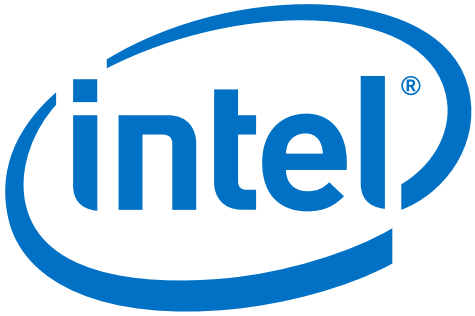

Movidius™

Movidius Profiler

Version v1.18.8.1 / 2018-07-26

Table of Contents

1. Introduction **2**

2. Prerequisites **3**

3. Profiling Types **4**

3.1. Function Profiler 5

3.1.1. Integration 6

3.1.2. Configuration 7

3.1.3. Running the application 9

3.1.4. Interpreting the results 10

3.2. Sample Profiler 11

3.2.1. Usage 12

3.2.2. Operation 14

3.2.3. VTune Amplifier 15

Import 15

Hardware events viewpoint - tabs 15

Hardware events viewpoint - tabs 15

3.3. Trace Profiler 16

3.3.1. Configuration 16

3.3.2. Trace Profiler API 17

Printf like strings 17

Bulk Data 17

Events 18

3.3.3. Sinks 19

3.3.4. FAQ 20

Copyright and Proprietary Information Notice

Movidius Ltd.
1730 South El Camino Real, Suite 200
San Mateo, CA 94402
http://www.movidius.com/

1. Introduction

> *The best preparation for tomorrow is doing your best today.*
>
> — H. Jackson Brown, Jr.

Summary

In order to detect and fix performance issues or inefficient resource utilization an application can be verified using more or less invasive tools. The Movidius/Intel Profiling Suite offers several options in order to achieve this goal.

- Sample Profiling - non invasive method to obtain a statistical view of the execution. This can provide a good overview in most cases even if it's not very accurate. Should be the first instrument to use.
- Function Profiling - instruments the code to monitor function calls. Highly invasive by default but can be very accurate if properly configured
- Trace Profiling - Provides means for sending events with timestamp, and printf-like messages. Accurate and a bit invasive but difficult to use. It is automatically activated by Function and Sample Profilers in order to retrieve specific data.

If you need to optimize something, first measure with some tools, don't rely on experience, code review, etc. This is because the compilers evolve and can optimize your code from a release to another.

Profiler results can be visualized in Eclipse, VTune, Chrome tracing, perf output, as text output report files, gprof, GraphViz, etc.

An application can be profiled by setting the variable PROFILE_INFO either as a global environment variable, or as a make variable using the current build system. To profile using the NextGen build system, you need to run the `make nconfig`. Please see the NextGen build system configuration procedure for more details

Example of how to execute function profiling from the command line:

```
make run PROFILE_INFO=function
```

Please see the MoviEclipse documentation regarding profiling from Eclipse

NOTE	If your application is using dynamic loading with the current MDK build system, you will need to migrate your application to the new build system before you can profile it

2. Prerequisites

1. Install latest MovidiusTools package
    a. Movidius Eclipse
    b. MoviDebug2
    c. MoviDebugServer or MoviSim
2. Install the MDK
3. Make sure the profiled elf file has DWARF information regardless of the optimization level. By default this is enabled. If not certain, verify that the `-g` flag is passed to the compiler.
4. The following tools have to be installed on your Linux system or in your Cygwin installation in case of running on Windows:
    a. **make**
    b. **coreutils**: *realpath*, *readelf*, *nm*, *grep*, *cut* and *awk*
    c. **gprof** [optional]
    d. **perf** [optional]
    e. **VTune** [optional]
    f. **cygpath** (Cygwin only)

3. Profiling Types

The Movidius profiler can assist with following types of profiling:

Function profiling

For the function profiler, moviProf is computing the runtime clock cycles of instrumented functions and other relevant information, outputting the result in csv files and in LTTNG format (to be used by TraceCompas, which is integrated within moviEclipse). LTTNG profiling data can be viewed also from the shell, using babeltrace (although the level of detail is less than what can be seen with TraceCompass).

