

1 Introduction

From the functional point of view, the profiler can be used to profile functions and/or data.

1.1 Functions Profiler

Functions profiler helps identifying performance bottlenecks. The user can find which functions are most time consuming or time critical and need to be optimized.

Its functionality is based on the trace, recording entries and exits from profiled functions. A profiler area can be any section of code with a single entry point and one or more exit points. Existing functions profiler concept does not support exiting two or more functions through the same exit point. Exit point can belong to one function only. In such cases, the application needs to be modified to comply with this rule or alternatively data profiler with code arming can be used in order to obtain functions profiler results.

The nature of the functions profiler requires quality high-level debug information containing addresses of function entry and exit points, or such areas must be setup manually. Profiler recordings are statistically processed and for each function the following information is calculated:

- total execution time
- minimum, maximum and average execution time
- number of executions/calls
- minimum, maximum and average period between calls

Functions profiler relies on a fact that recorded instructions are executed. This can be a problem on CPUs with the internal CPU pipeline, where fetched cycles are dropped from the CPU pipeline before they come to the execute stage due to the change in the program. For instance few instructions following executed branch or return from subroutine instruction are fetched but not executed. Hence, functions profiler is supported on development systems with full CPU bus trace being capable to distinguishing between fetched and executed cycles, and on development systems with message based trace, which provide information on executed program only by default.

1.2 Data Profiler

While functions profiler is based on analyzing code execution, data profiler performs time statistics on the profiled data objects, which are typically global variables. Typical use cases are task profiler and functions profiler based on code instrumentation.

Task Profiler

When an operating system is used in the application, task profiler can be used to analyze task switching. When the task profiler is used in conjunction with the functions profiler, functions' execution can be analyzed for each task individually.

Code instrumentation

Newer development systems may not provide full CPU bus trace on which functions profiler can be implemented without troubles. Such development systems (e.g. MPC5500, V850ES/Fx3, ARM ETM, MPC56x) normally provide a message based trace. Consequentially, a standard long lasting functions profiler is no longer

a default feature since normally the on-chip message based trace provides a very limited amount of address qualifiers or even non.

When there is no better alternative (e.g. on-line profiler based on Nexus RTR on MPC5500, ETM RTR on ARM7/9 and RTR Execution on V850ES/Fx3), the user can use data profiler in order to obtain functions profiler results. This approach requires code arming and an on-chip trace, which allows recording write cycles into a single data variable (data trace or OTM trace). CPUs featuring ownership trace (OTM) include a so-called dedicated ownership trace register (OTR) and the on-chip trace can be configured in a way that only accesses to the OTR are recorded. Ownership trace is available for instance on Freescale MPC56x, Freescale MAC7100/7200 and National Semiconductor CR16 family. Profiled functions must be armed with minimum source code at each function entry and exit point, writing a specific value to the OTR or to the predefined data variable. Note that these results provide very accurate information on functions execution time.

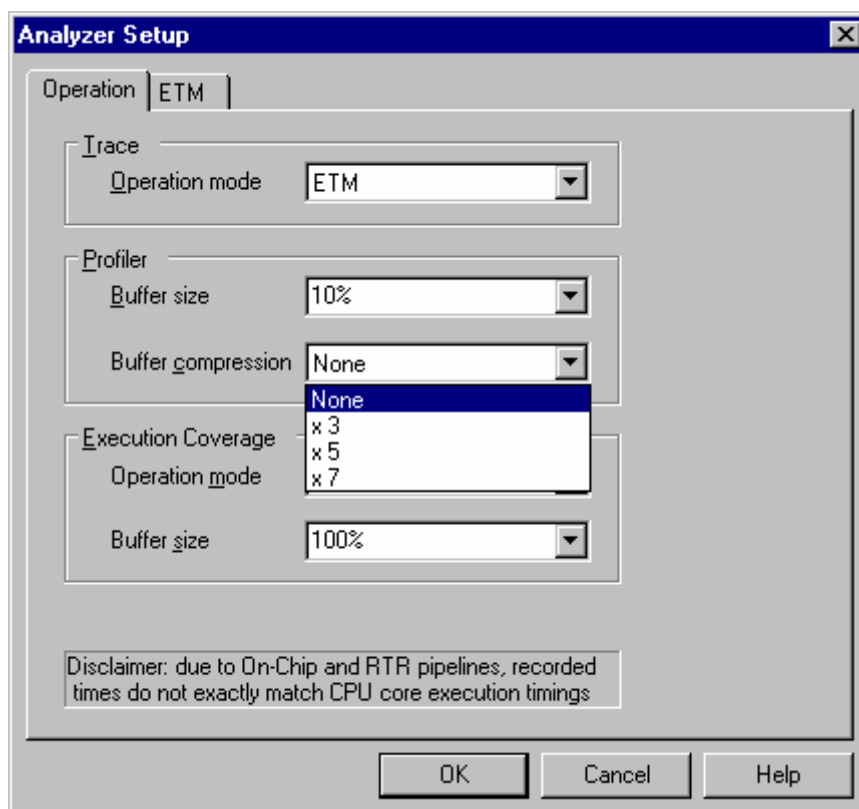
Some tool vendors advertise functions profiler functionality based on other techniques like periodically sampling the program counter, which don't yield exact results, or impose severe restrictions on the run-time environment.

Code arming technique can be used also on development systems with full bus trace however this makes sense only for CPUs with internal pipeline, where the tool cannot distinguish between fetched and executed cycles among the recorded cycles. (e.g. Freescale 68K and Freescale CPU32 family).

2 Hardware Operation

Depending on the architecture and the development system, profiler can operate either in off-line or real-time operation mode.

2.1 Off-line Profiler



Initially, off-line profiler has been introduced on on-chip debug emulation systems with the message based trace port, where a concept used in the in-circuit emulation systems seemed to be not feasible. A standard trace is used to collect the program flow and then software analysis is performed off-line to obtain the profiler results. Time of program execution being tested directly depends on the target CPU speed and the trace buffer size. Upload time to the PC depends on the trace buffer size and the iC3000 unit. In case of iTRACE GT or Active GT development platform with 1GB or more trace buffer, it is recommended to use ic3000GT unit which offers a high-speed trace upload to the PC. Off-line profiler buffer size is configurable in the

'Hardware/Analyzer Setup' dialog (1, 10 or 100%). Additionally, off-line profiler offers "trace" buffer compression selection, which is set to 'None' per default. Setting any other compression rate increases a total recording time but by sacrificing time stamp information. The higher the compression rate is more trace messages are packed into a single trace buffer frame together with single time stamp information. In worst case, this may yield packing several messages belonging to two or even more functions' calls into a single frame. Later when the trace buffer is loaded to the PC, the debugger would reconstruct a complete execution flow for

these functions but all the code belonging to the packed messages would have the same time information. In such case, profiling these functions doesn't work since profiler metric is based on measuring time between function entry and exit. For this particular case, displayed time difference would be zero which is not correct. Hence, if it is really required to prolong the profiler total session time, use compression with **caution!**

2.2 Real-time Profiler

Real-time profiler is based on a hardware logic, which in real-time monitors profiled addresses only. This yields longer total recording time comparing to the off-line profiler. For instance, to profile a function it is necessary to record only function's entry and function exit point, which real-time profiler does. However, off-line profiler records function's entry and exit and all the code executed in between and thereby wastes valuable trace buffer by recording information irrelevant for such particular profiler session.

