Profile4D 1.0

2003-10-03

What is it?

The Profile4D component is a group of methods that make it very easy to simultaneously profile multiple blocks of code very quickly and without making a mess of your existing code.

How it works.

You determine an area of code that you feel is running too slowly. Before you begin to optimize the code you need to know exactly which part is running slowest. Even the best developers are often surprised at which code is actually slowest. We will call this area of code that we want to test the Main Area.

You place a call to `Profile4D_Init` and the beginning of the Main area and a call to `Profile4D_Deinit` at the end of this area. `Profile4D_Init` sets up an array that will be used to keep track of information for each block of code being profiled within the Main Area. It also starts the Main Area timer. `Profile4D_Deinit` will be explained in a moment.

Now, identify blocks of code within the Main area (which might span multiple methods) that you think might be the cause of the slowness of the Main Area. Place a call to `Profile4D_Start()` at the beginning of each of these blocks and a call to `Profile4D_Finish()` at the end of each of these blocks. Pass a description of the block of code to these two methods and make sure you pass exactly the same description to each pair.

`Profile4D_Start` simply starts (or restarts) a timer for that block of code. `Profile4D_Finish` stops the timer and increments the time taken for this execution. `Profile4D_Finish` also increments the number of executions for this block of code.

You can have as many `Profile4D_Start/Profile4D_Finish` pairs as you want between each `Profile4D_Init/Profile4D_Deinit` pair.

Once the code gets to `Profile4D_Deinit`, the main timer is stopped and the developer is asked where they want the profiling information saved to. The information is then saved to disk where it can be interpreted later.

You can also pause the profiling by calling `Profile4D_Pause` and then continue profiling by calling `Profile4D_Resume`. Using these commands you can effectively stop any profiling timers for however long you like. An example where this is useful would be in a routine that has to stop to ask the user for information. You could pause just before the dialog and then resume after. Profile4D will report statistics as if a pause never happened.

Now you can decide what parts of your code need optimization. Change your code how you like and then run it again and compare the results. Repeat as needed.

When you are finished profiling, do a global search for "`Profile4D_`" and then delete those lines of code. So it does mess your code up a little, but not too bad, right? :-)

What it won't do.

Profile4D is not made to nest `Profile4D_Init/Profile4D_Deinit` pairs within the same process. Don't do this. However, there is no problem with nesting `Profile4D_Start/Profile4D_Finish` pairs. If you do, just remember to interpret the results accordingly. For example, the subtotals will become meaningless.

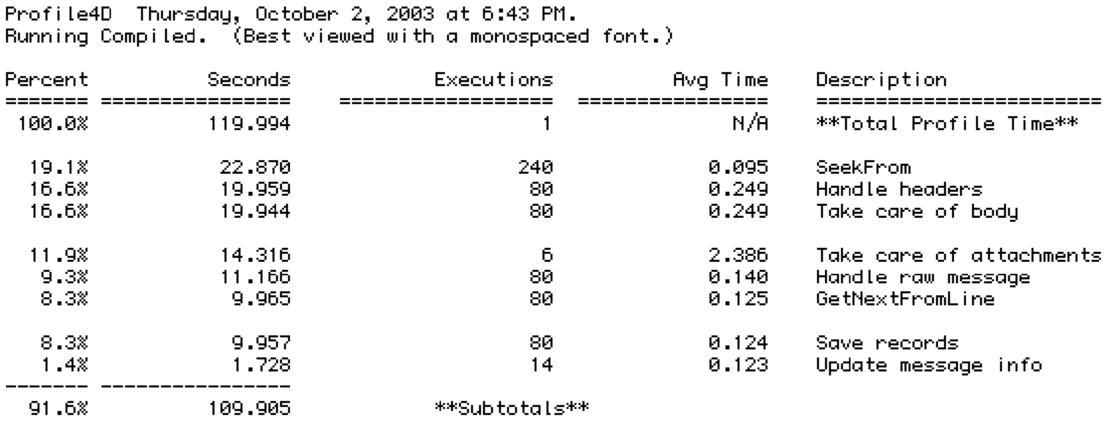
Error checking

There is very little error checking in any of the methods. This is partly because there is very little that can go wrong and mostly because all the `Profile4D_` methods need to execute very quickly so that they affect the profiling data as little as possible. So it is up to the developer to make sure that all the `Profile4D_` pairs line up properly and are in the right order. For example, if you have a `Profile4D_Start()` line in method `aaa`, and this method happens to be called from another process where `Profile4D_Init` hasn't been called, 4D might complain!