Sample profiling

The Sample profiler is not intrusive on the monitored cores, affects only the timing of monitoring core, but the result is not always 100% accurate. When using Sample Profiler, a timer interrupt is used to read the IP from all the other cores at fixed time intervals. The output is gprof/perf style results, which can be viewed either from moviEclipse, VTune, perf, or from other gprof-compliant viewers. Note that for sample profiling not all cores can be profiled at the same time since one Leon core is used to run the sampling interrupt.

Trace profiling

Trace profiler can be used alone, or in conjunction with any other profile type to send events programmatically. There are also a set of predefined events that are sent from MDK to the profile data stream when the trace profiling is activated.

3.1. Function Profiler

The Function Profiler is responsible for profiling the instrumented functions, applicable to both RTEMS and/or Bare Metal. It uses compiler options to insert custom hooks for function entry and exit, and RTEMS extensions for calling a thread switch hook each time a thread switch occurs.

By storing the exact timestamp (measured in clock cycles) at which a function was entered/exited and the exact timestamp at which a thread switch occured, we can create the call history of an application and profiling information such as inclusive, exclusive and total time spent in a function.

If some functions should not be instrumented, then those function should be decorated with `__attribute__((no_instrument_function))`. Alternatively you can use `FP_<CORE>_INSTRUMENT_ONLY` and `FP_<CORE>_EXCLUDE_FILE_LIST`/`FP_<CORE>_EXCLUDE_FUNCTION_LIST` variables to control what will be profiled and what not. See the Configuration chapter for more details

Inclusive time refers to the time spent in that function including the calls to other functions from within the function's body. However, this time does not include the time spent in other threads (in the case when the Leon is running RTEMS), or in interrupts (BareMetal case)

Exclusive time refers to the time spent in that function only, excluding the calls to other functions from within. Time spent in other threads is not included, similar to the case of inclusive time.

The time span of a profiling session is dependent on the size of the global buffer and the number of functions that are instrumented. You can filter out functions by a regular expression of a symbol name, and/or specify what source files to be instrumented. By default all functions compiled from all source files are instrumented.

NOTE	Precompiled libraries are not instrumented, you need to recompile the code in order to insert the hooks.

In order to use Function Profiling, either use MoviEclipse (see documentation), or run from the command line

```
make run PROFILE_INFO=function
```

In NextGen build system, select "Profiling" from the kconfig menu, check "Use Function Profiler" and run normally.

Depending on the processor for which profiling is needed, `FP_CORE_PROFILE` can be used to set core specific options. This will be explained later in this document.

3.1.1. Integration

IMPORTANT	The timer block TIM0 that is used for profiling has to be enabled and clocked for the entire duration of the profiling execution. Also, the application must ensure that the bus infrastructure accessing the used Timers block is always powered and clocked.

If RTEMS is used, then the maximum number of user extensions must be increased by one. Increment the value of **CONFIGURE_MAXIMUM_USER_EXTENSIONS** define, or define it as in the example below:

```
#define CONFIGURE_MAXIMUM_USER_EXTENSIONS    1
```

All the profiling information gathering is based on sampling the free running counter each time a statistic is gathered. The function profiling is using the free running counter from LeonOS Timers block by default. This can be specified otherwise, by setting the `FP_<CORE>_TIMER` variable to the base address of the LeonRT Timers block, where `CORE` can be one of LOS, LRT, or SHV.

In the NextGen buildsystem, you can set the timers in the **Profiling › Function profiler configuration › Timers**. The entries are "LOS Timer", "LRT Timer", and "Shave Timer" with TIM0_BASE_ADR as default value.

If the profiling buffers are saved in the DDR (default configuration), then the application must make sure that the DDR is powered and clocked. If the application needs to deactivate the DDR then the profile buffers should be defined in CMX, using the `FP_<CORE>_DATA_SECTION` variable.

