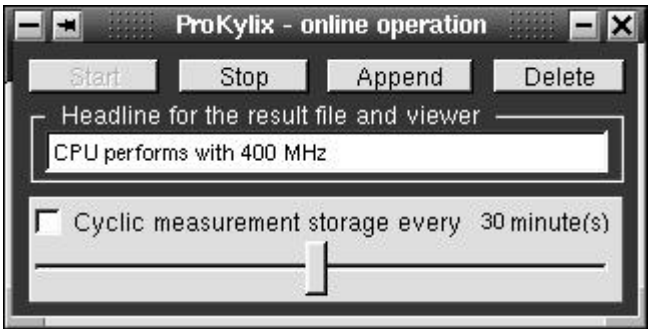
- Click the Profile-button. After a very short time all units are vaccinated. The vaccinated files are listed in a log window.

- Recompile the program.

**Files in and below the Kylix 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 testee. Without the autostart option you have to press the start button in the online operation window when you want to start the measurement, define activating methods or insert calls into your sources for activation or deactivation. See chapter A5 for the complete description. After the program has ended, you can view the results of the measurement

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

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

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

and maybe print it with Kylix.

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

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

## A1.1 Files created by ProKylix or the measured program

ProKylix creates the file ‘proflst.asc’, it contains information about the procedures to be measured for profiling or traced for post mortem review. The file profile.ini contains options for the time measurement and the last screen coordinates of the online operation window. The viewer will create the file ‘progname.hst’ if you use the history function (see A3).

Your compiled program creates ‘progname.ben’, it contains the results of the time measurement in a printable format if you have checked 'Additional output in ASCII' when profiling. The file with the name ‘progname.txt’ contains the data in the ASCII-semicolon-delimited format for data base export and ‘progname.tx2’ for the headlines for the different runs (for the built-in viewer). It creates ‘progname.swo’ with the list of procedures that have to be deactivated for time measurement at next program start. If creates also a file with the name ‘progname.nev’ into which the names of the uncalled methods are stored. This file is also used by the viewer.

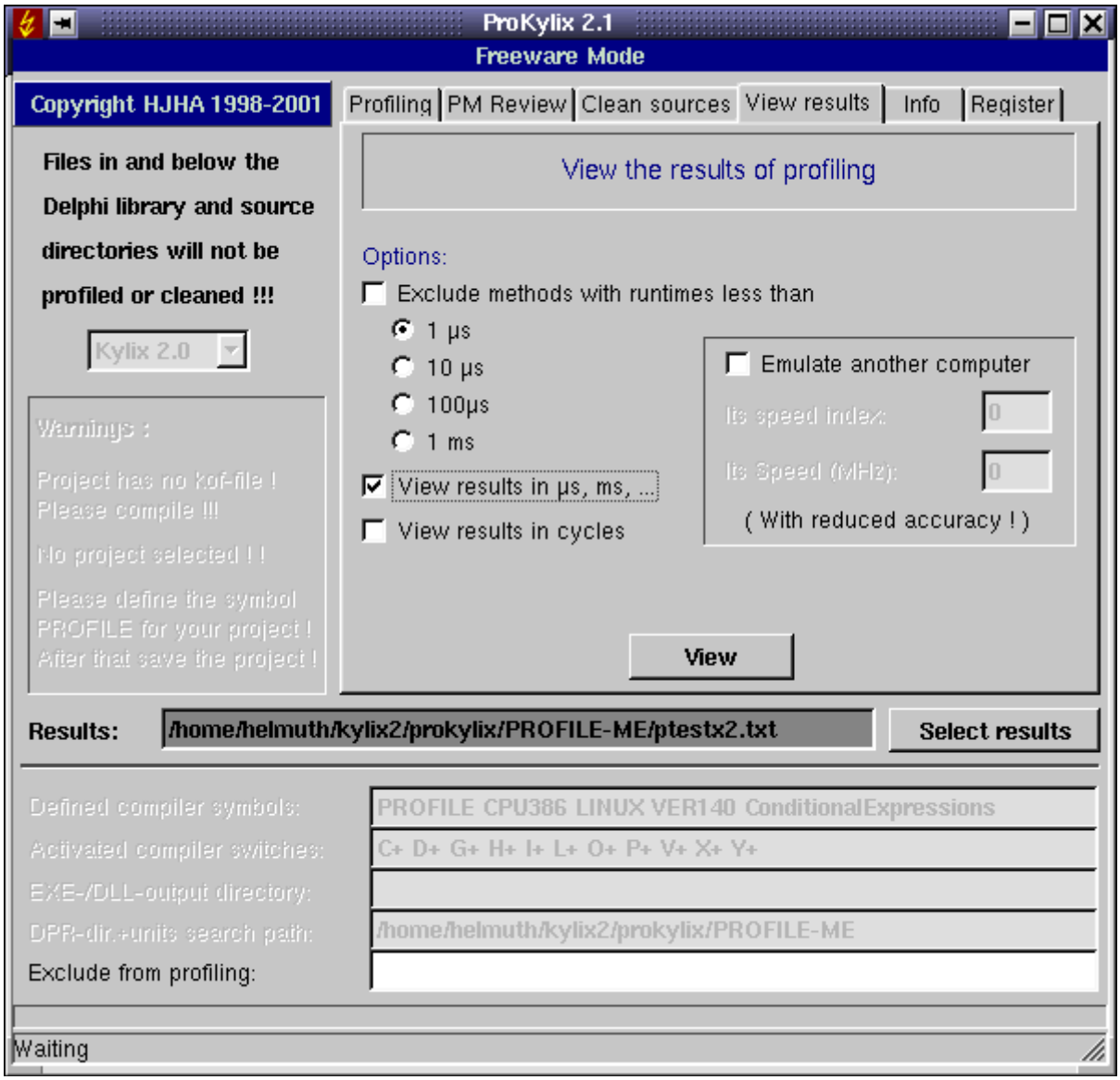
Your compiled program creates a file named ‘progname.pmr’ in case you have selected post mortem review and an exception occured and was trapped. It contents the call stack.

