

TABLE OF CONTENTS

Overview

How To ...

Compile for Call Profiling

Set up CAP.INI

Control Profiling

Use CapView

Procedures

Finding Most Expensive Function

Finding Critical Path

CapView Windows

Tree Window

List Window

General Overview

The **Microsoft Windows NT** Call Attributed Profiler (CAP) is an analytic tool that helps you optimize the performance of applications. The profiler lets you determine the amount of time the **Microsoft Windows NT** operating system spends executing sections of code.

There are two basic types of profiles, sampling profiles and execution profilers. A sampling profile interrupts the current process at a fixed interval, samples the program counter and uses the set of samples to determine where the program spends most of its time. The accuracy of this method is highest for single threaded systems (where nothing else is using time in the system -- including the system) and small sampling intervals. This was the most common type of profiler found in the DOS world.

Execution profiles on the other hand "debug" the program. This is to say that they cause interrupts (usually from breakpoints) to occur at fixed points in the program and measure the time take between the points. The two most common points to measure (or granularity) are functions (setting breakpoints at the entry and exit points of functions) and lines (setting breakpoints on every source line). This type of debugging is very intrusive and often requires cooperation from the compiler to be successful.

CAP is an execution profiler that uses two different granularities. Functions as defined by the compiler (this requires assistance from the compiler) and DLL entry points. The first level of granularity will yield an approximation, on a function by function basis, of the time spent executing each function. Programs must be compiled with special switches to get this level of information. The second level of granularity is much grosser and basically measures the amount of time spent in a DLL apportioned to each of the different entry points. Note that if call backs are used, these are not profiled as they are not executed via the normal DLL entry point mechanism.

The following measurement methods are supported:

- a) Measuring calls from within an EXE.
- b) Measuring calls from within a DLL.
- c) Measuring calls from an EXE to all of it's DLLs.
- d) Measuring calls from one DLL to all of its DLLs.
- e) Measuring all the calls to specified DLL's, from any EXE or DLL.
- f) Any arbitrary combination of a) through e).

Profiling consists of the following steps:

1. Setup Profilee: Set the program up for profiling: If you want to do a call level profile then you will need to recompile the program to insert code for procedure entry and exit profiling.
2. Setup CAP.INI: Set up the CAP.INI file to list the DLLs and exes to be profiled.
3. Run Profilee: Execute the program to create the profile output.
4. Analysis Output: Use CapView to determine the desired profiling information.

Caveats:

If coff symbol information is not available in an EXE/DLL that is being profiled, "???" is displayed for all function names.

Nonlocal GOTOs (i.e. setjmp/longjump calls) are supported only if you use the CRTDLL version of the runtime libraries. They are not supported if you use the libc.lib version of the libraries. In this case, profiling an application containing nonlocal GOTOs will result in unexpected results.

Execution of exception clauses will cause the profile to become confused unless every function between the exception cause and the excepting instruction is wrapped in a try/finally.

Profiling data is ***NOT*** collected for functions in assembler language programs without user programming support.