In NextGen the buffers can be placed in another section using **Profiling › Function profiler configuration › Data Section**

3.1.2. Configuration

Function Profiling can be configured using make variables passed as command line arguments, defined in the project's Makefile, or by using the Eclipse profile plugin provided with moviEclipse. Except `PROFILE_INFO` all other parameters have optional predefined defaults.

Table 1. Function Profiler Configuration Variables

Name	Default Value	Description
`PROFILE_INFO`	function	This flag instructs the buildsystem to use the function profiling. This is the only mandatory variable
`FP_CORE_PROFILE`	los lrt shave	The cores where the profiling will be activated
`PROFILER_DURATION_MS`	60000	After 60 seconds, the program will be stopped if did not exited successfully by then. -1 means infinite and is set to this value usually when `make debug` is executed
`FP_LOS_BUFFER_SIZE`	4194304	The buffer size. 4Mb is normally enough for any application, you could adjust this value by your need. More memory reduces the risk for data loss in circular buffer, but more time is needed to transfer the data. Also, for complex applications you might not have enough memory in DDR.
`FP_LRT_BUFFER_SIZE`	4194304	
`FP_SHV_BUFFER_SIZE`	1048576	Default is 1Mb for shaves. Total used memory for the shaves is this value multiplied by number of shaves (12 for Myriad2)
`FP_LOS_DATA_SECTION`	.ddr_direct.bss	Use uncached in order not to influence the caches and bss to fast load the elf
`FP_LRT_DATA_SECTION`		
`FP_SHV_DATA_SECTION`		
`FP_LOS_TIMER`	`TIM0_BASE_ADR`	Free running counter used by hook function. Make sure is initialized in code. The same counter is recommended to be used for all cores to synchronize the logs
`FP_LRT_TIMER`		
`FP_SHV_TIMER`		
`PROFILER_OUTPUT_DIR`	output/profile	This is used in order to differentiate the normal built elf and the profiled one since performance penalty can occur and the profiled ELF should not be mistaken for a production elf

Name	Default Value	Description
FP_INSTRUMENT_ONLY	*<unset>*	This is a list of object files. If provided, only the provided object files will have instrumented functions.
FP_COLLECT_STALLS	*<unset>*	Set this to true to collect SHAVE stalls and a more accurate execution instruction count. Requires more memory and introduces overhead
FP_<CORE>_EXCLUDE_FILE_LIST	*<unset>*	Comma separated files to be excluded from instrumentation. Works for Leon only.
FP_<CORE>_EXCLUDE_FUNCTION_LIST	*<unset>*	Comma separated values of demangled symbols
PRF_SYM_FILE	*<unset>*	This file will be used instead of the .elf file in order to read symbols. Useful for dynamic loading projects

NOTE

Please note that there is a single variable FP_SHV_BUFFER_SIZE specified for all Shaves, but in reality the build system will create one variable for each Shave (with a specific prefix). All the Shave buffers will have the size of FP_SHV_BUFFER_SIZE, so keep in mind that the total of DDR memory used for Shave buffers is FP_SHV_BUFFER_SIZE multiplied by the number of Shaves used by the application. It is not possible to configure the individual buffer sizes for the Shaves at this time.

In NextGen all these attributes are explained in the menu help for each item.

CAUTION

There might be other variables inside the profiler.mk that are not described in this table, but they may be subject to change without notice, so one should not rely on any other variable outside the ones described in the above table for fine tuning the mechanics of the function profiler component. The behavior of those other variables is not guaranteed to stay the same in the future.

3.1.3. Running the application

To run the application with function profiling enabled use the following make targets with the `PROFILE_INFO` variable defined as follows:

```
make run PROFILE_INFO=function
```

or

```
make debug PROFILE_INFO=function
```

Make will launch moviDebug2 with the application's default startup script as input. When the application exits or the timeout expires, the profile buffers are saved to /tmp/*.bin files indexed in a single .json file. After that, `moviProf` is invoked to decode the data from all buffers, creating csv, lttng, and all the other output files.