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

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

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

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



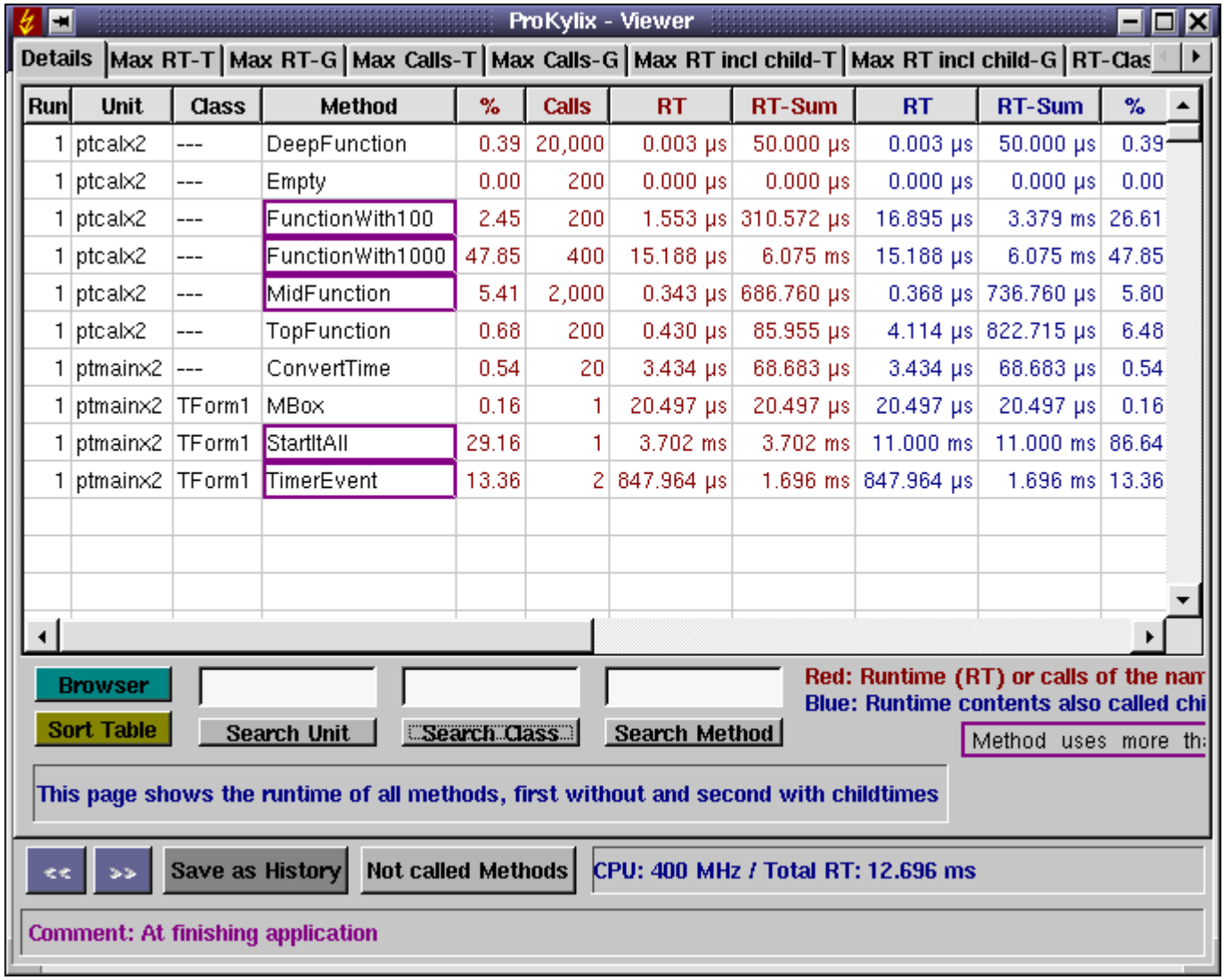

You can choose if you want to view the results in µs, ms ... or in CPU-Cycles.

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

Also you can emulate (recalculate) the measurements for a faster or slower PC. No need to install the IDE on that PC, just enter two constants in an edit field and let ProKylix tell you how fast or how slow your program would perform on that PC (see chapter 1.3).

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

**See next page please.**




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

**Explanation of this window:**

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

**The Sort Table - button:**

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

**The History - button:** see chapter A3

Meaning of **Run**:

Any time the program stores data into the result file, it puts a leading number before the measured times: the number of the measurement. With the << (Previous)- or >> (Next)- button you can switch between different measurements. At the next run of the program the counting starts at 1 again.

Meaning of the RED columns:

| | |
|---|---|
| % | Percentage of the total runtime the procedure took without their child procedures |
| Calls | How often the procedure was called |
| RT | Average runtime of the procedure in CPU-cycles or in µs, ms, sec or hour units |
| RT-sum | RT * Calls |

Meaning of the BLUE columns:

| | |
|---|---|
| RT | Average runtime of the procedure inclusive its child procedures in CPU-cycles or in µs, ms, ... |
| RT-sum | RT * Calls |
| % | Percentage of the total runtime the procedure took inclusive her child procedures. |

Meaning of the <<-Button and the >>-Button:

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

Meaning of **'Comment':**

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

The other availlable pages show:

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

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

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

The 12 sorted lasses that consumed the most runtime

The 12 sorted units that consumed the most runtime

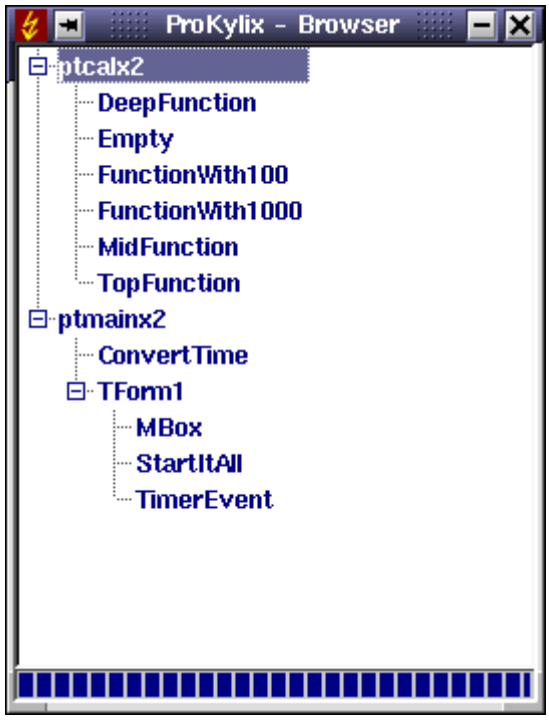
### The Not called Methods - button:

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

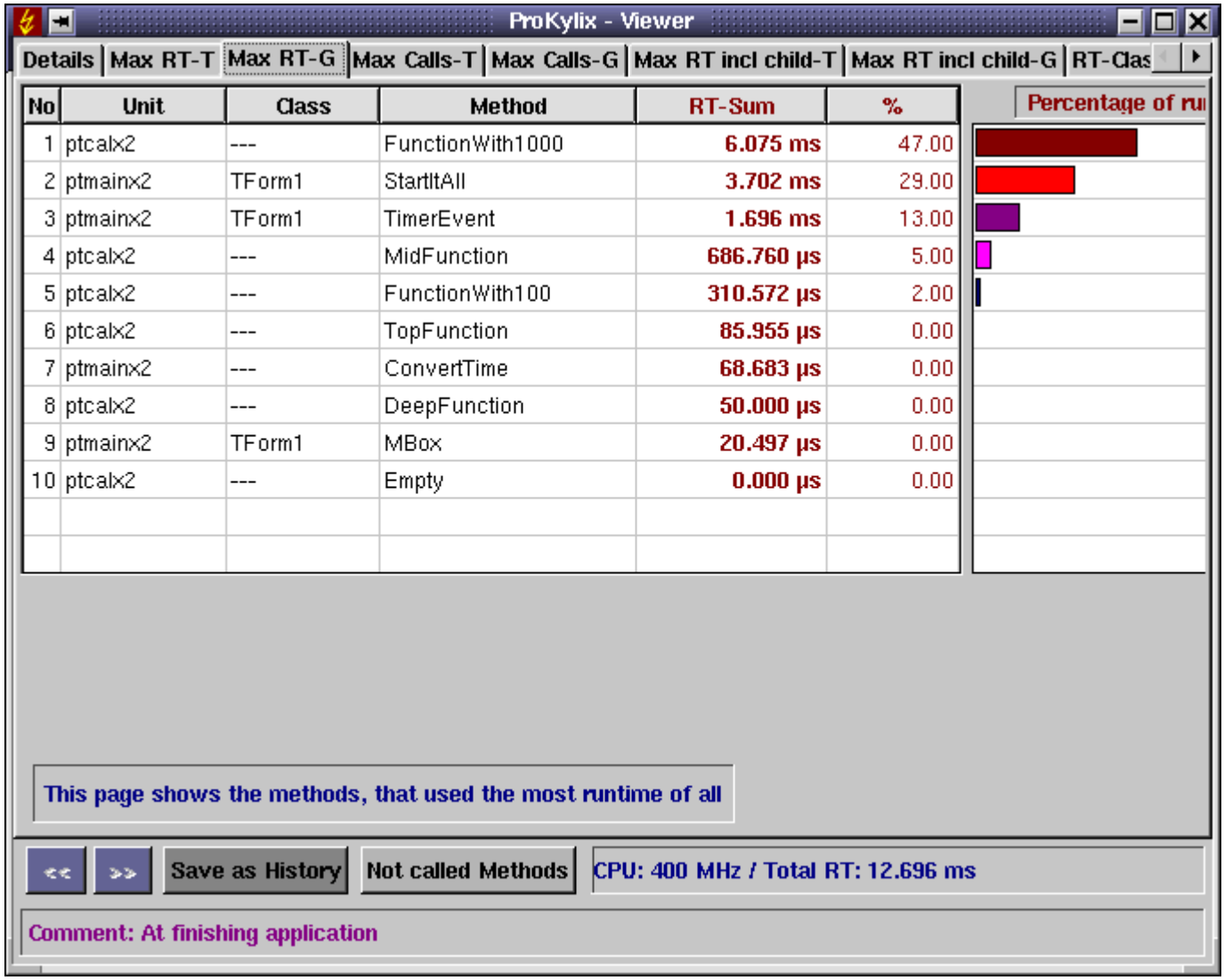
See next page, please.