CapView: Tree Window

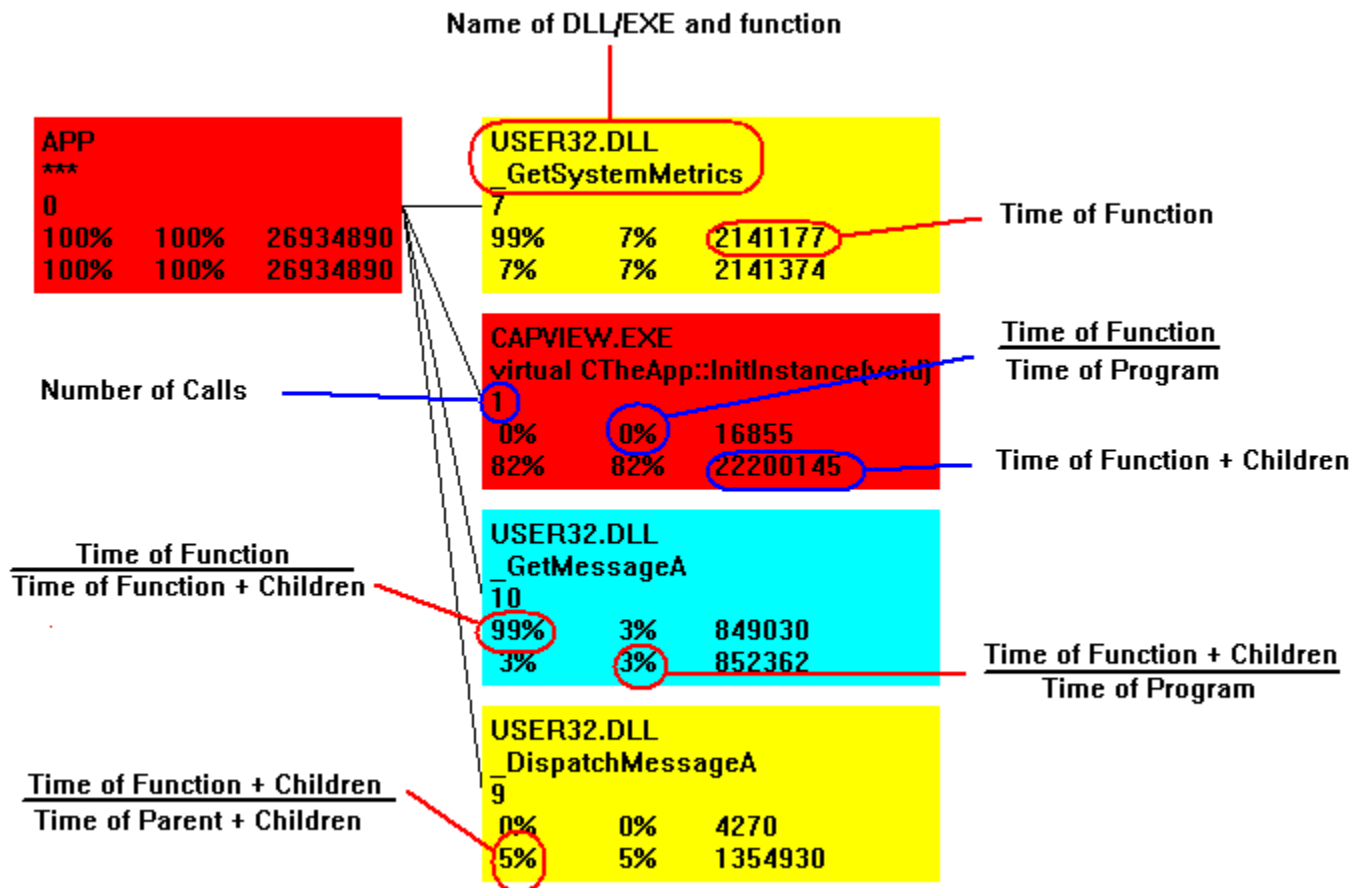
Purpose:

The tree window presents a hierarchical view of the Call Profiler data. The data is presented using the same dynamic call tree that the program actually used. The data however is presented in a more readable format than that emitted directly by the cap profiler.

Each instance of a thread in the data file causes creation of a tree window. Each thread has a "program" node at the root of the thread. DLL initialization routines and the starting routine of the thread are displayed as children of the root node in the tree. (Additional routines may also be specified if the base routine or the DLL initialization routines are not being profiled but they call routines that are being profiled.)

Layout:

The format of each node in the tree is as follows:



The fields displayed:

1. The name of the DLL or EXE module containing the symbol. This will be displayed as ??? if the name of the module cannot be found. This should only occur for DLLs that do not contain an export table.
2. The name of the routine (public symbol) or ??? (static symbol) which was profiled. If the name was generated by the C++ compiler, the name will be undecorated and displayed in a human readable form.
3. The number of times the routine was entered from its parent. If you are looking for the total number of times a routine was called, use the list window.

4. The percentage of time spent in this routine relative to the total amount of time spent in this routine plus all of its children. (Item 6 divided by item 9.)
5. The percentage of time spend in this routine relative to the total amount of time spent executing the entire program.
6. The total amount of time in milliseconds assigned to this routine. If a 'K' is displayed at the end of this value, the time is in seconds.
7. The percentage of time spend in this routine and its children relative to the total time spend in time routines parent and its parent's children.
8. The percentage of time spend in this routine and its children relative to the total time spend in the program.
9. The total amount of time in milliseconds assigned to this routine and its children. If a 'K' is displayed at the end of this value, the time is in seconds.