NOTE

If needed, you can edit the json file in order to change some configuration values and run again the profile decoder

```
moviProf /tmp/<file>.json
```

Optionally the -o flag can specify a different output folder. See all the arguments with `moviProf --help`

The TraceCompass viewer in moviEclipse is automatically updated with call-stack view and different statistics, once the user double-clicks the trace that becomes available within the project view.

In NextGen the init.tcl is taken from the Tools directory.

Please see the MoviEclipse documentation regarding running profiling

3.1.4. Interpreting the results

The results can be viewed in Eclipse, for more information see the help on "Help Contents" "Myriad Usage Guide" - "How to do profiling"

The csv files shows the code execution flow with timings, and can be viewed with any C/C++ code editor, works best with vim. It is recommended for all the cases when the result needs to be viewed/filtered/edited from command line using text only commands.

You can also see some chromium tracing output files, graphviz, etc.

For graphviz, you'll find in the output folder several *.gv files. You may combine several or simply view with `dot -Tpng output/graphviz_1.gv -o g.png && xdg-open g.png`

3.2. Sample Profiler

The sample profiler samples the Program Counter of running Shaves at configured time intervals. At the end of the execution, several .gprof files, LTTnG channels, and a perf output file are generated on disk (output/profile/)

The sampling interrupt can be set to trigger on any/both of the Leons (LRT and/or LOS) and is non-intrusive for SHAVE processors, as it does not require extra instrumentation. It can be run on RTEMS and/or baremetal environments. The Leons sample the SHAVE's instruction pointers and put them into memory buffers (each Leon has its own).

NOTE	There is a limitation for MX - the timer can't be set to values less than 1ms in the current implementation.

When the application finishes, the buffers are saved on disk, then passed to `moviProf` as input and the ".elf" file is parsed in order to determine the address space for each function from the Shaves. With this information, moviProf will create the gprof files and perf output including the time spent and the percentages out of total runtime for each function from each Shave. For a quick run-through, please try the sample project SampleProfilerExample from the MDK example/HowTo projects, to see the actual files that are produced.

The results can be viewed in Eclipse, from command line with gprof and perf, and with VTune importing the perf output file.

IMPORTANT	It is not possible to do sample profiling on the Leon cores with the current release of the sample profiler.

3.2.1. Usage

You can profile the application in current build system, or using the NextGen buildsystem (based on kconfig).

To run the application with function profiling enabled in legacy mode, use the following make targets with the `PROFILE_INFO` variable defined as follows:

```
make run PROFILE_INFO=sample
```

or

```
make debug PROFILE_INFO=sample
```

Table 2. Sample Profiler Configuration Variables

Name	Default Value	Description
`PROFILE_INFO`	sample	This flag instructs the buildsystem to enable sample profiling. This is the only mandatory variable
`SAMPLE_PERIOD_MICRO`	10	Instruction pointer sampling interval in microseconds. The default value should work for most cases; if the application - however - runs shaves for a long time, bigger values can be used for decreasing the used DDR size
`SAMPLE_DURATION_MS`	14000	Maximum total profiling duration, in milliseconds. This value should be configured to cover the total runtime of Shaves.
`SP_TIMER_INTERRUPT_LEVEL`	13	Timer interrupt level
`SAMPLE_PROFILER_DATA_SECTION`	`__attribute__ ((section (".ddr.bss")))`	Sample profiler data section name
PROFILER_OUTPUT_DIR	output/profile	This is used in order to differentiate the normal built elf and the profiled one since performance penalty can occur and not to be mistaken for a production elf
PROFILER_DURATION_MS	60000	After 60 seconds, the program will be stopped if did not exited successfully by then. -1 means infinite and is set to this value usually when `make debug` is executed

Name	Default Value	Description
PRF_SYM_FILE	*<unset>*	This file will be used instead of the .elf file in order to read symbols. Useful for dynamic loading projects

Update SAMPLE_PERIOD_MICRO and SAMPLE_DURATION_MS if absolutely necessary. These parameters will impact the DDR buffer size used by the profiler. The amount of used DDR space, in bytes, can be defined by the formula:

```
const int ddrBufferSize = (SAMPLE_DURATION_MS / SAMPLE_PERIOD_MICRO * 1000) * (ADDR_SIZE+1);
```

where ADDR_SIZE is 4 on current myriad platforms. The +1 value comes from the core ID that is saved in the buffer as well