**The Browse - button:**

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




**See next page please for another viewer window example**




Example of: Maximum run time consuming methods (graphical)

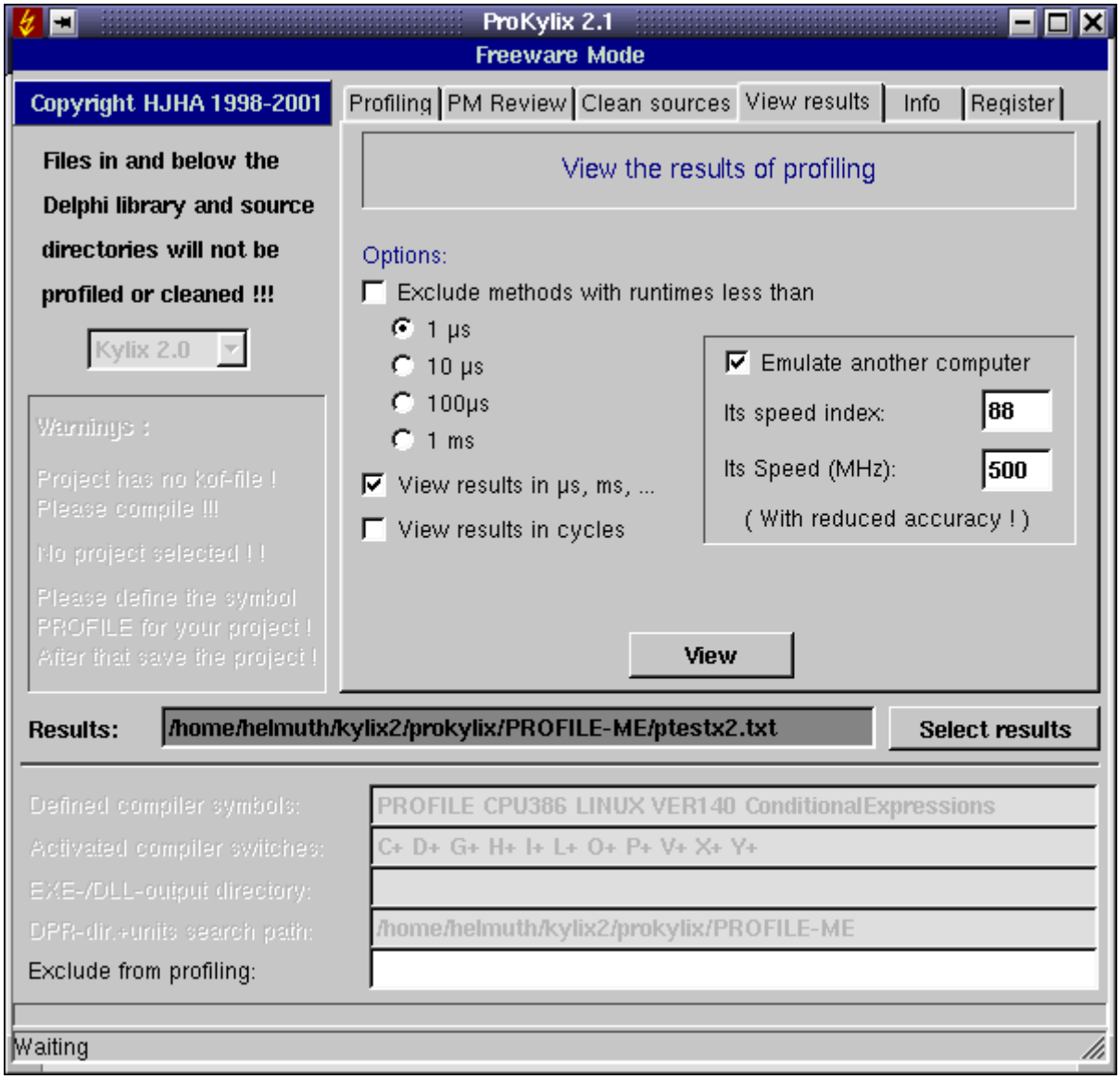
## A1.3 Emulation of a faster or slower PC

If you want to know, how fast (or slow) your program would perform on another PC, just use the program Getspeed to get the other PC's speed index, enter it in ProKylix, 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 measured with Getspeed and the clockrate of the PC to be emulated, has to be entered in the view results form (see below).

***This function is currently not supported in ProKylix (only in ProDelphi).***


ProKylix 2.1
Freeware Mode
Copyright HJHA 1998-2001
Profiling
PM Review
Clean sources
View results
Info
Register
Files in and below the Delphi library and source directories will not be profiled or cleaned !!!
Kylix 2.0
Warnings :
Project has no kof-file ! Please compile !!!
No project selected ! !
Please define the symbol PROFILE for your project ! After that save the project !
View the results of profiling
Options:
Exclude methods with runtimes less than
1 µs
10 µs
100µs
1 ms
View results in µs, ms, ...
View results in cycles
Emulate another computer
Its speed index:
88
Its Speed (MHz):
500
( With reduced accuracy ! )
View
Results:
/home/helmuth/kylix2/prokylix/PROFILE-ME/ptestx2.txt
Select results
Defined compiler symbols:
PROFILE CPU386 LINUX VER140 ConditionalExpressions
Activated compiler switches:
C+ D+ G+ H+ I+ L+ O+ P+ V+ X+ Y+
EXE-/DLL-output directory:
DPR-dir.+units search path:
/home/helmuth/kylix2/prokylix/PROFILE-ME
Exclude from profiling:
Waiting

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

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

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

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

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

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

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

**c. Summary**

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

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

**e. The total runtime of the tested program (= the sum of all measured procedures)**
For a quick test, points a. and b. can be disabled.

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

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

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

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

# A2 Getting exact results

If you measure program runtimes a few times, you will see that the measurement results differ from measurement to measurement with out that you have changed your sources. Two kind of results will offen differ: the runtime of a method and the percentage of their runtime of the complete program. The reasons are :

- there are events that disturb the measurement, e.g. programs running in the background.
- you measure methods which are activated by 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 ProKylix (1 CPU-cycle), you see these differences.

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

## A2.1 Common causes of disturbing influences outside of your program

Some disturbers everybody might be aware of:

- activated screen saver,
- Linux power management,
- background schedulers,
- online virus protection,
- automatic recognition of CD changing.

*These disturbing influences are easy to eliminate.*

## A2.2 Common causes of disturbing influences inside your program

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

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

*The three influences are also easy to eliminate.* You only need to exclude these procedures from measurement. Another way is not to use the autostart function of ProKylix but start measurement at the starting point of a certain action. How to exclude methods is described in Chapter A4, how to measure defined actions only is described in chapter A5.