Real-time profiler is available for all in-circuit emulation systems with bus trace (including V850ES/Fx3 ActivePRO POD, which has different trace port type) and on-chip debug emulation systems with trace port, where iSYSTEM proprietary RTR technology is implemented (MPC55xx except for MPC551x, MPC56x, ARM ETM)

2.3 Upload While Sampling

Typically, the execution time of the profiler is limited by the trace buffer size and cannot run infinitely. iSYSTEM now offers a distinct 'upload while sampling' feature on their high-end development systems. This allows infinite profiler, when the CPU runs slow enough that the trace buffer is uploaded faster than the new trace messages, coming from the CPU trace port, are filling the trace buffer. Even when the CPU runs at higher frequencies, 1GB (and more) trace buffer with 'upload while sampling' capability can significantly increase the total recording time.

This functionality requires no configuration and is operational on certain development systems (iC3000GT + iTRACE GT interface card + iTRACE GT) per default.

3 Toolbars



Toolbars are explained in order from the left to the right.

3.1 Analyzer Tools

Profile

The profiler allows profiling of Functions and Data objects (including OS objects), when the dev

The Function profiler is realized by tracing function entries and exits.

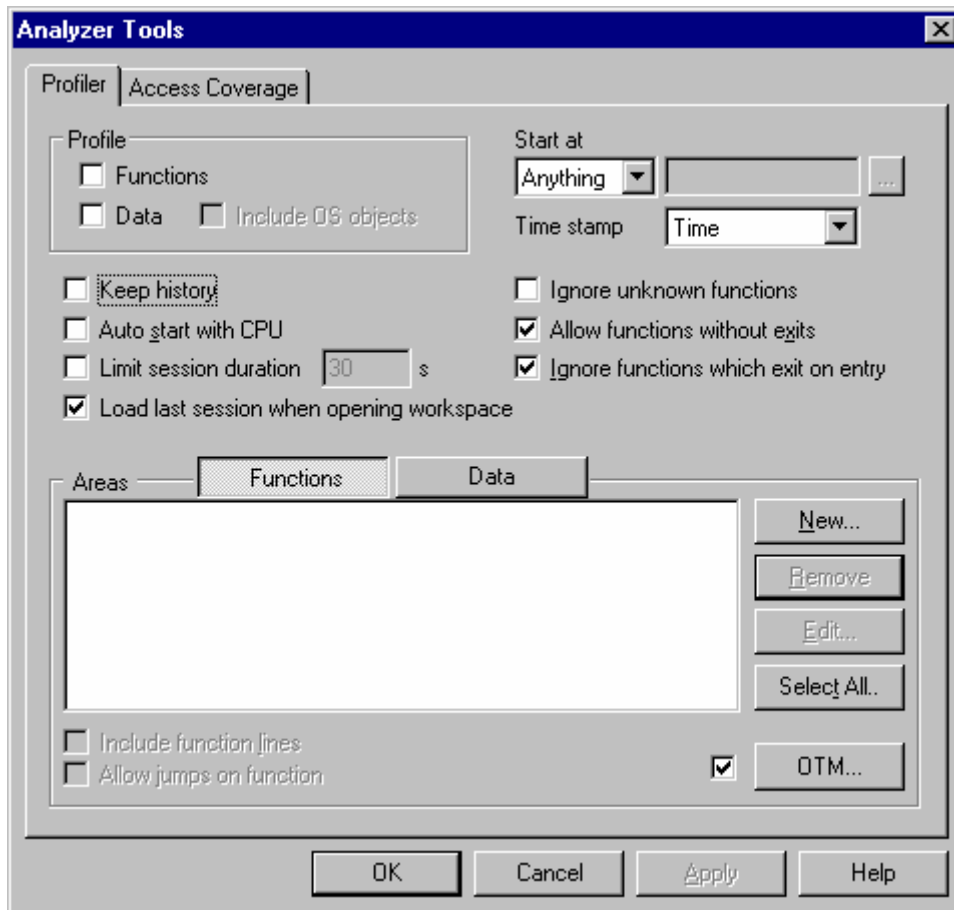
Data profiler is realized by tracing write accesses to a variable. Not all development system may feature data profiler.

Auto start with CPU

When checked, the winIDEA will start the profiler whenever winIDEA starts the CPU.

Start at

The starting point determines the program address that must be executed for the profiler to start. You may use this option to synchronize multiple profiler sessions on the same event.



Analyzer Tools dialog

Keep history

When checked, full history is maintained. When using deep buffers, this option may need to be unchecked to prevent system resource drain.

Limit session duration

The profiler session duration is limited to the specified number of seconds when checked. After the specified duration, the profiler stops recording.

Load last session when opening workspace

When checked, the last session will be reloaded when the workspace is opened.

Ignore unknown functions

When checked, winIDEA will ignore (instead of reporting an error) functions who were for instance manually defined but no match can be found in the symbol table.

Allow functions without exits

When checked, profiler allows functions without exits (typically main on embedded applications).

It is legal for a function to have no exit, but on some occasions the compiler might choose an instruction sequence other than the return instruction to exit from a function (Z80 example: RET = POP HL / JP(HL)). winIDEA does not detect such exit points and profiling such functions will cause a profiler error.

Ignore functions which exit on entry

When checked, winIDEA will ignore (instead of reporting an error) functions whose only instruction is the return statement. Such functions defeat the concept of the profiler, which requires separate entry and exit point(s).

3.2 Start Profiling

Starts profiler session.

3.3 Stop Profiling

Stops profiler session

3.4 Load Saved Data

Loads saved profiler session.

3.5 Save Recorded Data

Saves current profiler session.

3.6 Show statistics view

Shows profiler results in statistics view.

3.7 Show history view

Shows profiler results in history view.

3.8 Show Code

Shows Functions profiler results.

3.9 Show Data

Shows Data profiler results.

4 Examples

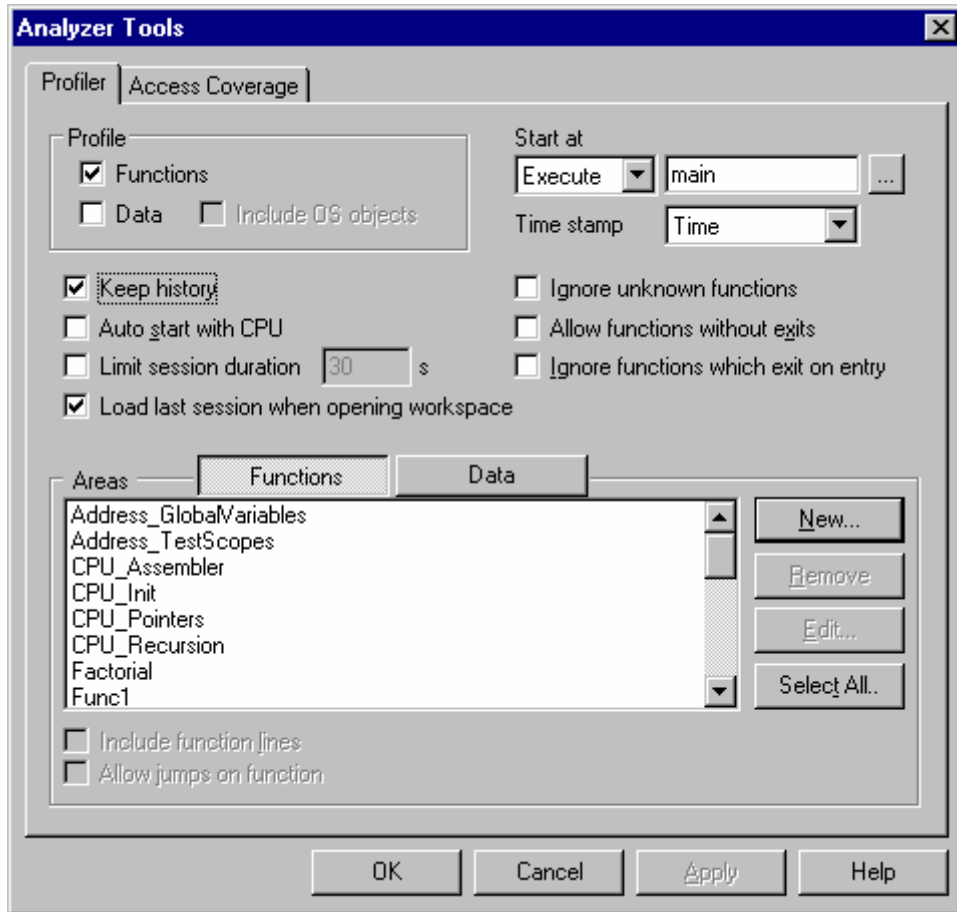
4.1 Example 1

A TMS470 application is profiled to get an overview on how much time the CPU spends in each C function.

