# DYALOG

# Tuning Applications

## Using the
## ]PROFILE User Command

## Dyalog Limited

Minchens Court
Minchens Lane
Bramley
Hampshire
RG26 5BH
United Kingdom

tel: +44 (0)1256 830030
fax: +44 (0)1256 830031
email: support@dyalog.com
http://www.dyalog.com

*Version 13.1.0*

*Second Edition March 2012*

# Contents

# C H A P T E R   1

# Quick Start Guide

## Introduction

After an application has been in use for some time, usage patterns and data volumes often change in ways that mean that original assumptions about performance no longer hold. The `⎕PROFILE` system function and the associated `]PROFILE` user command are designed to make it easy to locate the "hot spots" in your application, where significant quantities of CPU or Elapsed time is being spent. Tuning these hot spots is usually the most effective way to improve application performance.

Note: `⎕PROFILE` replaces an older system function called `⎕MONITOR`, which is now considered obsolete. The old function still works, so that tools which use it will continue to function until they are no longer needed - but Dyalog recommends rewriting tools to use `⎕PROFILE` as soon as possible, and is likely to retire `⎕MONITOR` in a future release. The new mechanism provides high precision timing, calling tree analysis, and it also handles dfns and recursive code well, which the older mechanism did not. For more detailed information about `⎕PROFILE` itself, see the Version 13.0 Release Notes or the Dyalog Language Reference.

The `]PROFILE` user command provides reporting functionality which is intended to make it easy to summarize, filter and "drill down" on the (frequently large amounts of) data returned by `⎕PROFILE`.

## Quick Start Guide

**Before you start:** `⎕PROFILE` can generate very large quantities of data. To profile a large application, you may need to increase workspace size significantly: You might need a few hundred megabytes of additional workspace. Saving a result set as an XML file is a particularly hungry operation.

Let us assume that we have decided to see whether the sample function `Rain` in the `RainPro` workspace can be speeded up. If we are using Dyalog APL under Windows, the easiest way to get a quick overview of the performance profile of this function is to use the `]PROFILE` "dashboard" feature:

```
      )load rainpro[1]
C:\...\ws\rainpro saved Fri Mar 11 09:12:28 2011
… Release date: 28th Aug 2008 …

      ⎕PROFILE 'start' ◇ Rain 93 ◇ ⎕PROFILE 'stop'
View PG ⍀⍀⍀ to see graph for 1993 rainfall
      ]profile
```

---

[1] The RainPro examples in this manual were developed using the version of RainPro which shipped with Dyalog APL v13.0.  The version of RainPro that is shipped with Dyalog APL v13.1 has changed slightly and the `Rain` function is located at `Samples.Rain`

(if your data set is large, it may take some seconds to open the dashboard)

Under Windows, the user command enters dashboard mode by default:
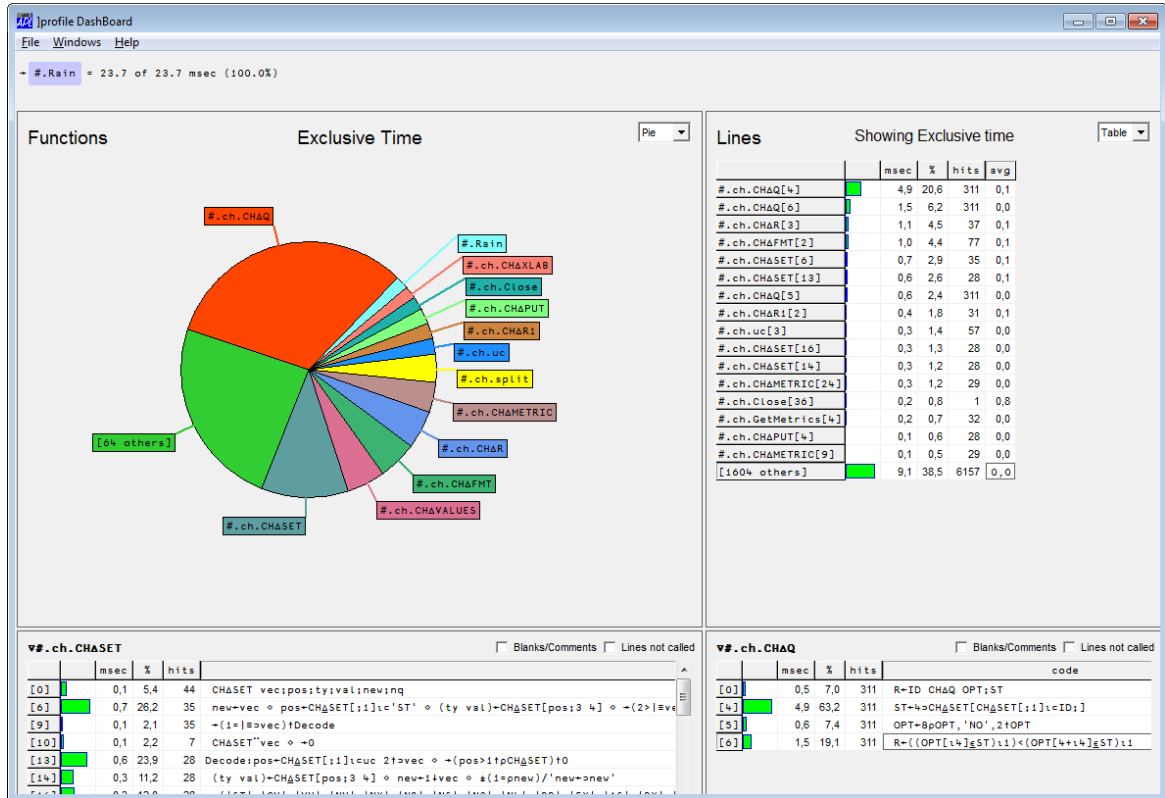


The default dashboard layout uses a pie chart in the top left quadrant of the screen showing usage broken down by function, and a table on the right showing the individual lines of code that were responsible for the highest consumption. On both sides, a drop-down allows you to switch between table and pie chart views of the same data.