What does it report?

The file that Profile4D creates is formatted to be viewed directly in a text editor like BBEdit using a monospaced font. It can also be dragged into a spreadsheet and keep the columns correctly. Mostly I wanted it to look nice with a simple text editor so that no extra steps needed to be taken to view the data.

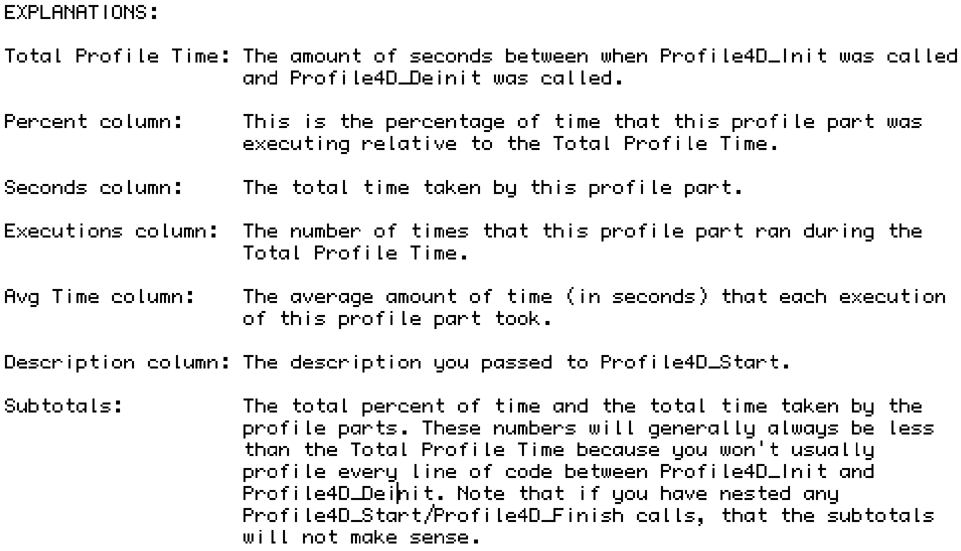
Here is an example (admittedly made up) of a report Profile4D created:



```
Profile4D  Thursday, October 2, 2003 at 6:43 PM.
Running Compiled.  (Best viewed with a monospaced font.)

Percent          Seconds            Executions          Avg Time    Description
======= ================    ==================  ================    ========================
 100.0%          119.994                     1               N/A    **Total Profile Time**

  19.1%           22.870                   240             0.095    SeekFrom
  16.6%           19.959                    80             0.249    Handle headers
  16.6%           19.944                    80             0.249    Take care of body

  11.9%           14.316                     6             2.386    Take care of attachments
   9.3%           11.166                    80             0.140    Handle raw message
   8.3%            9.965                    80             0.125    GetNextFromLine

   8.3%            9.957                    80             0.124    Save records
   1.4%            1.728                    14             0.123    Update message info
------- ----------------
  91.6%          109.905            **Subtotals**
```

The first line in the columns corresponds to time spent in the Main Area (between the `Profile4D_Init`/ `Profile4D_Deinit` pair). Then there is one line each (sorted by total execution time) for all the code blocks you wanted to profile. At the bottom, the subtotals show how all the blocks together stacked up against the total time in the Main Area.