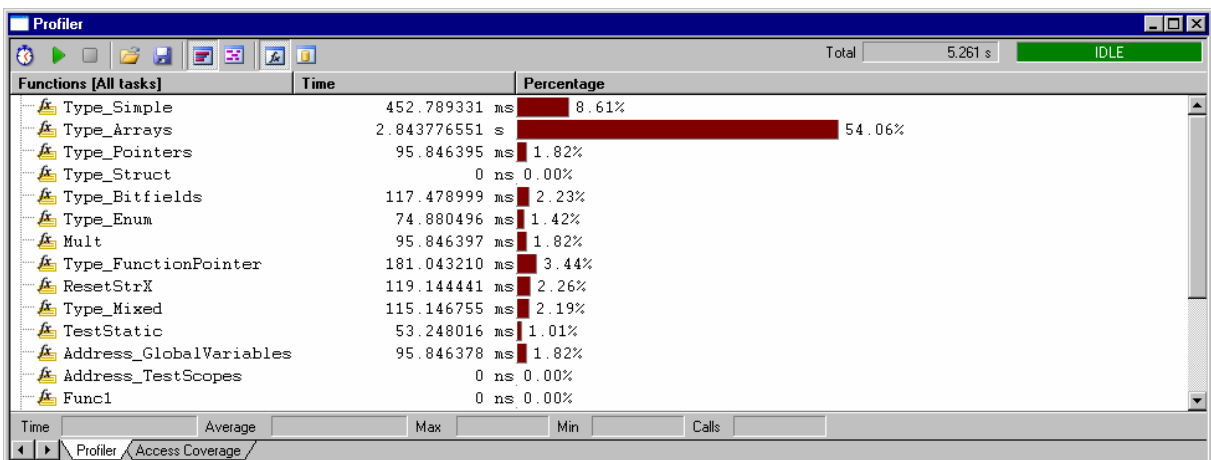
- Open the 'Analyzer Tools' dialog by pressing the 'Analyzer Tools' toolbar in the Profiler window.
- Check 'Functions' option in the 'Profile' field.
- Select function `main` for the start point or keep it blank, which yields start as soon as the application is running and the profiler is started.
- Press 'Functions' button in the 'Areas' field, then press 'New...' button and use default 'All functions' setting.
- Now, all project functions are defined for profiled areas.

- When profiler history view is required, make sure that the ‘Keep history’ option is checked. This slightly slows down the profiler operation since the recording results are kept in the PC memory, which are otherwise dumped after the profiler statistics is calculated.

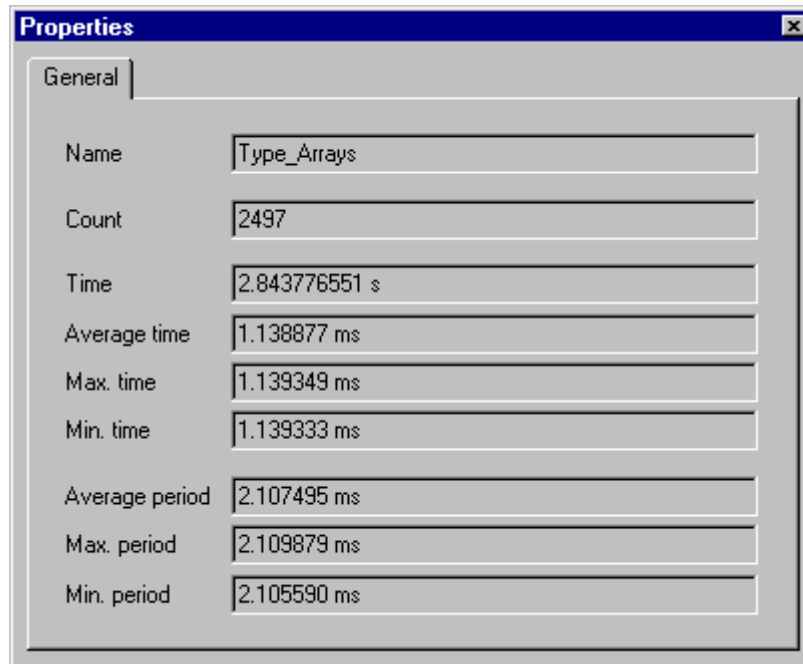
Next picture depicts current functions profiler settings.



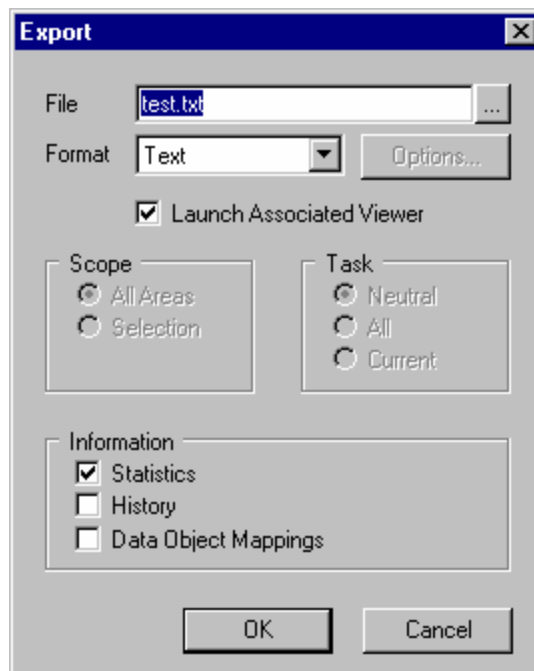
- Reset the system, start the profiler and run the application. Profiler stops recording after the trace buffer fills up or the program is stopped and then displays the results. Let’s inspect the profiler results.
- Check if ‘Show code’ toolbar is pressed and then press ‘Show statistics view’ toolbar for statistics view.



- Click on a specific function (e.g. Type_Arrays in our example) in the profiler window and select ‘Properties’ from the local menu to obtain all the profiler details for this particular function.