A click on a pie segment or row of a table will cause the body of the function in question to be displayed in the corresponding window at the bottom of the screen. The screen shot above was taken after the user had clicked in the pie segment for the CHΔSET function on the left, and the row for the CHΔQ function in the table on the right.

From the above, we can see that the function CHΔQ is responsible for about a third of the CPU consumption, with a single line (line 4) responsible for 20% of CPU consumption on its own. We can see that CHΔQ was called a large number of times, which means that there may be two opportunities for reducing the consumption: We can tune the function itself, and investigate whether it is being "misused" by the functions that call it.

We can look at how the function is being used by right-clicking on the relevant pie segment and selecting "Show Calls" from the menu. In the following screen shot, the user has also double-clicked in the top function quadrant, which gives the two function windows all the available space, and then clicked in the CHΔAXES pie segment to inspect that function.

In "calls" mode, the information that is displayed shows how many calls were made, and how much of the consumption that occurred in the calls that originated in a given function or line.
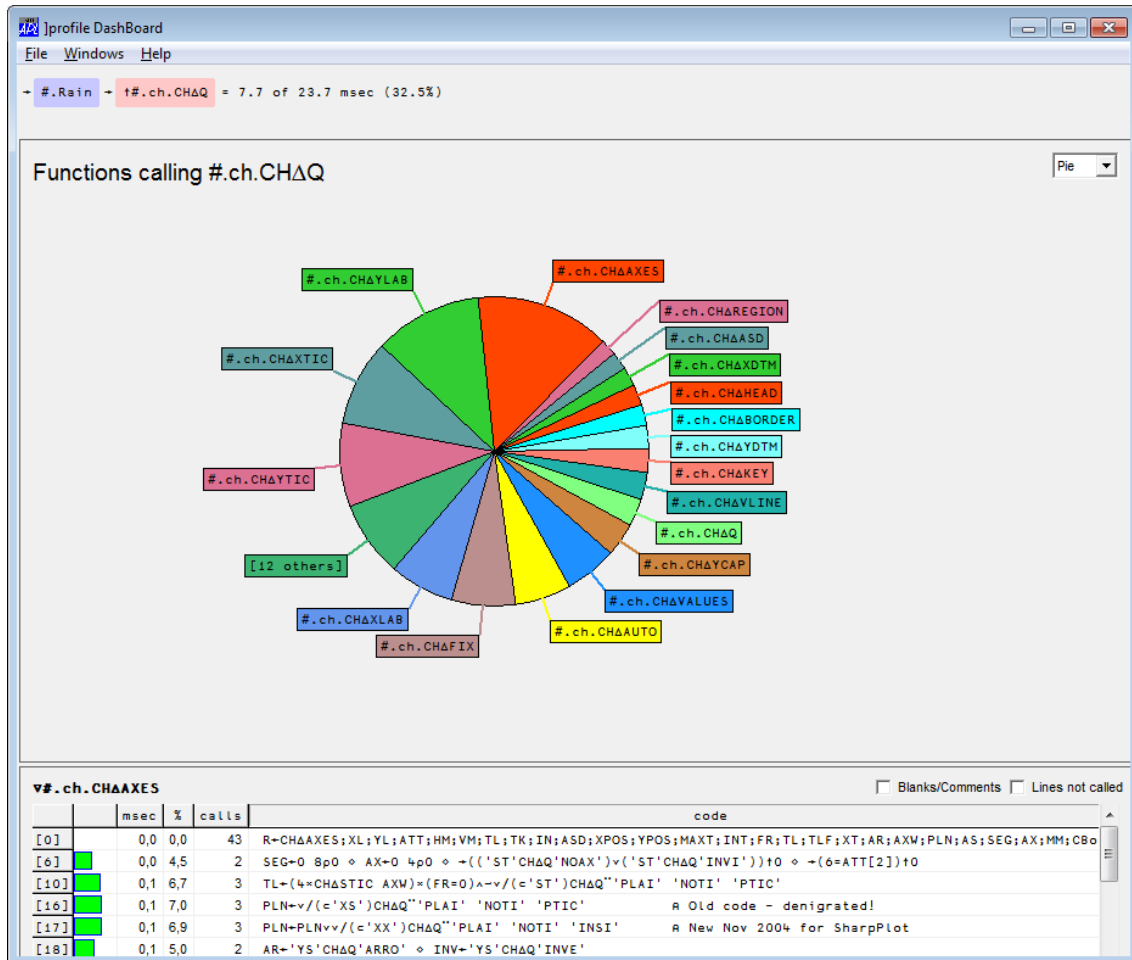