The boxes have two colors displayed, the primary (or box color) plus the outline color. This coloration is based on the percentages displayed, the current selection of coloration ranges and the *Parent Relative* and *Total* menu items. Below is a chart describing the different combinations. The upper line in each pair is the frame color and the lower line is the box color. The colors are found by taking the correct percentage and looking it up in the range table.

	Parent Relative - Off	Parent Relative -- Checked
Total -- Off	TotalTime/ParentTotal (7)	TotalTime/Parent (7)
	TotalTime/ParentTotal (7)	TimeThisFunc/Parent (4)
Total -- Checked	TotalTime/ProgTime (8)	TotalTime/ProgTime (8)
	TotalTime/ProgTime (8)	TimeThisFunc/ProgTime (5)

Procedures:

This view of the data is generally used for two purposes: location of critical paths and detection of abnormal calling patterns.

Menu Items:

Font: This menu item is used to change the font for the current tree view window. It brings up the common font selection dialog box. Changes will affect only the current window, if one wishes to change the default font for new windows use the Default Font menu item.

Zoom: This menu item toggles between the zoomed and un-zoomed displays of the tree view window. In the zoom display (the default), all of the above listed information is displayed. In the zoomed display no information is listed, rather a small box of the appropriate color is displayed. This provides the ability to get a larger overview of the call tree and look for critical paths.

Parent Relative: This menu item controls coloration of the boxes displayed.

Total: This menu item controls coloration of the boxes displayed.

Set Ranges: This menu item brings up the range selection dialog box.

Other Operations:

Double Click: If you double click on a box in this view, the box is ether expanded or contracted depending on its current state. This allows one to eliminate those paths or nodes that are either not of interest or at least no longer of interest from the call tree display. This pruning is often combined with elimination of the bottom range display to remove un-interesting data. Range based pruning will override the double click pruning, thus one cannot display pruned nodes even if one expands a node containing range pruned items.

Range Dialog Box

Colors are used to help set focus on the information most in need of examination. The color assigned to a block or line is based on one of the percentage values and the definition of the ranges. The default set of ranges uses 5 range blocks:

RED	20% - 100%
YELLOW	5% -- 20%
CYAN	2% -- 5%
GREEN	1% -- 2%
BLUE	Less than 1%

Note that the ranges are set up in a logarithmic fashion. This is due to the fact that as you walk down the call tree times tend to drop in a logarithmic order. Additionally, this represents the best areas to focus on for optimizations. Reducing a routine that takes less than one percent of the total time of a program by 100% only reduces the total program time by 1%. However reducing a routine that takes 20% of the time by 10% will reduce the total program by 2 percent. Much less effort yields the same amount of total time reduction.

Operations:

Color: This button invokes the common color dialog box to change the color of the currently selected range.

Add: This button is used to create a new range. One of the above or below values should be changed. This will then create a new range splitting the currently selected range on the changed value. After a range is added one should change the color for it.

Delete: This button is used to remove a range. The selected range is combined with the range below it.

Help: This button brings up this help item.

Change: This button is used to accept the set of changes edited it. Until this button is pressed changing the color, or text color is not actually completed.

Text Color: This button allows the user to select either black or white text on the background color. This choice should be made to make the text most readable.

Above: This edit field contains the top of the range. The top of the range may be changed by editing in a new button and pressing the change button.

Below: This edit field contains the bottom of the range. The bottom of the range may be changed by editing in a new value and pressing the change button.

Prune Out Range: This check box is only valid for the tree window. If it is checked and the function plus child as a percentage of the total program time falls in the first range, the box will not be displayed in the tree window. This check box has no effect in the list window.

Setting up the CAP.INI File

In order to profile you must setup a CAP.INI file. This file must reside on the root of the C: drive. The ini file controls the set of items profiled. The format of the ini file is as follows:

[EXES]

A list of applications to be profiled. Each application must be on its own line. When CAP.DLL runs its initializer, it checks the current executable name against this list and will start profiling if the name is on the list. If the name is not on the list, profiling is not done.