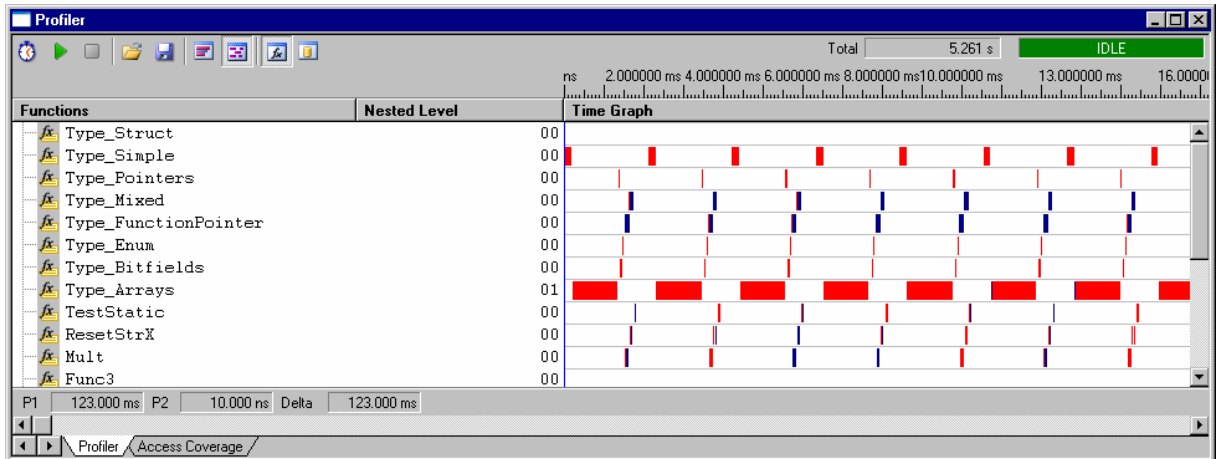


Details (Statistics, History) of all profiled functions can be easily exported in a text file using the 'Export...' from the local menu (right mouse click) in the profiler window.



When it's necessary to obtain more details about behaviour within a profiled function (for instance, in which source line or part of the function the CPU spends most of the time), include source lines when adding a function to the profiled areas. Note that more profiler areas defined means less total session time due to the limited trace buffer depth. This applies for real-time profiler operation mode.

- Press the 'Show history view' toolbar for history view.

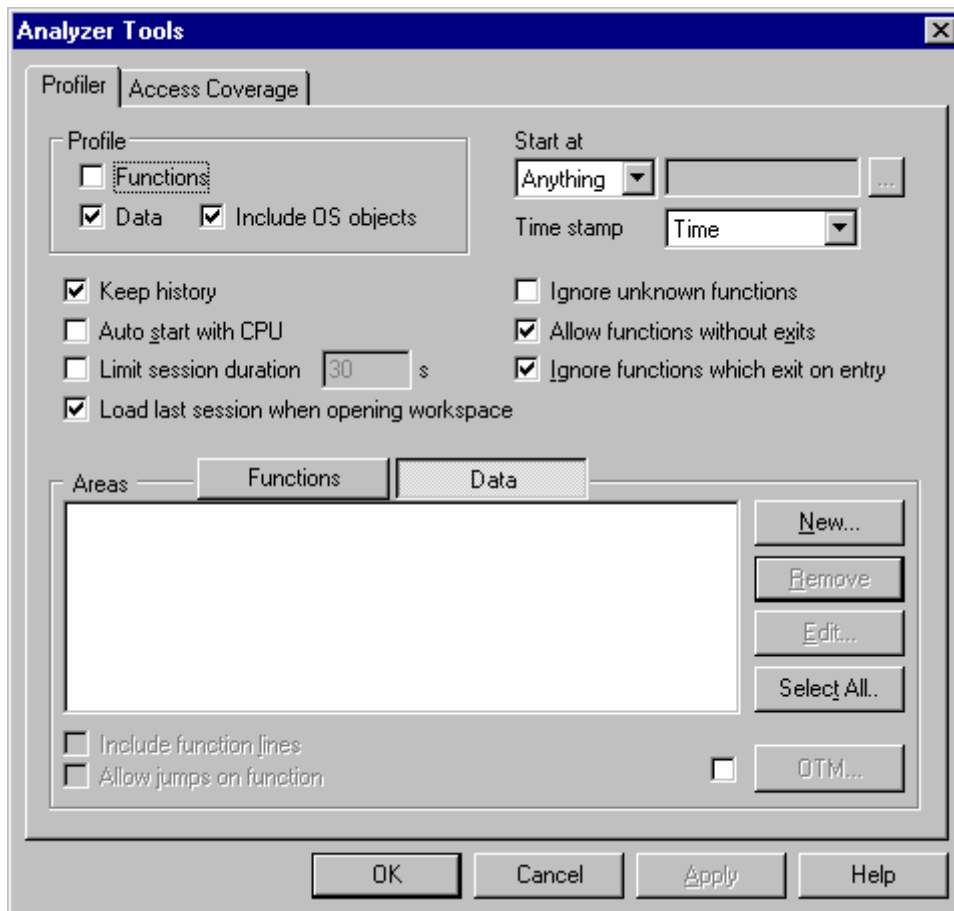


Two markers can be set in the history view, which allow time measurements. Left mouse click sets marker 1 and Ctrl + left mouse click sets marker 2. Time difference (Delta) between the markers can be seen at the bottom in the profiler window.

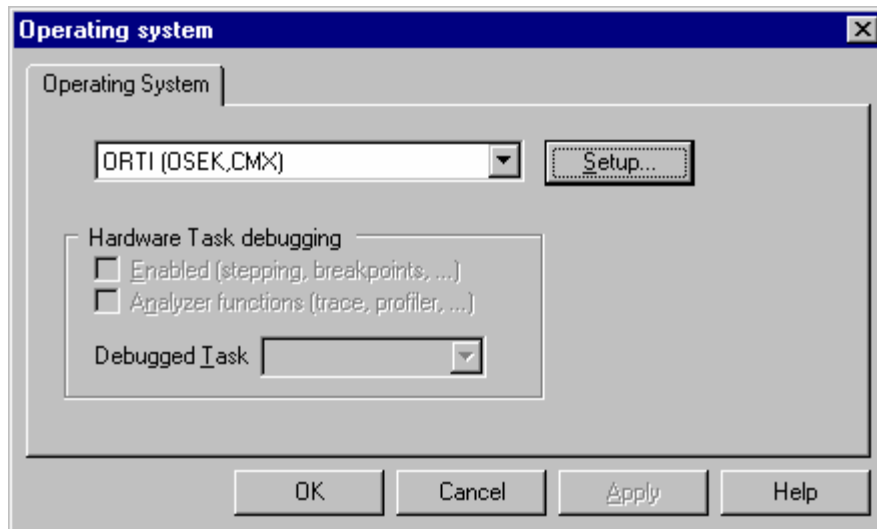
4.2 Example 2

The development environment is built around Freescale MPC5554 platform running OSEK operating system. Setup task profiler using data profiler and functions profiler to profile all C functions.

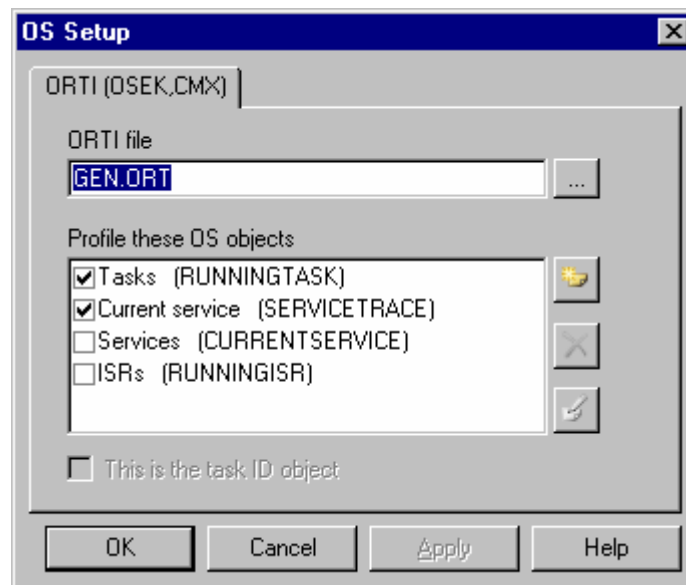
OSEK task profiler is setup by checking 'Data' and 'Include OS objects in the profiler configuration dialog.