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

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

## A2.4 Summary

If you eliminate the disturbances mentioned in A2.1 / A2.2 and measure defined actions, you will see the differences between two measurements is very low, most times only a few CPU-cycles. Larger differences appear only when neasuring procedures with disk transfers. A good trick is, to use the second measurement for comparison with later optimizations, specially when the disk transfer is a reading transfer. The first run of the program will get the most data into disk cache, the second measurement reads the data from cache.

## A3 Interactive optimization

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

***Important is, which method is worth to be optimized: A method, that uses 10 % runtime must be optimized by 10 % to decrease the total program runtime by 1 % !!!***

There are different ways of comparing the measurement results:

- to use the ASCII-output and print it,
- to use the viewer and make screen dumps or
- to use ProKylix's history function.

### A3.1 The history function

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

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

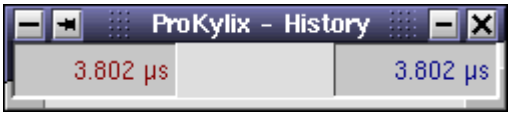
If you have stored the results on disk, the next time you open the viewer window, automatically the history stored before is read and compared with the actual measurement. By colouring the cells of the viewers table, you have a quick overview about all changes of runtime: Red means method got slower,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

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

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

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

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



# A4 Measuring only parts of the program

## A4.1 Exclusion of Parts of the program

If you define a function that, for instance, handles mouse moves, ProKylix 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 ProKylix: all procedures are measured, the measurement starts with the start of the program (if option 'Activation of measurement / At program start' is checked).

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

There are different ways of excluding parts of the program:

1. Files in and below the Kylix lib- and source- directories are always excluded.

2. 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**

3. Exclusion of functions

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

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

4. Automatic exclusion

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

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

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

## A4.2 Dynamic activation of measurement

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

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

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

1. By dialog

   In the main window of ProKylix 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 ProKylix. You can optionally select further activating methods, but it is not necessary.

3. By using API-calls.

   This method is described in the next chapter. It is the only way versions of ProKylix earlier than 8.0 could handle this problem. In principle, this way can still be used, but it is not very comfortable. Using that third method you always need to insert two calls, one for activation and one for deactivation.

## A4.3 Measuring specified parts of procedures

For the case of very large procedures sometimes it might be interesting to know which *part* of it consumed the most run time. One way to find this out is to restructure the procedure into neat parts or to devide it up by means of local procedures. Another idea would be that ProDelphi would measure each block of a structure and not the whole procedure. The last solution would cost a lot of measurement overhead and would make timecritical applications stop working.
For the case that both solutions given is too much work or to risky, ProDelphi has the feature of defining blocks to measure.

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

Just insert this line before the block to measure:

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

and this one behind it:

```
//PROFILE-END
```

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

Using this feature is only possible when taking care to insert these statements so, that the block structure of the program

remains unchanged. E.g. it is not possible to insert the statement into an ELSE-part without BEGIN and END, this would cause compiler errors.

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

Example:

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

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

The same example with measuring part-b separately:

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

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

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

Remark:

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

Example for nesting:

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

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

# A5 Programming API

## A5.1 Measuring defined program actions through Activation and Deactivation

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

***Example1:***

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

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

You can modify the code in three different ways:

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

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

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

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

***Be sure that you use more than one space between $IFDEF and PROFILE you inserted, otherwise the statements will be deleted the next time that the source code is vaccinated by ProKylix. 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
{$IFDEF      PROFILE}Proftimx.ProfActivate; {$ENDIF}
END;

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

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

## A5.2 Preventing to measure idle times

Some Linux-API functions and Kylix functions interrupt the calling procedure and set the program into an idle mode. A well-known example is the Kylix-call Application.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 Kylix-calls are replaced automatically by the Unit 'Proftimx.pas'. Proftimx automatically interrupts the counting of CPU-cycles for the calling procedure only and reactivates it after returning from Kylix runtime library.

To make this possible, there are the ProKylix-API-calls StopCounting and ContinueCounting. In chapter A9 you can find the list of calls, which are redefined in the unit 'Proftimx.pas'. They automatically call these functions before using the original Kylix calls.

**Among all Linux-API-calls only the Sleep-functions are handled yet, others will follow in a later version.**

Some functions cannot be replaced by 'Proftimx.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:

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

      Object.IdleModeSettingMethod;

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

***Important:***

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

### A5.3 Programmed storing of measurement results

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

Just insert the statement

```
{$IFDEF     PROFILE}Proftimx.ProfAppendResults; {$ENDIF }
```

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

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

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

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

into your source.

Another way to produce intermediate results is to use the *online operation window*. Any time you click on the 'Append'-button the actual measurement values are appended to the result file and all result counters are set to zero (see chapter A5 also).

***Important:***

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

## A6 Options for profiling

Profiling options are divided into three groups:

- Code instrumenting options (or vaccination options): How and what to vaccinate.
- Runtime measurement options: How to measure and what to do the results.
- Activation of measurement: Where or when to start measuring runtimes.

### A6.1 Code intrumenting options:

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

***Profile Assembler procedures***

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

***Initialization and finalization***

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

***Profile local procedures***

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

## A6.2 Runtime measurement options

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

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

with the sub-option:

- Output in CPU-Cycles instead of µSeconds

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

***Deactivate functions consuming < 1 µS***

Any time the measurement results are stored in the result file, those procedures that are called at least ten times and consume less then 1 µS are deactivated for the future. The deactivated functions are stored in the file 'ProgramName.swo' for the next run.

***Inherited for parent***

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

Normally times are measured separate for each procedure. Use this method if you want, that, if a method calls a method with the same name of an upper class (e.g. by INHERITED), the time of the inherited method is counted for the calling method.