The above shows little evidence that the function is being over-used, so we should probably look at tuning the function itself. We can see from the first screen shot that the line:

```
CHΔQ[4] ST←4⊃CHΔSET[CHΔSET[;1]ι⊂ID;]
```

… is responsible for 20% of the CPU consumption on its own. By putting a stop on the function, we can discover that the `CHΔSET` matrix is a 233×5 nested array, and using the `cmpx` function from the `dfns` workspace we can see that repeatedly indexing the first column out of the matrix in order to search it is costing about half the CPU time (in other words, 10% of overall CPU consumption):

```
      cmpx 'CHΔSET[CHΔSET[;1]ι⊂ID;4]' 'CHΔSET[;1]'
  CHΔSET[CHΔSET[;1]ι⊂ID;4] → 1.2E¯5 |   0% ⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕
* CHΔSET[;1]               → 5.5E¯6 | -53% ⎕⎕⎕⎕⎕⎕⎕⎕⎕
```

Since the application searches this column 311 times, we can see that we might be able to speed the application up by 10% by devising a new strategy for the representation of option settings, where the IDs are cached separately for lookups.

## Textual Reports

The dashboard is the easiest way to get started if you are on a system which supports the Windows GUI, but the same information is available as textual output from the `]profile` user command. We could have performed the same analysis that was described above using a few commands. The information in the pie chart (consumption by function) can be found using:

```
        ]profile summary -expr="Rain 93" -first=10
Total time: 35.6 msec

  Element               msec        %  Calls
  #.Rain                35.6    100.0      1
  #.ch.Step             24.5     68.9      1
  #.ch.CHΔQ             11.8     33.1    311
  #.ch.CHΔSET           11.1     31.1     44
  #.ch.Set               5.8     16.4      9
  #.ch.CHΔSTEP           4.7     13.1      1
  #.ch.CHΔVALUES         4.1     11.5      2
  #.ch.Vline             3.4      9.5      1
  #.ch.CHΔXLAB           3.1      8.8     13
  #.ch.CHΔAXES           2.7      7.5      1
```

The `-expr` switch allows the specification of an APL statement to run, and is equivalent to executing:

```
⎕PROFILE 'clear' ◇ ⎕PROFILE 'start' ◇ Rain 93 ◇ ⎕PROFILE 'stop'
```

`-first=10` limits the output to the top 10 functions. To see the top 5 *lines* of code, you can follow the above with:

```
        ]profile summary -lines -first=5
 Total time: 35.6 msec

  Element               msec        %  Calls
  #.Rain[25]            24.6     68.9      1
  #.ch.CHΔQ[4]           7.6     21.3    311
  #.ch.Set[4]            5.2     14.6      9
  #.ch.Step[53]          4.7     13.2      1
  #.ch.Step[43]          4.4     12.4      1
```

And finally, we can get the call analysis report for the `CHΔQ` function:

```
        ]profile calls -fn=#.ch.CHΔQ -first=5
 Total time: 35.6 msec; Selected time: 11.1 msec

  Element               msec     %  Calls
  #.ch.CHΔAXES           1.7    4.8     43
  #.ch.CHΔYLAB           1.3    3.8     34
  #.ch.CHΔXTIC           1.1    3.0     28
  #.ch.CHΔYTIC           1.0    2.9     26
  #.ch.CHΔFIX            0.8    2.2     21
```

# C H A P T E R   2

# Collecting Data

## Basics

The ⎕PROFILE system function registers usage data for all APL functions that are executed between calls to (⎕PROFILE 'start') and (⎕PROFILE 'stop'). Data collection can be turned on and off several times, and the results will be accumulated. To start a completely new recording session, use (⎕PROFILE 'clear'). By default, ⎕PROFILE will register CPU usage data using the best available counter. If you are more interested in elapsed time, use (⎕PROFILE 'start' 'elapsed'). Note that you must do a clear before you can switch from recording CPU to elapsed time or vice versa.

**Note:** Unlike ⎕MONITOR, which needed to be turned on and off for each function to be monitored and which stored data in the function body within the active workspace, ⎕PROFILE is turned on or off for all functions in a single operation and stores usage data outside the workspace. Thus, it can be used for functions which are dynamically paged in and out of the workspace, although the results will be confusing if several different functions with the same name are used at different times during execution.

At any time, you can determine whether ⎕PROFILE is collecting data using (⎕PROFILE 'state'):

```
      ⎕PROFILE 'start' 'Elapsed'
      ⎕PROFILE 'state'
 active  elapsed  0.001215608791  0
```

The first element of the result is either **active** (started) or **inactive** (stopped). The second reports the name of the counter being used for timing, which will either be **CPU** or **elapsed**. The third and fourth element report the best estimate for the cost of calling the timer that is in use, and - if it has been possible to determine it - the granularity of the timer (smallest measurable interval measurable). On most platforms, the fourth element will be zero, indicating that the granularity is smaller than the cost, and can therefore not be estimated.

The timer cost and granularity are estimated the first time ⎕PROFILE is used in an APL session. If you suspect that the system was particularly busy at that time, you can request a new calibration using (⎕PROFILE 'calibrate').