It's presumed that the OSEK awareness is already configured in the 'Debug/Operating System' dialog.

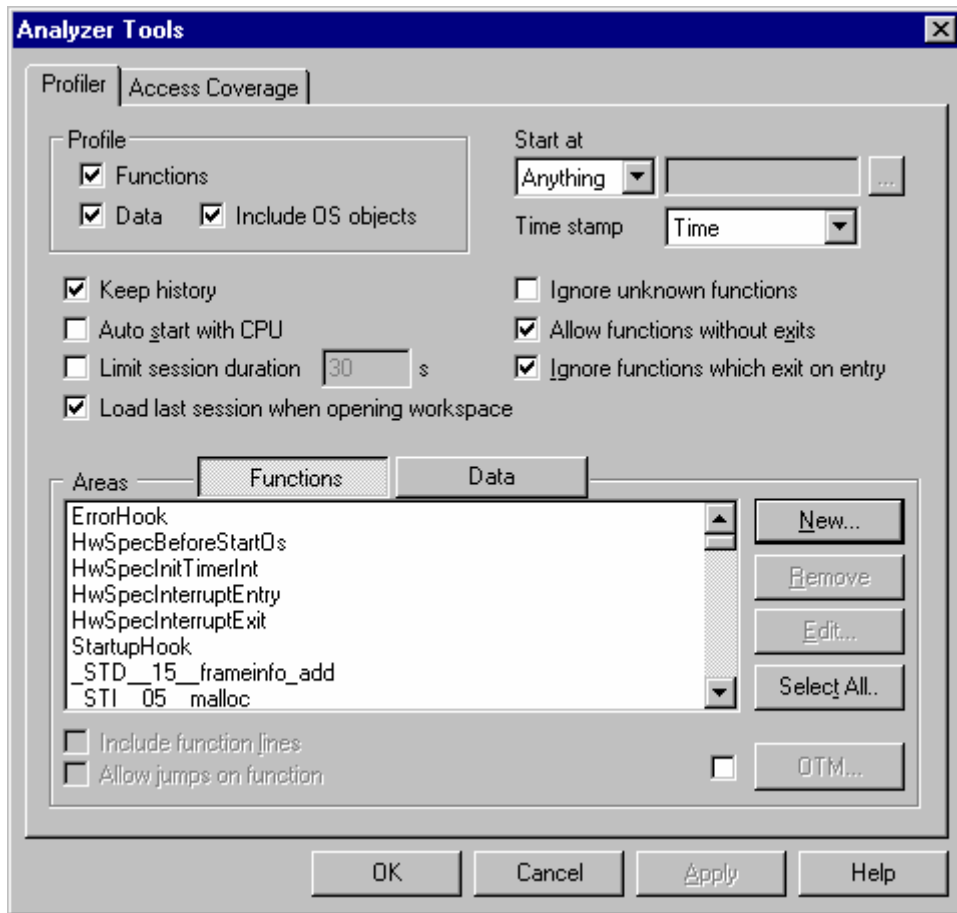


OSEK operating system selected in the 'Debug/Operating System' dialog

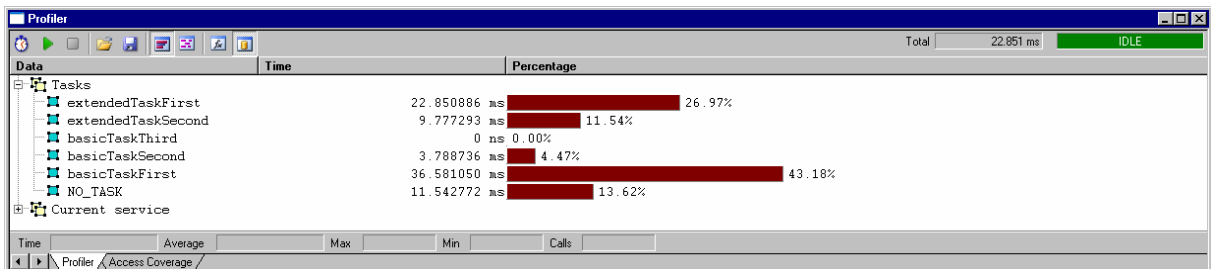


OSEK specific information setup in the 'OS Setup' dialog

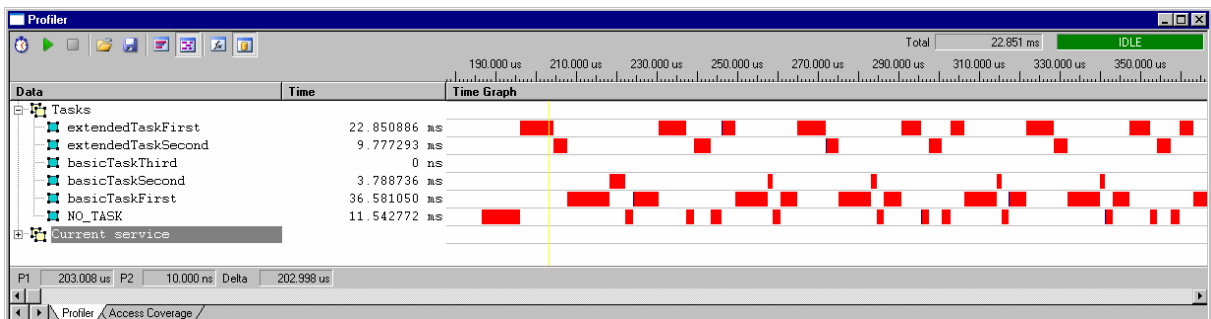
Functions profiler is configured in the same way like in the previous example. Below picture depicts final profiler configuration.



- Press 'Show Data' toolbar for task profiler information.