***Testee contains threads***

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

***Main thread only***

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

## A6.3 Measurement activation options

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

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

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

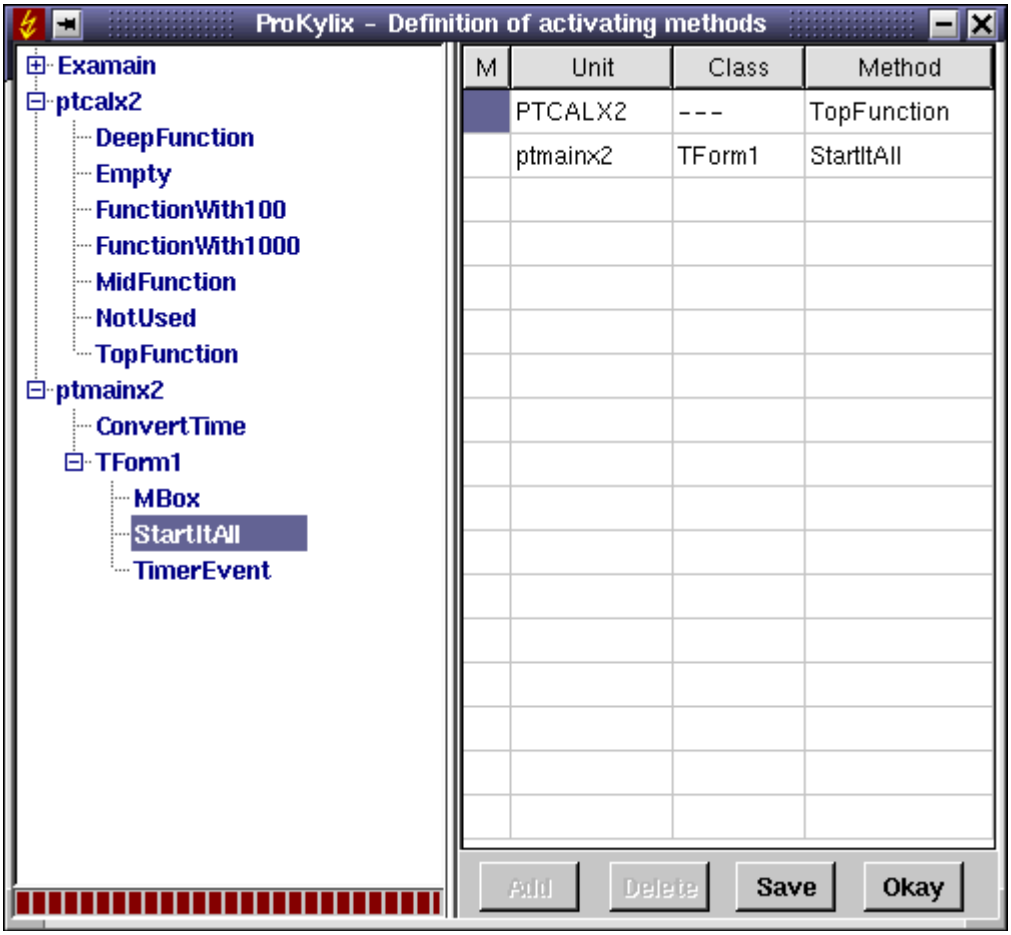
***By API-Calls or online operation window***

(see chapter A5.1 and A7 for details)

***By entering a selected method***

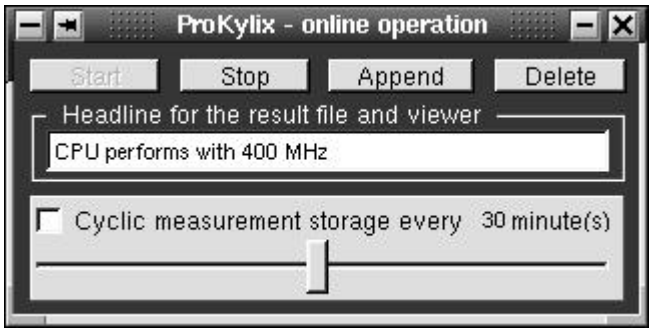
You'll be requested to enter methods (or you have already inserted //PROFILE-ACTIVATE statements into your source code (see also chapter A 4.2). If you use this option, you should not use the Online-operation window.

*See next page for an example.*




## A7 Online operation of the profiled program

With the online-operation window




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

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

The default value for the headline for intermediate results is:
'CPU performs with XXX MHz' where XXX depends on your PC.

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

## A8 Profiling Shared Object Librarys (DLL's)

Not yet supported in this release.

## A9 Treatment of special Linux- and Kylix-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 ‘Proftimx.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 µsleep function (defined in ‘Proftimx.pas):

```
PROCEDURE usleep(time : LongWord);
BEGIN
  StopCounting;
  Libc.usleep(time);
  ContinueCounting;
END;
```

Because of this redefinition, the Proftimx-unit must be named after the units Libc and QDialogs. This is normally done. The only exception is, if you name these units in the implementation part of the unit. Kylix itself places them into the interface part.

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

### A9.1 Redefined Linux-API functions

- Sleep, usleep and nanosleep (others will follow).

### A9.2 Redefined Kylix-API functions

- ShowMessage,
- ShowMessageFmt,
- ShowMessagePos,
- MessageDlg,
- MessageDlgPos.