Another method that sample profiler can be used, is by running the application in the simulator and enabling the core logs. This can be done by running moviSim with -enCoreLogs. This will create several Core_SHAVE*.out files on disk. From the folder where those moviSim output files are created, run <TOOLS_DIR>/linux64/bin/simProf.sh <path_to_elf> <args to moviProf>. This script is available only on Linux and parse the moviSim logs creating a moviProf input file for sample profiling.

3.2.2. Operation

The sample profiler registers to start/stop Shave events send by the Trace Profiler. This means that Trace profiling will be enabled automatically when Sample Profiler is active.

At start shave event a timer is started configured on an interrupt for Bare Metal. For RTEMS, a task is created in order to start the timer. In the timer interrupt, the instruction pointer of each running Shave is saved int the internal buffer along with the shave id, at the configured `SAMPLE_PERIOD_MICRO` interval.

The saved binary data is converted into gprof compatible files that can be viewed either in command line or moviEclipse.

The perf output can be viewed in VTune and/or in console too.

Example in console for perf:

```
perf report
```

3.2.3. VTune Amplifier

To get started, create a symbolic link to perf file called `data.perf`

```
ln -s perf.data data.perf
```

Import

Open VTune Amplifier and create a new project, in order to import `data.perf` file.

1. Select `New Project⋯` from Welcome page.
2. Enter a project name and change the default location if needed.

Traces can be imported now.

1. Select `Binary/Symbol Search` from the right.
2. Select `Source Search` if needed.
3. Use `Import a single file` an then `Browse⋯` for file `data.perf`.
4. Select `Import`.

Hardware events viewpoint - tabs

- Summary - analyze application-level event count;
- Event Count - analyze event count per grouping;
- Sample Count - analyze event sample count per grouping;
- Caller/Callee - explore event count for hot subtree;
- Top-down Tree - explore event count for functions and their callees;
- Platform - analyze CPU, GPU and bandwidth usage with platform metrics.

Hardware events viewpoint - tabs

- Summary - identify application-level hotspots;
- Bottom-up - analyze hotspots and their callers;
- Caller/Callee - analyze hot subtrees;
- Top-down Tree - explore event count for functions and their callees;
- Platform - analyze CPU, GPU and bandwidth usage with platform metrics.

NOTE

Traces can be grouped in more ways. Change the value for `Grouping` to achieve this.

For a cleaner view, hide inline functions. In order to do this, from the bottom bar, on right, change `Inline Mode`.

3.3. Trace Profiler

The trace profiler is used to manually instrument code and print data

3.3.1. Configuration

Table 3. Trace Profiler Configuration Variables

Name	Default Value	Description
TRACE_BUFFER_SIZE	1024	Trace buffer size in bytes
SINK_FUNCTION	_printf_clone	the function used to process printf-like strings
SINK_BULK	_bulk_hexdump	function dumping out bulk data
DEFAULT_LOG_LEVEL	LOG_LEVEL_INFO	Minimum log info. Everything below this value will be discarded. This is the initial value of the log level that can be changed at runtime
MAX_STATIC_LOG_LEVEL	LOG_LEVEL_TRACE	Everything below this value will be compiled out
MV_UNIT_NAME		Used to set the dynamic log level per unit basis. Define it before including this header.
MV_DBG_FMT_STR_SIZE	256	Max size for printf-like string messages. If a message is bigger, will be truncated
SYNC_LOG_MSG	*<undefined>*	If defined, will synchronize messages. Define this if you see scrambled messages
DBG_ARGx	*<undefined>*	(x=1..6) - prefix used in DBG_LOG()/DBG_ERROR()/...