Once data has been collected, summarised data per function and function line can be retrieved using (⎕PROFILE 'data'). Data can also be broken down by calling trees, so that data is summarized separately for every different path that led to the function being executed. (⎕PROFILE 'tree') returns this data. In both cases, a very large quantity of data may be returned, and ]PROFILE is provided in order to produce a number of reports based on this data - and to save recorded information and analyze it later.

For each function, and for each individual line in a function, the following data is recorded:

- **Calls**: The number of times the line or function was executed

- **Exclusive Time**: Milliseconds spent, *excluding* time spent in functions that were called.

- **Inclusive Time**: Milliseconds spent, *including* time spent in functions that were called.

- **Exclusive Timer Count**: Number of times the timer was inspected in order to produce the Exclusive Time total.

- **Inclusive Timer Count**: Number of times the timer was inspected in order to produce the Inclusive Time total.

## Timer Overhead

Unfortunately, most system timers have a significant cost associated with them (it takes a significant amount of time to ask what time it is). In applications with relatively inexpensive lines of APL code, the overhead can be significant. It is rarely enough to defeat the search for hot spots, but can skew certain results.

The estimate for the timer cost and the timer counts are provided in order to allow results to be corrected for the effects of the overhead of constantly inspecting the system timer (by multiplying timer counts with the estimated cost of asking what time it is). By default, the reports produced by `]PROFILE` automatically adjust for "timer bias", using the recorded bias. You can disable this adjustment for a report using a switch setting of `-bias=0`.

**NOTE:** Experiments show that the cost of querying a timer can be extremely variable if the system is loaded. **Repeatable timings are only possible if there is very little activity on the system other than the APL system which is being profiled.** It is always a good idea to increase the priority of a system which is being profiled, but this will not remove the variability of the timer cost, as this involves calls to the operating system kernel, which is servicing all processes on the machine.

## Storing Data

All of the reports will accept a switch `-outfile=`, which names a file to which the report data should be written. By default, the data is stored in an XML format. The switch `-format=` accepts three options: `xml`, `csv` and also `txt`, which just means that the formatted report output is written to a file. Although all of the reports can be written to file, the `tree` report in `xml` format is the only way to store a complete data set that can be reused for reporting at a later time using the `-infile=` switch. In other words:

```
]profile tree -outfile=c:\temp\one.xml -title="Testing"
```

… can later (and not necessarily in the same APL session) be followed by …

```
]profile summary -infile=c:\temp\one.xml
```

Note that user commands can be executed under program control, which means that your application can record its own usage data. For example:

```
⎕SE.UCMD 'profile tree -outfile=c:\abctree.xml -title="ABC"'
```

# C H A P T E R  3

# Reporting

## Overview

The `]profile` command is always followed by a command keyword, which is typically the name of a report. The `summary` and `calls` commands are the most frequently used reporting tools. `dashboard` provides a graphical user interface, and `state` simply displays the current ⎕profile state in a friendly fashion. The `tree` command is typically used to extract a complete set of profile data and write it to file. Finally, the `data` command can be used to write raw data to file, most often for analysis using Excel or other tools.

The reporting commands take a number of switches which filter the data which is displayed, or add optional output columns, and switches which can be used to read input from a previously saved file, or store the results of a command in a file.

An overview of commands and the switches that they accept are provided below:

## Command Summary

| Command | Description |
|---|---|
| summary | Summary report showing the number of calls, total consumption, and consumption as a percentage of overall consumption. This is the default on systems with no GUI. |
| calls | Call analysis report, showing how the consumption of a named function (the `-fn=` switch is required) is broken down by calling function. |
| dashboard | Starts the graphical dashboard under Microsoft Windows only, and is the default if no command is provided. |
| state | Formats the result of (⎕PROFILE 'state'). |
| tree | Returns the raw data produced by (⎕PROFILE 'tree'). Intended as a tool for storing data using the `-outfile=` option, for subsequent reporting using `-infile=`. |
| data | Returns the raw data produced by (⎕PROFILE 'data'). This command is essentially provided as a mechanism for writing this data to file for use with other tools than `]profile`. |

## Data Selection Switches

The `summary` and `calls` commands support the following switches, which select data to be included in the report.

| Switch | Description |
|---|---|
| `-lines` | Display consumption by individual line. The default is to produce totals per function. The `-code` switch also forces `-lines` mode. |
| `-exclusive` | Show the consumption of each line or function *excluding* consumption in sub-functions called. |
| `-first=n` | Only displays the n first functions or lines after sorting in descending order by consumption |
| `-avg` | Include a column showing the average consumption per execution of the line or function. |
| `-cumpct` | For each row of output, show the percentage of overall consumption that this row *and all rows above it* were responsible for. This column of output is *usually* only useful if `-exclusive` is also selected. |
| `-pct=n` | As an alternative to `-first=`, show only those entries where the cumulative percentage is less than or equal to n. |
| `-fn=name` | For `calls`, this switch is required and selects the function for which a call report will be produced. For `summary` and `dashboard`, it filters the output so that it only includes data for the selected function and other functions that it calls. A list of function names separated by commas specifies a drill down path (see dashboard chapter for details). |
| `-code` | When output is *not* directed to file, this switch can be used to add a column containing the source code for the line in question. Note that `-code` forces `-lines` mode. |