[PATCH IMPORTS]

List of DLLs/Exes to be profiled for all of their imported entries. Each name should be on a new line. A DLL or EXE should not appear in this field if the EXE/DLL (or a DLL imported by the EXE/DLL) was compiled for call profiling.

[PATCH CALLERS]

List of DLLs to be profiled for their exported entries if called from the applications (listed in the [EXE] section) or any of their DLLs. Each name should be on a new line.

[NAME LENGTH]

The maximum length of a symbol. This number must be in the range from 20 to 2048. It is recommended that for C++ program this value be set to at least 128 due to the name mangling that is performed by the linker. If a symbol is longer than this value, it is truncated. If the field is not specified or is 0 then the value defaults to 40. This field is optional.

All sections must be present in the CAP.INI file, but the contents of any field may be left blank.

Example of a CAP.INI file

Example of a CAP.INI File

The following CAP.INI file does the following:

It will profile the executables WINDBG.EXE, CAPVIEW.EXE and SYMEDIT.EXE.

For the above executables, it will profile all calls into the DLLs' USER32.DLL, KERNEL32.DLL, GDI32.DLL, ADVAPI32.DLL and CRTDLL.DLL.

It will allow 128 characters for symbol names rather than the standard 40.

[EXES]

WINDBG.EXE
CAPVIEW.EXE
SYMEDIT.EXE

[PATCH IMPORTS]

[PATCH CALLERS]

USER32.DLL
KERNEL32.DLL
GDI32.DLL
ADVAPI32.DLL
CRTDLL.DLL

[NAME LENGTH]

128

Controlling Profiling

While generally one wants to profile the entire execution of a program, there are times where it is desirable to only profile a small portion of a program. The CAP system currently has three methods to control the extent of profiling done. These methods are:

1. CAP.DLL exported functions: These functions can be used to programmically control profiling under the control of the program being profiled. Functions are provided to start profiling, stop profiling and dump the current profile data set.
2. CAPDUMPEXE: This program is used to externally control the starting and ending points for profiling a program. The program may be used to start profiling of a program, stop profiling and dump the profile data to disk.
3. CAPSETUPEXE: Attaches CAP to all 32-bit Windows programs.

Profiling information is dumped when the program exits.

Output from the profiler will be appended to the file BASE.END where BASE is the base name of the program being profiled. This permits multiple runs to be accumulated in a single location and analysis in one step.

If all that is desired is to get a full run of a program then the methods of controlling profiling listed above may be ignored. Running the program to be profiled yields a full program of the program from start to finish.

CAP.DLL Exported Functions

The exported entry points listed below can be used to control profiling certain sections of code. Note that profiling will not occur if the EXE is not listed in the CAP.INI file.

The functions exported from CAP.DLL for the purposes of controlling the extent of profiling are:

StartCAP() : This function is used to start a profiling run. The function will clear out any data currently in the profiling buffers and initiate profiling from the current point on. This function will start profiling on all threads in a multi-threaded application at their current program counters. When the program starts, profiling is always started. Calling StartCAP() will eliminate all data previously collected and start from scratch.

StopCAP() : This function is used to stop a profiling run. This function will stop profiling on all threads in application.

DumpCAP() : This function is used to dump the current set of collected data into the output file. The data will be appended to the end of the output file so that multiple calls will not overwrite each other. This function causes profiling to be suspended and if called multiple times will do multiple dumps.

There is no support for pausing and resuming profiling. The only way to have multiple runs combined together is to have CapView perform a merge.

Example:

```
StartCAP();    // Clear existing profiling data and restart profiling.
..
..            // application's code
..

StopCAP();     // Stop profiling without dumping data.
DumpCAP();     // Dump profiling data to FOO.CAP file.
```

Using CapDump

CapDump is a program that is used to externally control the extent of profiling being done for an application. CAPDUMP.EXE can start profiling, stop profiling and dump profiling data for all the applications being profiled, at any time.

The following options are available via CAPDUMP.EXE:

Stop: Stops profiling (applications continue to run).