### A9.3 Replaced Kylix-API functions

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

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

## A10 Conditional compilation

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

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

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

**These are the limitations:**

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

**This is evaluated correctly:**

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

**This example causes problems:**

```
CONST
  xxx = 4;
{$IF xxx > 5 }
  PROCEDURE AddIt(VAR first, second, sum : Int64);
  BEGIN
{$ELSE }
  PROCEDURE AddIt(VAR first, second, sum : Comp);
  BEGIN                    <- first Profiler statement is inserted after this BEGIN instead of after the previous
{$ENDIF }
  sum := first + second;  <- second Profiler statement inserted correctly here before END
END;
```

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

```
CONST
  xxx = 4;
{$IF xxx > 5 }
  PROCEDURE AddIt(VAR first, second, sum : Int64);
{$ELSE }
  PROCEDURE AddIt(VAR first, second, sum : Comp);
{$ENDIF }
  BEGIN                    <- first Profiler statement is inserted correctly after this BEGIN
  sum := first + second;  <- second Profiler statement inserted correctly here before END
END;
```

## A11 Limitations of use

Console applications have no online operation window. Procedures in a dpr-file can not be measured. The measured times differ about +-5 % (max) from those of an unprofiled program. The reason is that the program code is not so often replaced in the cache than without measuring.

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

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

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

In the freeware mode of ProKylix only 30 procedures can be measured, in the professional mode 32000.

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 run 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 of the return instruction 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. With AMD K6-2/K6-3, the differences were only a few CPU-cycles, on a Celeron they were great.

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.

## A12 Assembler Code

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

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

## A13 Modifying code vaccinated by ProKylix

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

## A14 Error messages

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

Vaccinating a file is done in this way:

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

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

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

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

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

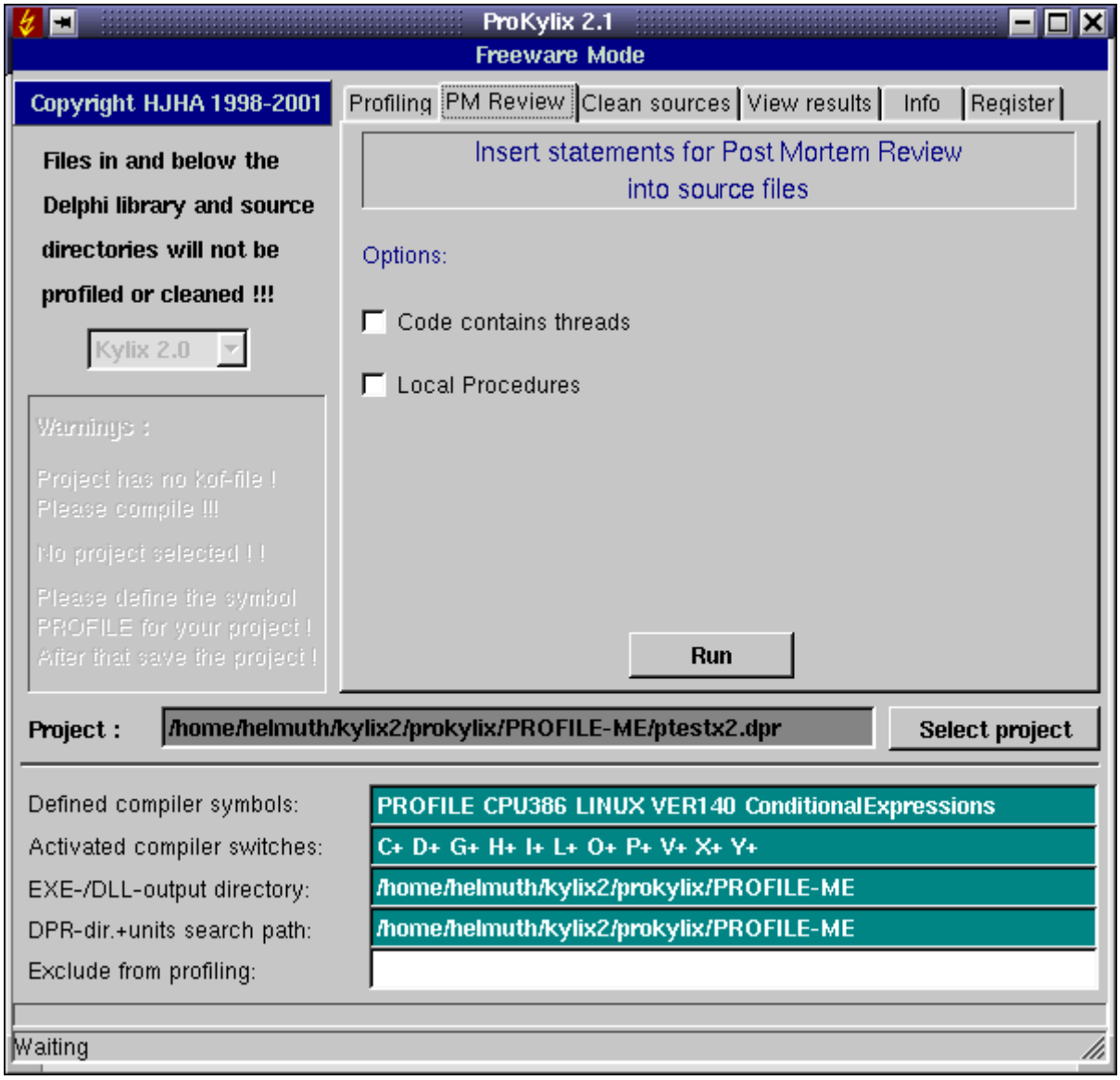
## A15 Security aspects

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

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

# B Post mortem review

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




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

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

For the availlable options see chapter A4.