## Output (or Input) Redirection

The following switches can be used to direct output to a file rather than display it in the session - or load

| Switch | Description |
|---|---|
| `-outfile=` | Causes output to be directed to a file rather than being displayed in the APL Session. The parameter value must be a valid file name in an existing folder. |
| `-format=`<br>`xml|csv|txt` | Selects a file format for `-outfile` (the default is `xml`). `txt` indicates that the formatted report should be saved in a text file exactly as it would have been displayed. The other formats write unformatted data to the file. |
| `-separators=` | For use with `-format=csv`, allows you to set the decimal and comma separators. In Europe, you will typically want to use `-separators=",;"`. |
| `-infile=` | Files in `xml` format, resulting from the use of the `tree` command, can be used as input to the summary and calls commands. When used, the report or views are generated based on the data found in the file, rather than the current data reported by `⎕PROFILE`. |

## Other Switches

| Switch | Description |
|---|---|
| `-decimal=`*n* | For columns which are not whole numbers, decides the number of decimals to display in formatted output. |
| `-bias=`*n* | Overrides the function call overhead estimated by `⎕PROFILE` during the current session (or read from an `infile`), and uses n instead. Use *n*`=0` to ignore bias, or some other fixed value if you want to make sure that you use the same bias for data collected at different times. |

## Examples

The following examples are intended to show at least one use of every command and switch:

```
      )load rainpro
C:\...\ws\rainpro saved Fri Mar 11 09:12:28 2011
```

Execute the expression "Rain 93" and bring up an summary showing the five functions which consumed the most CPU:

```
      ]profile summary -expr="Rain 93" -first=5
Total time: 68.4 msec

  Element        msec        %  Calls
  #.Rain         68.4    100.0      1
```

```
#.ch.Step        47.1    68.9        1
#.ch.CHΔQ        23.0    33.7      311
#.ch.CHΔSET      20.8    30.4       44
#.ch.Set         11.0    16.0        9
```

Show the five biggest CPU consumers, *excluding* CPU spend in sub-functions. Use 3 decimals to display fractions, include a cumulative % column and only include functions up to 65% of the cumulative CPU:

```
      ]profile summary -exclusive -dec=3 -cumpct -pct=65
 Total time: 68.4 msec

 Element             msec        %  Calls    %(cum)
 #.ch.CHΔQ         23.032   33.672    311    33.672
 #.ch.CHΔSET        7.866   11.500     44    45.172
 #.ch.CHΔFMT        3.422    5.003     77    50.176
 #.ch.CHΔR          3.402    4.973     37    55.149
 #.ch.CHΔVALUES     3.240    4.737      2    59.885
 #.ch.split         2.381    3.481     33    63.367
```

Include the average CPU consumption per call, and do not adjust for timer bias (note the numbers are somewhat higher, the total time is 3.3 msec higher and the function CHΔQ is reported as having consumed 0.788 msec more:

```
      ]profile summary -excl -dec=3 -avg -bias=0 -first=3
  Total time: 71.7 msec

  Element            msec        %  Calls     avg
  #.ch.CHΔQ        23.820   33.232    311   0.077
  #.ch.CHΔSET       8.112   11.317     44   0.184
  #.ch.CHΔFMT       3.497    4.879     77   0.045
```

The data command allows us to take a look at the raw data recorded for a function (without bias adjustment, but with counters for the number of clock inspections):

```
      ]profile data -fn=#.ch.CHΔQ
 Total time: 276.9 msec; Selected time: 23.8 msec

 Element       Calls msec(inc) msec(exc)  ticks(inc)  ticks(exc)
 #.ch.CHΔQ       311      23.8      23.8        4043        4043
 #.ch.CHΔQ[1]    311       0.3       0.3         311         311
 #.ch.CHΔQ[2]    311       0.3       0.3         311         311
 #.ch.CHΔQ[3]    311       0.3       0.3         311         311
 #.ch.CHΔQ[4]    311      15.3      15.3         311         311
 #.ch.CHΔQ[5]    311       1.7       1.7         311         311
 #.ch.CHΔQ[6]    311       4.3       4.3         311         311
```

We can see that the timer was inspected 4043 times while profiling the function. There fact that there is no difference between inclusive and exclusive time tells us that no sub-functions were called. If we multiply the count by the estimated cost, found using the state command:

```
      ]profile state
state       : inactive
timer       : CPU
timer cost  : 0.0001949486972
granularity : 0
```

… we can verify that the difference caused by the timer "bias" is 4043×0.0001949486972, or 0.788 msec, which is fortunately the same as the difference that we had observed when we applied -bias=0 earlier.

For a summary or calls report, the -code switch can be used to add source code to the report:

```
      ]profile summary -code -lines -first=5
Total time: 68.4 msec

 Element          msec     %  Calls  Code
 #.Rain[25]       47.2  69.0      1  ch.Step(¯1↓1,⌽v\⌽TOT[;2]>0)≠TOT
 #.ch.CH∆Q[4]     15.3  22.3    311  ST←4⊃CH∆SET[CH∆SET[;1]ι⊂ID;]
 #.ch.Set[4]       9.7  14.2      9  CH∆SET vec
 #.ch.Step[53]     8.9  13.1      1  cht←NZ CH∆STEP DATA
 #.ch.Step[43]     8.6  12.6      1  PG←PG,CH∆HEAD ◊ ax←CH∆AXES
```

The **-outfile** switch makes it possible to direct output to a file instead of displaying in the session. By default, the data format is **xml**, but the **-format=** switch can be used to select **csv** or **txt** as alternatives. The appendices contain examples of the **xml** and **csv** formats; using the **txt** format will simply cause the report to be written to file in exactly the same format as it would otherwise have been displayed.