Clear and Restart: Clears any existing profiling data and restarts profiling.

Dump and Stop: Dumps any existing profiling data and stops profiling (applications continue to run).

Data is dumped to a text file using the profiling applications name with .CAP extension. All fields are tab separated. Data is appended to data files with each dump.

If calls are being profiled when data clearing is requested, the time of clearing is used as the starting time for those calls.

Using CapSetup

CapSetup is used as a short hand way of listing every DLL in the system in the PATCH CALLERS section of the CAP.INI file. When you attach CAP.DLL to all of the DLLs in the system they will be automatically profiled (if listed in the Exes section of the CAP.INI file) whenever started and for all DLLs in the system.

Usage: CapSetup A | D

- A Attaches CAP.dll to all Windows applications
- D Detaches CAP.dll from all Windows applications

Notes:

- 1) Administrative privileges are required in order to run CapSetup.
- 2) System needs to be rebooted in order for the change to take effect.

Compiling for Profiling

If you want to do call level profiling, you need to do some special compiling to have a special profiling function called at the start and end of each function. If all you are interested in is profiling the exported functions of a DLL then this step should be omitted.

For EXPORTED function profiling only:

Link the debuggee with coff symbol information (-debugtype:coff or -debugtype:both).

For call level profiling:

1. Add the options -Gh and -Zd to the compiler line for the **Microsoft C** compilers. For Mips, add the -Od flag to turn off optimizations.
2. Add the **cap.lib** library to the link command line.
3. Link the debuggee with coff symbol information (-debugtype:coff or -debugtype:both).

See Also [Profiling Assembly](#)

Overview of CapView

CapView is a sample program provided with the Win32 SDK that can be used to provide a visual method of looking at the output from the CAP profiler.

The output from cap is a large table (a normal profile of CapView is over 8000 lines long), the amount of information requires tools to deal with presenting the information in a better manner. CapView was written to address this problem. CapView re-interprets the data and presents it in two different views:

Views:

Call Tree View: The call tree view is designed to present the profile data based on the dynamic call tree. This view allows the user to search out critical paths and locating unexpected calls to functions. This is accomplished by allow uses to expand and contract nodes in the tree view. Nodes in the tree are also colored according to criteria selected by the user to provide an immediate feedback on critical paths.

Function List View: The list view is designed to allow for examining the profile data on a function by function basis. This view presents information from all occurrences of a function independent of where it was called. All occurrences of a function will be lumped together into a single entry in the list window.

CapView creates two windows for every thread in the profile data file. Since the profiler appends data to the end of existing files a thread may occur multiple times. A new pair of windows will be created for each occurrence of a thread. (Future versions should permit combining of threads into a single view.)

Command Line:

CapView <file name>

A single file name may be specified on the command line. This file will be read in on startup. There are currently no options to CapView.

Public Name Decoration:

The form of function names listed is the public name for C code and an undecorated form for C++ method. There is currently no way from the user interface to control the amount of information about a C++ method which is reconstructed from the public name and displayed. No undecoration is done for C functions since the mapping cannot necessarily be undone and is generally obvious. Static C functions are generally displayed as "???" since the function name is missing from the debug information.

See also: Finding Critical Paths, Finding High Use Procedures, Finding Unexpected Calls, List View, Tree View.

CapView: List Window

Purpose:

The list window presents a view of the profile data that is based on individual functions. All of the data about one function is gathered together and placed on a single line. This view permits the user to locate functions displaying unexpected calling patterns. Functions that are taking far too much time on a per call basis or functions that are called far too many times.

Every instance of a thread in the profile data file causes a list window to be created. Future versions of CapView will have the provision to combine multiple instances of threads into a single view.

Layout:

The list window is laid out one line per function. All information about a single function is combined into this single entry in the table. Not all of the information placed in the output file is carried into this table. Specifically the program does not remember the maximum and minimum times for a routine.

The information displayed for each function:

Name: The name of the DLL/EXE containing the function plus the name of the function. The name displayed will be converted from the public name to a human readable for C++ publics.

Num Calls: The total number of all calls into the function.

Function Only -- Function: The total number of milli-seconds attributed to this function plus the percent of the total time attributed to this function as a percentage of the total program time.