At the bottom of each report the following explanation is given:



```
EXPLANATIONS:

Total Profile Time: The amount of seconds between when Profile4D_Init was called
                    and Profile4D_Deinit was called.

Percent column:     This is the percentage of time that this profile part was
                    executing relative to the Total Profile Time.

Seconds column:     The total time taken by this profile part.

Executions column:  The number of times that this profile part ran during the
                    Total Profile Time.

Avg Time column:    The average amount of time (in seconds) that each execution
                    of this profile part took.

Description column: The description you passed to Profile4D_Start.

Subtotals:          The total percent of time and the total time taken by the
                    profile parts. These numbers will generally always be less
                    than the Total Profile Time because you won't usually
                    profile every line of code between Profile4D_Init and
                    Profile4D_Deinit. Note that if you have nested any
                    Profile4D_Start/Profile4D_Finish calls, that the subtotals
                    will not make sense.
```

Installation

When you install the Profile4D component, you will be installing only the following eleven public methods. Nothing else is installed. The component will compile with All Variables Typed. No inter-process variables are used. A handful of process variables are used and all have the `Profile4D_` prefix.

`Compiler_Profile`
`Profile4D_Init`
`Profile4D_Deinit`
`Profile4D_Pause`
`Profile4D_Start`
`Profile4D_Finish`
`Profile4D_Pad`
`Profile4D_Resume`
`Profile4D_GetMaxDescriptionLeng`
`Profile4D_ArrayResize`
`Profile4D_ArrayAppend`

Parameters

You will only use four methods directly.

`Profile4D_Init`	*No parameters*
`Profile4D_Deinit`	*No parameters*
`Profile4D_Start(Description)`	*Text* (Pass the unique description for this block of code.)
`Profile4D_Finish(Description)`	*Text* (Pass the same value passed to `Profile4D_Start`.)
`Profile4D_Pause`	*No parameters*
`Profile4D_Resume`	*No parameters*

Example

Here is a very simplistic example of a single method we might want to profile. Here is the method before we start:

```
DIAGNOSTIC_BeginMethod (Current method name)
PRINT_LockArea
PRINT_PrepareForNewMessage
PRINT_InsertHeader
PRINT_InsertTextSelection
PRINT_FinishedInsertingMessages
PRINT_DoPrint (True)
PRINT_UnlockArea
DIAGNOSTIC_EndMethod
```

Lets say that we suspect the calls to `PRINT_InsertHeader`, `PRINT_InsertTextSelection`, and `PRINT_DoPrint`. We would modify the code to look like this:

```
DIAGNOSTIC_BeginMethod (Current method name)

Profile4D_Init `We want the main timer to start here
PRINT_LockArea
PRINT_PrepareForNewMessage
Profile4D_Start("PRINT_InsertHeader") `We start a profile part timer
PRINT_InsertHeader
Profile4D_Finish("PRINT_InsertHeader") `We end a profile part timer
Profile4D_Start("PRINT_InsertTextSelection")
PRINT_InsertTextSelection
Profile4D_Finish("PRINT_InsertTextSelection")
PRINT_FinishedInsertingMessages
Profile4D_Start("PRINT_DoPrint")
PRINT_DoPrint (True)
Profile4D_Finish("PRINT_DoPrint")
PRINT_UnlockArea
Profile4D_Deinit `The main timer stops here
```

```
DIAGNOSTIC_EndMethod
```

After running the method we can view the report and see what is taking the longest. Maybe we would find out that `PRINT_InsertTextSelection` is taking 86% of the entire time. So we would next delete all the Profile4D calls except for the Init and Deinit methods. Then we would open `PRINT_InsertTextSelection` and insert some start/finish calls within the method and rerun the method.

Just yesterday I did something similar for an import routine I had. I was able to find two individual lines of code that together were taking 90% of execution time! I had just created the methods that these two lines of code called and assumed that they were pretty fast. Obviously they weren't. So I tried a different approach and tested and the two lines of code went down to 2% of the total! Profiling made a big difference and in a place I hadn't suspected.

Legal Stuff

Profile4D is freeware and may be given or changed however you like. Of course, I'm not responsible for any problems you might encounter so use it at your own risk! However, if you find any bugs I wouldn't mind knowing about them and there is a good chance I might even fix them :-)

Thanks and good luck with your optimizations!

Cannon Smith
P.O. Box 65
Hill Spring, AB Canada
T0K 1E0
(403) 626-3236
~~<can-nyk@agt.net>~~ cannon@synergyfarmsolutions.com
<www.synergyfarmsolutions.com>