The **xml** format generates very large files, but they have the advantage that they can be used as input to the **]profile** command, as they can contain a complete representation of the current state.

The **csv** format can be used to export data in a format that many external tools will be able to load. For example:

```
]profile data -outfile=c:\temp\data.csv -format=csv
               -separators=",;"
```

The above creates a **csv** file using comma as the decimal separator and semicolon as the field separators - for use with Danish Excel. We can now proceed to:

```
      'XL' ⎕WC 'OLEClient' 'Excel.Application'
      XL.Visible←1
      XL.Workbooks.Open⊂'c:\temp\data.csv'
```

… with the following result:
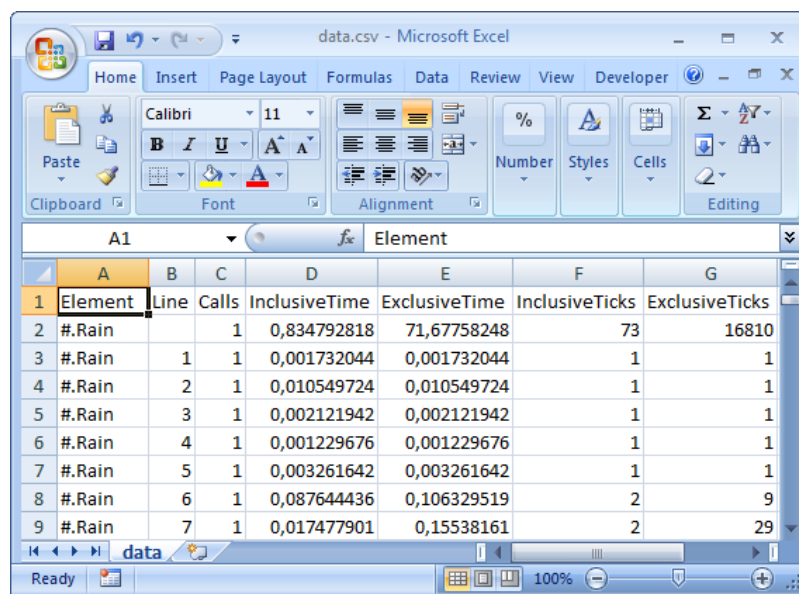
# C H A P T E R  4

# The DashBoard

## Introduction

Under Windows (or to be more precise, on systems where the Win32 API is available to the Dyalog GUI), `]PROFILE` provides a graphical "DashBoard", which quickly provides an overview of the resource consumption of an application, and allows you to drill down in pursuit of interesting tuning opportunities.

Where available, the dashboard is the default mode of operation. If you type `]profile` with no arguments, the dashboard will give an overview of the data which is currently stored by `⎕PROFILE` (`⎕PROFILE` must be stopped, or "inactive"). The `-expr=` and `-infile=` switches make it possible to start the dashboard to look at other data than that which is currently active, for example:

```
]profile -expr="Rain 93"
```

(runs the expression "`Rain 93`" and then starts the dashboard to analyse the results - this will destroy any existing `⎕PROFILE` data)

```
]profile -infile="c:\temp\rain93-2011-03-22.xml"
```

(loads a previously saved dataset for analysis - and will ***not*** interfere with the current state of `⎕PROFILE`).

As can be seen on the next page, the dashboard is divided up into four quadrants by splitters. By default, the left hand side is used to display consumption broken down by function, while the right hand side breaks consumption down by line. A pie chart is used for the function break down, and a table for the lines - but a drop-down in the top right corner of each side allows you to change the display form.

## Drill Down

A single click in a pie segment or table row will cause the source code for the selected function to be displayed in the quadrant below. A double-click will cause a "drill down" to occur. A trail of "bread crumbs" is displayed at the top left of the form, to illustrate the current filtering. A right arrow preceding a blue "crumb" indicates a direct call to the function without intermediate functions. A star before a light green crumb identifies a call sequence where other functions may have been called in-between. A pink bread crumb is used to indicate a "show calls" step (which will always be the final crumb, as no further drill down is possible from this mode).