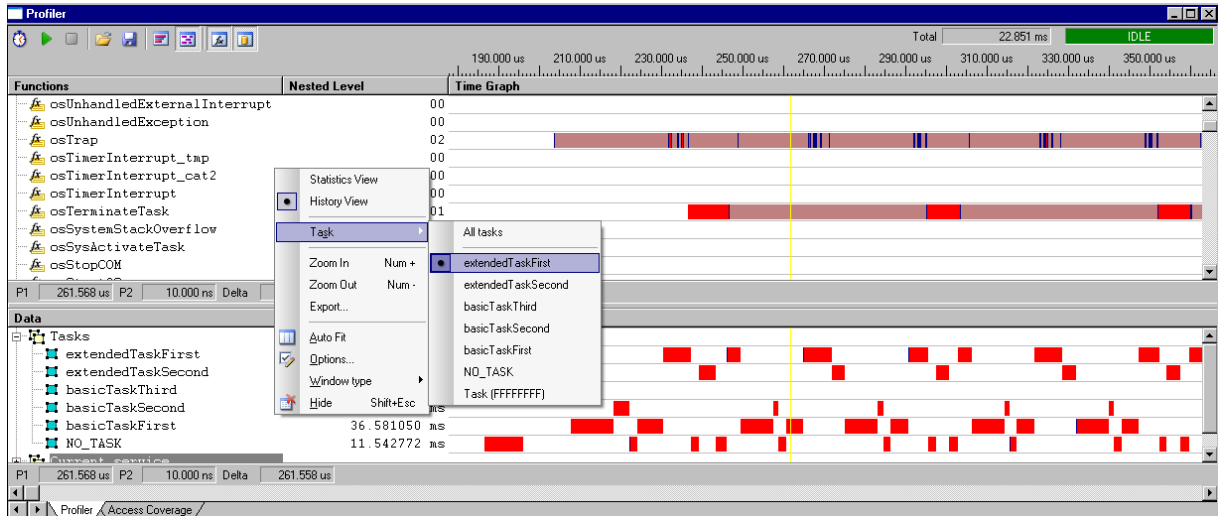


Task profiler statistics view



Task profiler history view

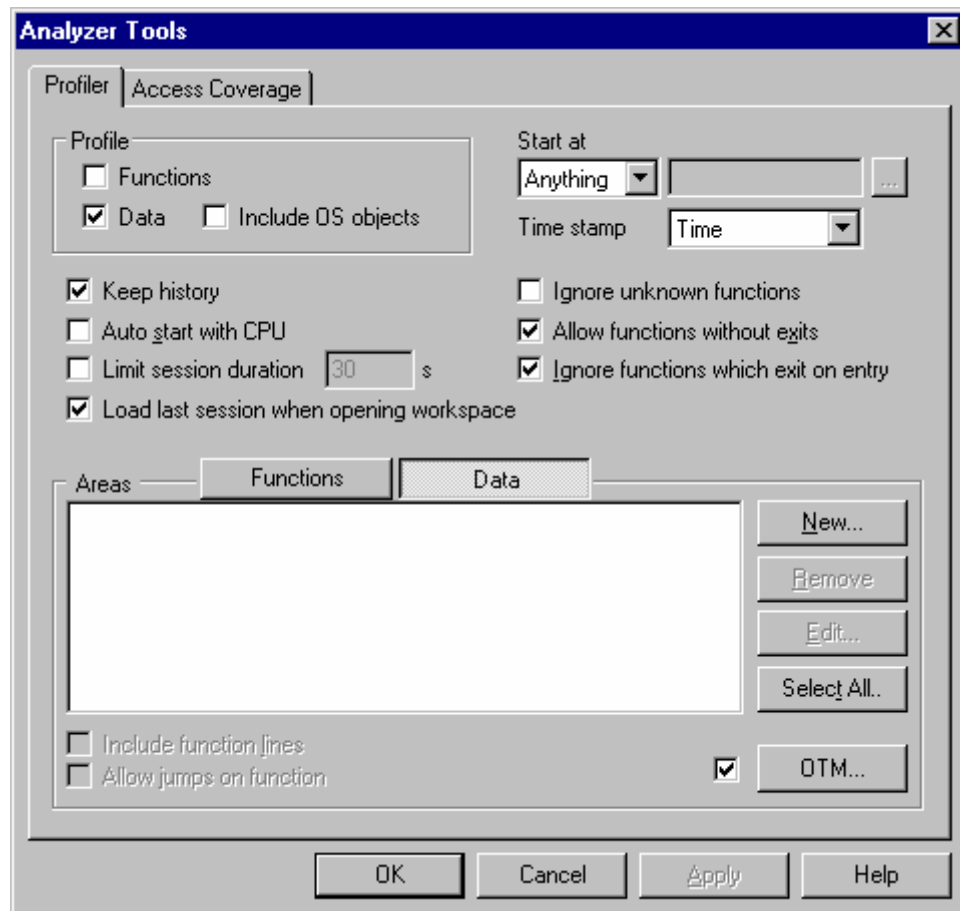
Functions profiler results (press 'Show Code' toolbar) can be displayed for all tasks or just a specific one (selectable from the local menu). Below picture shows functions profiler results for the specific task named extendedTaskFirst in this particular example.



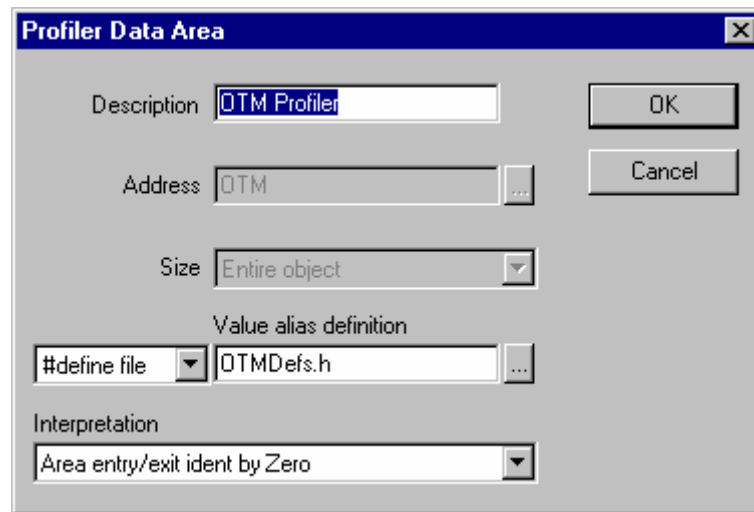
4.3 Example 3

Data profiler will be configured to obtain functions profiler results on Freescale MPC56x development system, which features an ownership trace (OTM).

- Open the 'Analyzer Tools' dialog by pressing the 'Analyzer Tools' toolbar located in the Profiler window.



- Check 'Data' option in the Profile field
- Check the check box beside the 'OTM...' button and press the 'OTM...' button to open the Profiler Data Area dialog.



- Name a data object.
- Select '#define file' option from the combo box and specify the define file in the 'Value alias definition' field. It's assumed that define file is written prior to this step.

Per default, the values that the data object assumes will be shown verbatim in the profiler window. If the definitions for these values are provided in the define file, the more descriptive form is displayed in the profiler window. In this example, OTMDefs.h describes possible OTR values. The excerpt from the OTMDefs.h:

```
#ifndef __OTMDefs_h__
#define __OTMDefs_h__

#define fn_main          1
#define fn_mainExit     0

#define fn_Type_Simple   2
#define fn_Type_SimpleExit 0

#define fn_Type_Arrays   3
#define fn_Type_ArraysExit 0

#define fn_Type_Pointers 4
#define fn_Type_PointersExit 0

...

#endif
```

- A data object serves as an indicator of function/service entries and exits. Select 'Area entry/exit ident by Zero' option in the 'Interpretation' field. In this case, the data object value 0 indicates that the function has exited. All profiled functions should write zero to the data object (OTR) on exit. The profiler assumes which function exited on recorded 0 by examining previously recorded entry point.
- Close the 'Profiler Data Area' dialog. The profiler is now configured.