Function Only -- Per Call: The total number of milliseconds attributed to this function divided by the number of calls to this function. Plus this value expressed as a percentage of the total program time.

Function + Children -- Function: The total number of milliseconds attributed to this function plus all of its children. In addition this value expressed as a percentage of the total program time. This value may actually be greater than the total program time. If a function is recursive, a function's contribution is added multiple times, both for the function itself and as a part of the calling tree from the root function.

Function + Children -- PerCall: The total number of milli-seconds attributed to this function plus all of its children divided by the number of calls to the function. In addition this value expressed as a percentage of the total program time.

The table may be sorted according to each of the columns in the table. The current sort may be determined by examining the Options menu.

Coloration is always done based on the function's percentage of the total time of the program. Coloration is changed using the Options Set Ranges dialog box.

Procedures:

This view is used to detect unexpected number of calls to any particular routine and to check for routines that are taking too long. If a routine takes less than 5 percent of the total program time it may not be worth while optimizing the routine unless the number of calls to it can be reduced.

Menu Items:

Sort By Time: If this menu item is checked, the table is sorted by the total time attributed to each function.

Sort By Alpha: If this menu item is checked, the table is sorted by the name of each function.

Sort By Calls: If this menu item is checked, the table is sorted by the total number of calls into each function.

Sort By Per Call: If this menu item is checked, the table is sorted by the per call time attributed to each function.

Include Children: If this menu item is checked, the time and per call sorts use the function plus children values

rather than the function only values.

Font: This menu item causes the common font dialog box to be opened. Font changes will affect only this instance of the list window.

Set Ranges: This menu item causes the set range dialog box to be opened. It can be used to affect the coloration. Coloration is always based on the function's percentage of the total program.

Profiling Assembly Files

Assembly files may also be profiled. However this requires intervention on the part of the writer of the assembly file. In order to profile an assembly file, **_penter** must be called on entry all functions.

For ix86 assembly code:

1. **_penter** must be called before any prolog code is executed.
2. The function takes no arguments and will destroy the contents of the **eax** register.
3. The function assumes that the first item on the stack is the return address of the current procedure. The routine will modify this to return to someplace else first.

For MIPS and Alpha assembly code:

Profiling of assembly code is not currently supported.

Suppose we have Windows .EXE's called ZOOMAN and HUNTED, and .DLLs called ELEPHANT, MONKEY, SNAKE, WATER, and FOOD. Let's assume the following intercall dependencies exist:

DLL/EXE name	DLLs which are imported
zooman.exe calls:	water.dll, food.dll
hunted.exe calls:	elephant.dll, monkey.dll, snake.dll
water.dll calls	none
food.dll calls	none
elephant.dll calls	water.dll, food.dll
monkey.dll calls	water.dll, food.dll
snake.dll calls	none

Suppose we have Windows .EXE's called ZOOMAN and HUNTED, and .DLLs called ELEPHANT, MONKEY, SNAKE, WATER, and FOOD. Let's assume the following intercall dependencies exist:

zooman.exe calls:

water.dll
food.dll

hunted.exe

elephant.dll
water.dll
food.dll
monkey.dll
water.dll
food.dll
snake.dll

a) Measuring calls from within ZOOMAN.EXE:

1. Recompile ZOOMAN and link it with CAP.LIB. (See Compiling for CAP.)
2. Setup CAP.INI as follows:
[EXES]
zooman.exe

[PATCH IMPORTS]

[PATCH CALLERS]

All the C functions in ZOOMAN.EXE will be profiled by CAP.

b) Measuring calls from within WATER.DLL:

1. Recompile WATER and link it with CAP.LIB.
2. Setup CAP.INI as follows:
[EXES]
zooman.exe
hunted.exe

[PATCH IMPORTS]

[PATCH CALLERS]

Both ZOOMAN.EXE and HUNTED.EXE will be profiled for all the C functions within WATER.DLL.

c) Measuring calls from HUNTED.EXE to it's DLLs:

1. Run CapSetup.exe to attach CAP.DLL to all Windows Applications, and reboot the system. (See Control Profiling.)

2. Setup CAP.INI as follows:

[EXES]
hunted.exe

[PATCH IMPORTS]
hunted.exe

[PATCH CALLERS]