The bread crumbs below show us that the root function is `Rain`, and we have drilled down on the function `CH∆SET`[2].
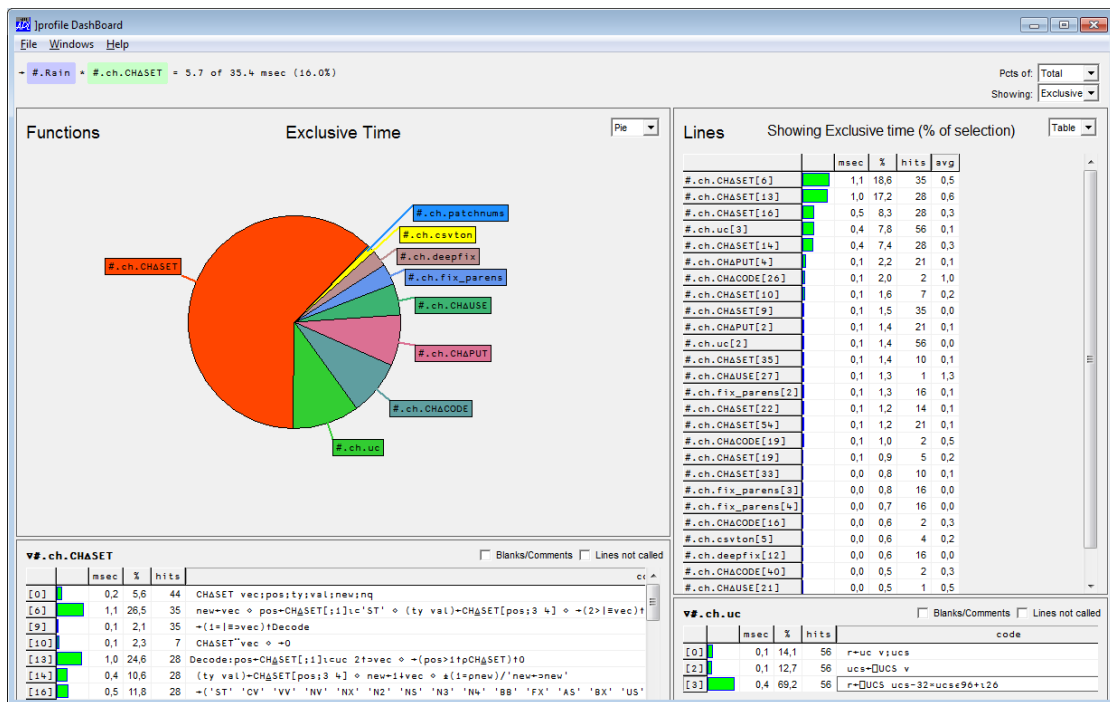


## Display Options

Immediately following the bread crumbs is a label which reports how large a percentage of the overall consumption is currently included. The drop-down at the top right allows you to choose whether percentages that are reported in tables or "tips" on pie labels are computed as percentages of the overall consumption, or of the selection.

The drop-down immediately below that allows you to select whether table view will report inclusive or exclusive time. Pie charts always report exclusive time.

In the function detail views at the bottom of the screen, two check boxes allow you to select whether you want to include lines which are either blank or consist of a single comment, and whether lines which were not called at all should be displayed.

---

[2] The current user interface does not provide a way to select a direct call, "drill down" always allows indirect calls. The first crumb will be blue if `]profile` has identified a single top-level function.

## Right Click Menus

A right click on a pie segment or label, or in a table row, provides the following options:

**Drill Down:** Drills down on the function in question - same as a double click.

**Make Root:** Shows only consumption which originates in the selected function (equivalent to starting with the `-fn=` switch selecting the function).

**Show Calls:** Switches to a mode where consumption is broken down according to the functions and lines which have called the selected function. If drill down has already occurred, the higher levels of filtering are also retained.

**Reset:** Returns to the starting position.

**Up 1 Level:** Equivalent to clicking on the next-to-last bread crumb.

## File and Help Menus

The following functionality can be selected from the File menu:

**File|Open:** Allows analysis of a saved file - equivalent to starting the dashboard with the `-infile=` switch.

**File|Save:** Saves the current dataset - equivalent to using the `-outfile=` switch.

**File|Reset:** Returns to the initial state.

**File|Exit:** Leaves the dashboard.

**Help|About:** Reports the version of the `]PROFILE` user command.

## Windows Menu

The Windows menu provides a number of alternatives to resizing windows using the splitters:

**Windows|Reset:** Moves the vertical splitter to the middle and the horizontal splitters to a position about ¾ of the way down.

**Windows|Functions:** Moves the vertical splitter all the way to the right, showing only the functions breakdown (equivalent to double-clicking at the top of the function pane).

**Windows|Function Details:** Vertical splitter to the right, horizontal splitter to the top, showing only function details (equivalent to double-clicking at the top of the function detail pane).

**Windows|Lines:** Moves vertical splitter all the way left, showing only the line breakdown (equivalent to double-clicking at the top of the lines pane).

**Windows|Line Details:** Vertical splitter to the left, horizontal splitter to the top, showing only function details  (equivalent to double-clicking at the top of the lines pane).

As mentioned above, focus can be given to a window or pair of windows by double-clicking at the top of one of the panes. A subsequent double-click will return the splitters to the position that they had previously.

## Single Function Mode

If the dashboard is opened on a data set which only contains data for a single function, for example as a result of:

```
)load dfns
]profile -expr="ρqueens 8"
```

… then the dashboard will open in mode where the left hand side is used only to display the detailed view of the function body (horizontal splitter at the top), and the right hand side shows only the breakdown by line (splitter at the bottom):

# Appendix A: XML File Format

The XML format produced by the `]profile` user command consists of an outer <ProfileData> element, which contains a <ProfileSettings> element followed by a number of <ProfileEntry> items, one for each row of output data.

The <ProfileSettings> element contains version number of the `]PROFILE` user command that produced the file, the report title, information about timer cost and other information  regarding the report, including the total registered time for the report.

Each <ProfileEntry> contains one element for each output column, depending on the command and switches, selected from the following set:

| Element | Description |
|---|---|
| Depth | Tree depth |
| Element | Function Name |
| Line | Line number, empty for a function summary entry |
| Calls | Number of times the function or line was called |
| InclusiveTime ExclusiveTime | Inclusive and Exclusive time |
| InclusiveTicks ExclusiveTicks | Number of times the clock was inspected to record inclusive and exclusive time, respectively |
| PctOfTot | Consumption as percentage of total |
| CumPct | Cumulative percentage |
| AvgTime | Average Time per Call |

These element names also appear as column headers when the `csv` file format is used. The following page contains examples of `xml` output files produced by `]profile`.