In final step, the source code is modified. For all function entry and exit points, a write to the OTR is added. OTR, used later in the source code, is defined in the following way:

```
#define OTR *(volatile unsigned long *)0x38002C
```

Below is an excerpt of the modified code from this example.

```
void main()
{
    OTR= fn_main;
    ...
    Type_Simple();
    ...
    Type_Arrays ();
    Type_Pointers ();
    ...
    OTR= fn_mainExit;
}

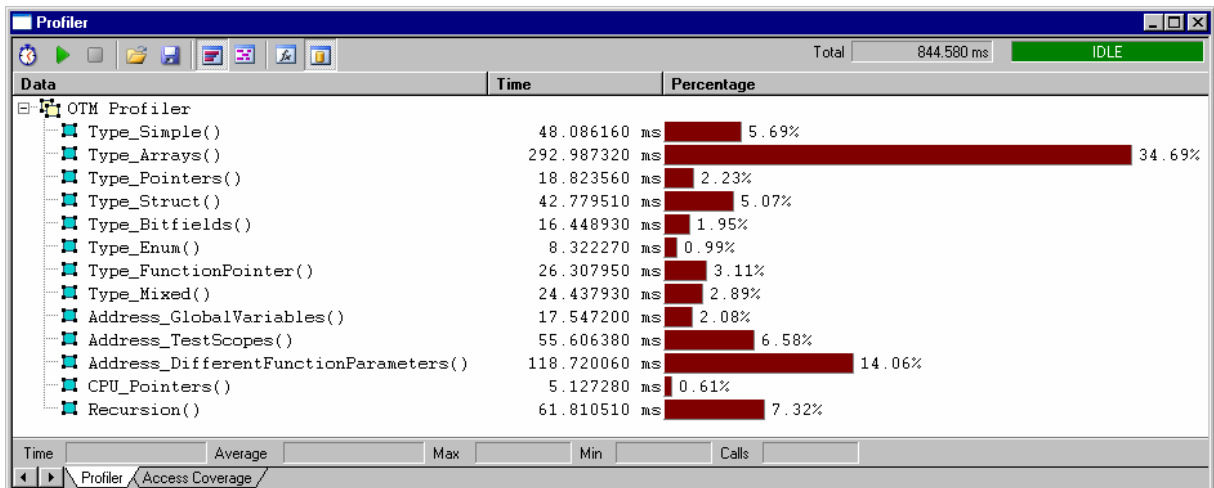
void Type_Simple()
{
    OTR= fn_Type_Simple;
    ...
    ...
    OTR= fn_Type_SimpleExit;
}

void Type_Arrays ()
{
    OTR= fn_Type_Arrays;
    ...
    ...
    OTR= fn_Type_ArraysExit;
}

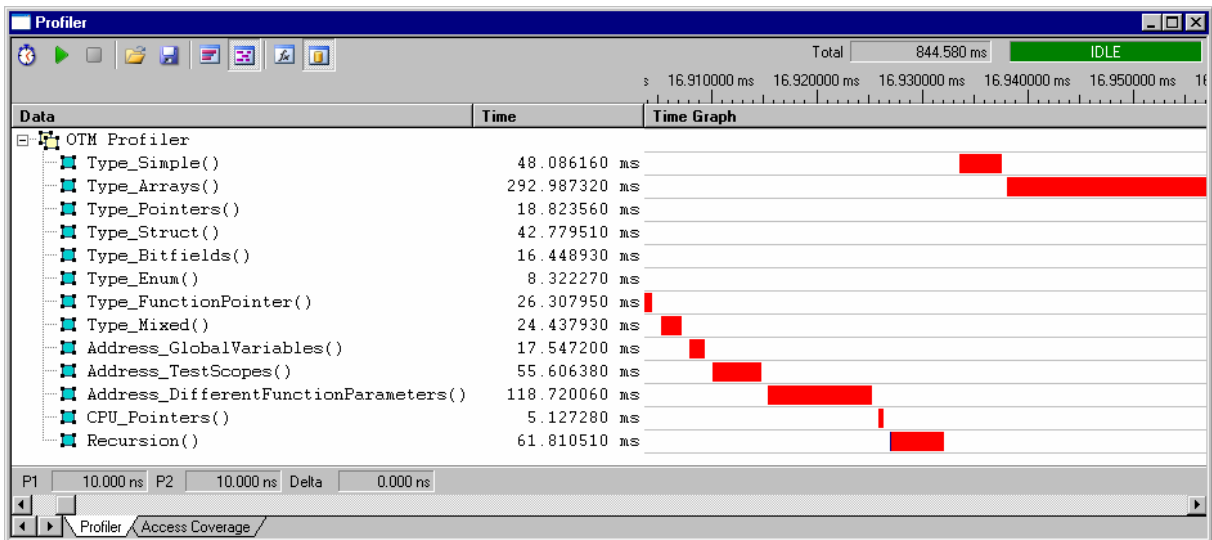
void Type_Pointers ()
{
    OTR= fn_Type_Pointers;
    ...
    ...
    OTR= fn_Type_PointersExit;
}
```

Only functions, which need to be profiled, have to be code armed. Finally, the profiler can be run.

- Press 'Show Data' toolbar for OTM profiler results.

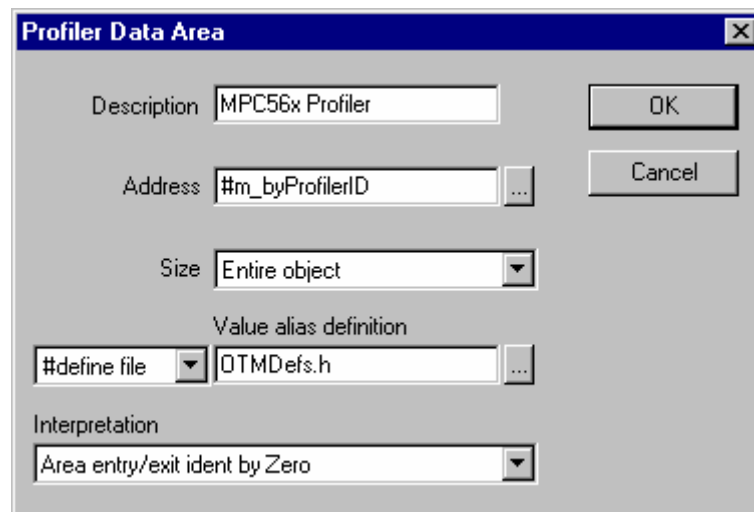


Statistics view



History view

Let's see the differences if the OTR is replaced by a global variable. Instead of pressing 'OTM...' button in the 'Analyzer Tools' dialog, press first the 'Data' button to select the data areas and then press 'New...' button to open Profiler Data Area dialog. Configure a new data object, which is a byte variable called `m_byProfilerID`. Next picture depicts the settings, which yields profiling `m_byProfilerID` variable.



Profiler Data Area dialog

All writes to the OTR in our source code should be now replaced with writes to the `m_byProfilerID`. Below is an example of main function writing to the global variable (e.g. `m_byProfilerID`) instead to the OTR.

```
void main()
{
    byProfilerID=otrMainEntry;
    ...

    ...
    Type_Simple();
    ...
    Type_Arrays ();
    Type_Pointers ();
    ...
}
```

```

    byProfilerID=otrMainExit;
}