All the calls from HUNTED.EXE to ELEPHANT.DLL, MONKEY.DLL, and SNAKE.DLL will be profiled. These are all the C functions exported by ELEPHANT, MONKEY, and SNAKE DLLs which are used by HUNTED.exe.

d) Measuring calls from ELEPHANT.DLL and MONKEY.DLL to their DLLs:

1. Run CapSetup.exe to attach CAP.DLL to all Windows Applications, and reboot the system.

2. Setup CAP.INI as follows:

[EXES]
hunted.exe

[PATCH IMPORTS]
elephant.dll
monkey.dll

[PATCH CALLERS]

All calls from ELEPHANT.DLL and MONKEY.DLL to WATER.DLL and FOOD.DLL will be measured. These are all the C functions exported by WATER and FOOD DLLs which are used by either ELEPHANT.DLL or MONKEY.DLL.

e) Measuring all the calls to FOOD.DLL:

1. Run CapSetup.exe to attach CAP.DLL to all Windows Applications, and reboot the system.

2. Setup CAP.INI as follows:

[EXES]
zooman.exe
hunted.exe

[PATCH IMPORTS]

[PATCH CALLERS]
food.dll

Both ZOOMAN.EXE and HUNTED.exe will be profiled separately for all the calls to FOOD.DLL. For ZOOMAN.EXE, these are all the C functions exported by FOOD.DLL which are used by ZOOMAN.EXE. And for HUNTED.EXE these are all the C functions exported by FOOD.DLL which are used by either ELEPAHNT.DLL or MONKEY.DLL.

a+c) Measuring calls from within ZOOMAN.EXE and calls to it's DLLs:

1. Recompile ZOOMAN and link it with CAP.LIB as described in section 3.
2. Setup CAP.INI as follows:

```
[EXES]
zooman.exe
```

```
[PATCH IMPORTS]
zooman.exe
```

```
[PATCH CALLERS]
```

c+d) Measuring calls from HUNTED.EXE to it's DLLs and calls from ELEPHANT and MONKEY DLLs to their DLLs:

1. Run CapSetup.exe to attach CAP.DLL to all Windows Applications, and reboot the system.
2. Setup CAP.INI as follows:

```
[EXES]
hunted.exe
```

```
[PATCH IMPORTS]
hunted.exe
elephant.dll
monkey.dll
```

```
[PATCH CALLERS]
```

a+c+d) Measuring calls from within HUNTED.EXE and calls to its DLLs plus calls from ELEPHANT and MONKEY DLLs to their DLLs:

1. Recompile HUNTED and link it with CAP.LIB.
2. Setup CAP.INI as follows:

```
[EXES]
hunted.exe
```

```
[PATCH IMPORTS]
hunted.exe
elephant.dll
monkey.dll
```

```
[PATCH CALLERS]
```

b+d) Measuring calls from within ELEPHANT.DLL and calls to its DLLs:

1. Recompile ELEPHANT and link it with CAP.LIB.
2. Setup CAP.INI as follows:

```
[EXES]
hunted.exe
```

```
[PATCH IMPORTS]
elephant.dll
```

```
[PATCH CALLERS]
```

Examples of Illegal CAP.INI files:

Do NOT measure calls within a DLL if the DLL's exported functions are being measured from another EXE/DLL. The following example shows this incorrect combination which should be avoided:

x) Measuring calls from ZOOMAN.EXE to its DLLs and calls within WATER.DLL:

1. Recompile WATER and link it with CAP.LIB as described in section 3.
2. Setup CAP.INI as follows:

```
[EXES]
zooman.exe
```

```
[PATCH IMPORTS]
zooman.exe
```

```
[PATCH CALLERS]
```

Note that in addition to measuring all C functions (including the exported routines) within WATER.DLL, all the C functions exported by WATER.DLL which are used by ZOOMAN.EXE are measured. This means that the same function within WATER.DLL will be measured twice. These types of scenarios are not supported and will result in unexpected behavior.

Call level profiling is used to profile based on the function as the smallest profiled item.