## XML Example Files

```
]profile tree -outfile=c:\temp\tree.xml

  <ProfileData>
    <ProfileSettings>
      <Version>0.9.0</Version>
      <Title>2011/03/23 22:49:19</Title>
      <TimerBias>0.00019494869718528207</TimerBias>
      <Command>tree</Command>
      <TotalTime>272.9692285</TotalTime>
      <SelectedTime>272.9692285</SelectedTime>
    </ProfileSettings>
    <ProfileEntry>
      <Depth>0</Depth>
      <Function>#.Rain</Function>
      <Line></Line>
      <Calls>1</Calls>
      <InclusiveTime>0.8205615629464091</InclusiveTime>
      <ExclusiveTime>68.40049487792044</ExclusiveTime>
      <InclusiveTicks>73</InclusiveTicks>
      <ExclusiveTicks>16810</ExclusiveTicks>
    </ProfileEntry>
(… many more occurrences of <ProfileEntry> …)
</ProfileData>
```

```
]profile summary -outfile=c:\temp\tree.xml

  <ProfileData>
    <ProfileSettings>
      <Version>0.9.0</Version>
      <Title>2011/03/23 22:48:16</Title>
      <TimerBias>0.00019494869718528207</TimerBias>
      <Command>summary</Command>
      <TotalTime>68.40049488</TotalTime>
      <SelectedTime>68.40049488</SelectedTime>
    </ProfileSettings>
    <ProfileEntry>
      <Function>#.Rain</Function>
      <Line></Line>
      <InclusiveTime>68.40049487792044</InclusiveTime>
      <PctOfTot>100</PctOfTot>
      <Calls>1</Calls>
    </ProfileEntry>
    (… many more occurrences of <ProfileEntry> …)
  </ProfileData>
```

# Appendix B: CSV File Format

This section contains a few examples of output files created using `-format=csv` (all files are encoded as UTF-8). The first row of each file contains column names, selected from the same list as the element names that can appear in XML files, listed in Appendix A.

`]profile tree -outfile=c:\temp\data.csv -format=csv`

```
"Depth","Element","Line","Calls","InclusiveTime","ExclusiveTime
","InclusiveTicks","ExclusiveTicks"
0,"#.Rain",,1,0.8205615629,68.40049488,73,16810
1,"#.Rain",1,1,0.001537095514,0.001537095514,1,1
1,"#.Rain",2,1,0.01035477506,0.01035477506,1,1
1,"#.Rain",3,1,0.001926992908,0.001926992908,1,1
1,"#.Rain",4,1,0.001034727703,0.001034727703,1,1
1,"#.Rain",5,1,0.003066692969,0.003066692969,1,1
1,"#.Rain",6,1,0.0872545383,0.1045749803,2,9
2,"#.Gilling.LEAP",,1,0.01732044204,0.01732044204,7,7
3,"#.Gilling.LEAP",1,1,0.0008172849135,0.0008172849135,1,1
3,"#.Gilling.LEAP",2,1,0.001597079731,0.001597079731,1,1
3,"#.Gilling.LEAP",3,1,0.01222178376,0.01222178376,1,1
1,"#.Rain",7,1,0.01708800315,0.1497280978,2,29
2,"#.Gilling.TOTΔMTHS",,1,0.07632241511,0.1326400946,8,27
3,"#.Gilling.TOTΔMTHS",1,1,0.0008022888807,0.0008022888807,1,1
3,"#.Gilling.TOTΔMTHS",2,1,0.06257103391,0.1188887134,2,21
(…etc…)
```

`]profile data -outfile=c:\temp\data.csv -format=csv -separators=",;"`

```
"Element";"Line";"Calls";"InclusiveTime";"ExclusiveTime";"Inclu
siveTicks";"ExclusiveTicks"
"#.Rain";;1;0,8347928178;71,67758248;73;16810
"#.Rain";1;1;0,001732044211;0,001732044211;1;1
"#.Rain";2;1;0,01054972375;0,01054972375;1;1
"#.Rain";3;1;0,002121941606;0,002121941606;1;1
"#.Rain";4;1;0,0012296764;0,0012296764;1;1
"#.Rain";5;1;0,003261641666;0,003261641666;1;1
"#.Rain";6;1;0,08764443569;0,1063295186;2;9
"#.Rain";7;1;0,01747790054;0,15538161;2;29
"#.Rain";8;1;0,01985477505;2,402712708;2;445
"#.Rain";9;1;0,02023717445;1,841845304;2;466
(…etc…)
```

`]profile summary -first=5 -outfile=c:\temp\data.csv -format=csv`

```
"Element","Line","Time","PctOfTot","Calls"
"#.Rain",,68.40049488,100,1
"#.ch.Step",,47.14893489,68.93069264,1
"#.ch.CHΔQ",,23.0318165,33.67200272,311
"#.ch.CHΔSET",,20.80886149,30.42209201,44
"#.ch.Set",,10.97602407,16.04670272,9
(…etc…)
```