```

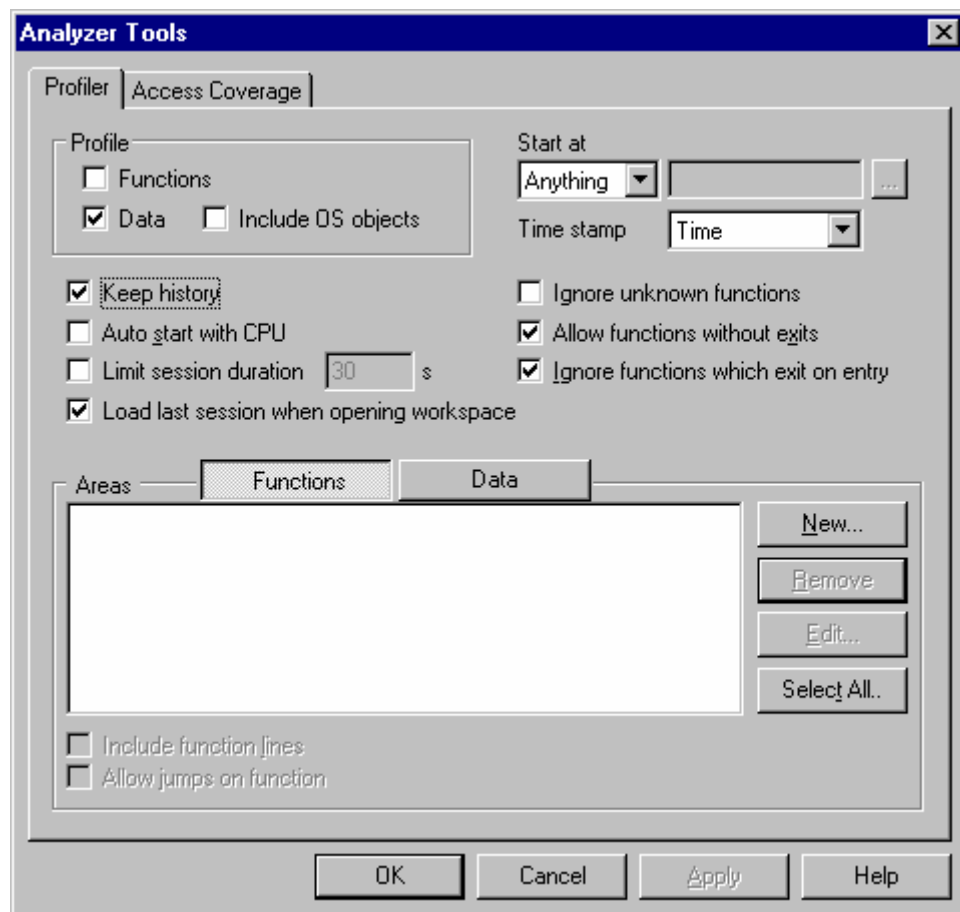
Now the profiler can be run and the results analyzed as before.

4.4 Example 4

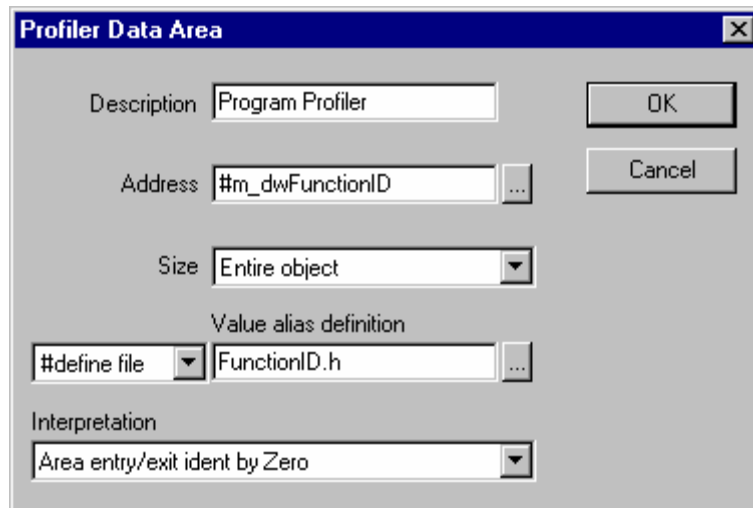
This example is very similar to the example 3 since the same concept is used but on an in-circuit emulator. Typically, an in-circuit emulator features functions profiler by default. Still, in this example, the data profiler is configured to obtain the functions profiler results on NEC V850ES/Fx3 ActivePRO development system.

This same approach can also be used to profile custom events in the application. For instance, the user first identifies custom events (e.g. check points) in the program and then equips those with extra code writing to a global variable. The data profiler can then profile that global variable and the user gets insight into the events' behaviour respectively application behaviour.

- Open the 'Analyzer Tools' dialog by pressing the 'Analyzer Tools' toolbar located in the profiler window.
- Check the 'Data' option in the Profile field.



- Press the 'Data' button to define the data areas. Next, press the 'New...' button which opens the 'Profiler Data Area' dialog.



- Put a name (e.g. Program Profiler) in the Description field and specify the address of the variable to which the code armed application will write on function entry and exit (e.g. dwFunctionID).
- Select '#define file' option from the combo box and specify the define file in the 'Value alias definition' field. It's assumed that the define file is written prior to this step.

Per default, values that the data object assumes will be shown verbatim in the profiler window. If the definitions for these values are provided in the define file, the more descriptive form is displayed in the profiler window. In this example, FunctionID.h describes possible m_dwFunctionID global variable values. Below an excerpt from the FunctionID.h:

```
#define fn_Address_DifferentFunctionParameters 1
#define fn_Address_GlobalVariables 2
#define fn_Address_TestScopes 3
#define fn_Func1 8
#define fn_Func2 9
#define fn_Func3 10
#define fn_Type_Mixed 21
#define fn_Type_Pointers 22
#define fn_Type_Simple 23
#define fn_Type_Struct 24
#define fn_main 39
```

It is recommended that the application is modified in a way, that the added code is compiled and linked into the application conditionally. That means that the original unmodified application can still be easily tested too by simply commenting for instance one #define. Let's assume that we define Profiler.h header file, which is then included in the main.c. The excerpt from the Profiler.h:

```
#ifndef __Profiler_h__
#define __Profiler_h__

#define PROFILER_ENABLE
#define PROFILER_TYPE long

#ifdef PROFILER_ENABLE
#include "ProfilerFuncs.h"
extern PROFILER_TYPE g_ProfilerID;

#define PROFILER_ENTRY(FN_NAME) g_ProfilerID=FN_NAME;
#define PROFILER_EXIT g_ProfilerID=0;

#else /* PROFILER_ENABLE */
#define PROFILER_ENTRY(FN_NAME)
#define PROFILER_EXIT

#endif /* PROFILER_ENABLE */
#endif /* __Profiler_h__ */
```

Now, as long as the application needs to be code armed for debugging needs, set

```
#define PROFILER_ENABLE
```

in the Profiler.h. When performing the final application test or when functions profiler via data profiler is no longer required, simply comment the define

```
// #define PROFILER_ENABLE
```

and recompile the project, which results in the original application code without code arming.

- A data object dwFunctionID serves as an indicator of function/service entries and exits. Select 'Area entry/exit ident by Zero' option in the 'Interpretation' field. In this case, the data object value 0 indicates that the function has exited. All profiled functions should write zero to the dwFunctionID data object on exit. The debugger can assume which function exited on recorded 0 by examining previously recorded entry point. Note that this concept works always as long as all functions writing 0 on exit are profiled.
- Close the Profiler Data Area dialog. The profiler is now configured.
- In final step, the original source code must be code armed. For all profiled functions, entry and exit points must write to the m_dwFunctionID global variable. Below is an example of the code armed source:

```
void main()
{
    PROFILER_ENTRY(fn_main);
    ...
    Type_Simple();
    ...
    Type_Arrays ();
    Type_Pointers ();
    ...
    PROFILER_EXIT
}

void Type_Simple()
{
    PROFILER_ENTRY(fn_Type_Simple);
    ...
    ...
    PROFILER_EXIT
}

void Type_Arrays ()
{
    PROFILER_ENTRY(fn_Type_Arrays);
    ...
    ...
    PROFILER_EXIT
}

void Type_Pointers ()
{
    PROFILER_ENTRY(fn_Type_Pointers);
    ...
    ...
    PROFILER_EXIT
}
```

Note that only functions, which are profiled, have to be code armed. Finally, run the profiler and inspect the results.

Disclaimer: iSYSTEM assumes no responsibility for any errors which may appear in this document, reserves the right to change devices or specifications detailed herein at any time without notice, and does not make any commitment to update the information herein.

© iSYSTEM. All rights reserved.