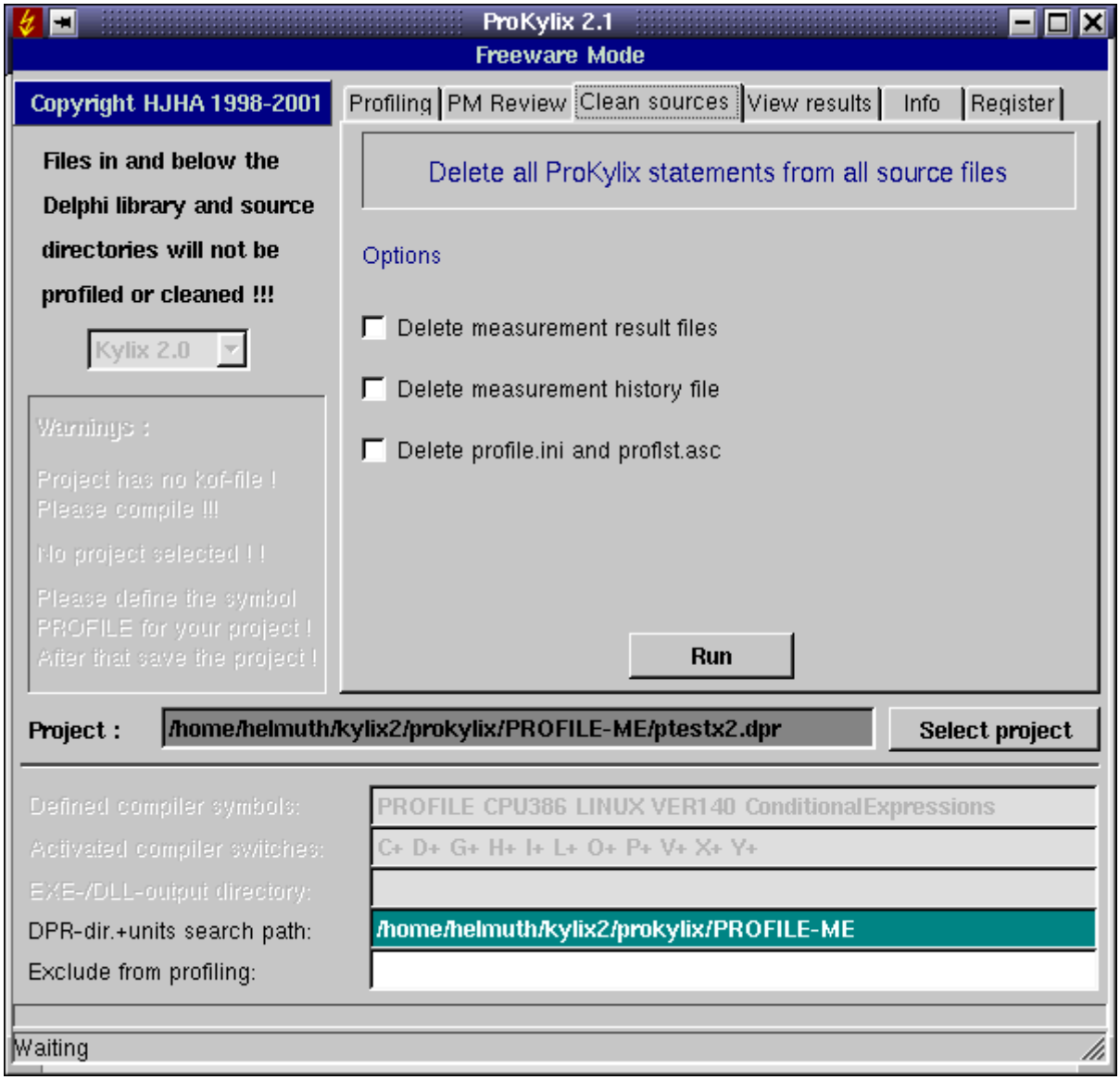
The handling of ProKylix is the same as for profiling. You also have to define the compiler symbol PROFILE:

If you have vaccinated ProKylix with statements for post mortem review and work with the IDE of Kylix 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 ProKylix itself needs stack space. And if there is no stack any more, ProKylix can not work properly. The overflow might as well appear in the ProKylix stack tracing routines. ProKylix can not handle this.

## C Cleaning the sources

If you want to delete all lines that ProKylix 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 ProKylix statements in the source code and inserts new statements. For that purpose it scans the code for statement that start with

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

and deletes them completely (except you have more than 1 space between IFDEF and PROFILE.

## D Compatibility

ProKylix was testet under
- Suse Linux 7.0,
- AMD K6-3 400 MHz.
- the Windows-version was also tested on a lot of Intel-processors and AMD K6, K6-2 and Athlon.

## E Installation of ProKylix

ProKylix is most comfortably installed with the included setup program (Setup). This program copies all necessary files into the Kylix-lib-directory. Also integrates ProKylix into the Kylix tools menu.

## F Description of the result file (for data base export and viewer)

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

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

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

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

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

## G Updating / Upgrading of ProKylix

Updates and upgrades can be loaded from the authors 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 registration key for unlocking the professional mode

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

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

The key to unlock the professional mode can be ordered by ShareIt shareware registration service (see the files order.txt). A link on http://www.prodelphi.de directly leads to the registration page at ShareIt.

## I Author

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

E-Mail: helmuth.adolph@prodelphi.de
Home pages: http://www.prodelphi.de

# J History

**Versions:**

| | |
|---|---|
| Version 1.0 : 02/2001 | First release for Kylix (for companion CD). |
| Version 1.2 : 03/2001 | First release for public use. |
| Version 1.3 : 04/2001 | Processing of compiler directives $IF, $ELSEIF and $IFEND added. |
| Version 1.4 : 04/2001 | Processing of Initializion an Finalization part corrected. |
| Version 1.5 : 04/2001 | Processing of boolean expressions for conditional compilation added. |
| Version 1.6 : 09/2001 | Bug in setup program fixed. |
| Version 2.0 : 10/2001 | Adapted to Kylix 2.0. |
| Version 2.1 : 12/2001 | Warning if compiler symbol PROFILE is not defined, parser bug fixed. |
| Version 2.2 : 01/2002 | Some small bugfixes |
| Version 2.3 : 03/2002 | Some small bugfixes |
| Version 2.4 : 05/2002 | Cyclic automatic storage of measured values, measuring specified parts of a procedure |

# 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).