The DBG_ARGx (x=1..6) is used to prefix messages with some informations. You can use predefined or custom one. The values are printf-like arguments, and several predefined values can be used

The number x represent the position in string. Gaps are allowed. Numbers out-of-range will be silently ignored

Predefined values:

- DBG_PRINT_FILE_LINE
- DBG_PRINT_MODULE_NAME
- DBG_PRINT_TIMESTAMP
- DBG_PRINT_LOG_LEVEL
- DBG_PRINT_THREAD
- DBG_PRINT_CORE_ID

Example:

```
// file.c
#define DBG_ARG4 DBG_PRINT_THREAD
#define DBG_ARG2 DBG_PRINT_TIMESTAMP
```

```
# gcc -DDBG_ARG6=DBG_PRINT_FILE_LINE
```

```
// file.c
#define DBG_ARG1 "Hello %s", "world!"
```

3.3.2. Trace Profiler API

It is defined in the logMsg.h

The API can send out three types of data: printf-like message strings, bulk data, and events.

It is a system of a macro functions expanded at build time, based on the configuration. Each function can have TRACE, DEBUG, INFO, WARNING, ERROR, or FATAL in it's name with the corresponding severity. This is used for filtering the logs. Also the entire tracing can be disabled with 0 overhead. The severity can be static and dynamic. For example one may choose to compile out all logs below INFO, and at runtime to set dynamically the logs from WARNING to FATAL.

Printf like strings

```
LOG_TRACE(fmt, ...);
LOG_DEBUG(fmt, ...);
LOG_INFO(fmt, ...);
LOG_WARNING(fmt, ...);
LOG_ERROR(fmt, ...);
LOG_FATAL(fmt, ...);
```

Example:

```
LOG_ERROR("Invalid handle %d\n", hInv);
```

Bulk Data

```
LOG_BULK_TRACE(data, size);
LOG_BULK_DEBUG(data, size);
LOG_BULK_INFO(data, size);
LOG_BULK_WARNING(data, size);
LOG_BULK_ERROR(data, size);
LOG_BULK_FATAL(data, size);
```

For the bulk data, a sink function and a decoder must be provided. For convenience the following sink is provided: _bulk_hexdump - print bytes in hex to stdout.

Example:

```
int main() {
     char byteArray[] = { 0xca, 0xfe, 0x1e, 0x1e, 0x01, 0x23, 0x45, 0x67, 0x89 };
     // send out of the chip 9 bytes of data from byteArray
     LOG_BULK_INFO(byteArray, sizeof(byteArray));
}
```

Events

Some events are preconfigured and can be used. The standard events are defined in dbgLogEvents.h and have id's starting from 1 to 9999 upwards. User defined events must have id's bigger than 9999 (`LOG_EVENT_LAST_EVENT`).

```
LOG_TRACE_EVENT(eventId, data);
LOG_DEBUG_EVENT(eventId, data);
LOG_INFO_EVENT(eventId, data);
LOG_WARNING_EVENT(eventId, data);
LOG_ERROR_EVENT(eventId, data);
LOG_FATAL_EVENT(eventId, data);
```

Example:

```
#include <logMsg.h>
#include <dbgLogEvents.h>
#define MY_SPECIAL_EVT (LOG_EVENT_LAST_EVENT+1)
//...

// send MY_SPECIAL_EVT with value 1
LOG_INFO_EVENT(MY_SPECIAL_EVT, 1);
```

If you want to create a special visualization, provide a json file having .mtrj extension in your project.

Please see the MoviEclipse documentation regarding this feature

```
{"eventTypes" : [
    {
        "id" : "10000",
        "filter" : "acceptEvent(trace, 10000);",
        "name" : "MY_SPECIAL_EVT",
        "class" : "MoviLogicEvent",
        "config" : {
            "bgcolor" : "0x00c4c61c"
        }
    }
    ]
}
```

3.3.3. Sinks

SINK_FUNCTION and SINK_BULK values are function symbols that must be found at link phase. The signature is the following:

```
/// msg - null terminated string to be sent out of the chip
void SINK_FUNCTION(const char* __restrict msg);

/// data - address to the binary data to be sent out
/// @size - data size
void SINK_BULK(void * __restrict data, size_t size);
```

Predefined values for SINK_FUNCTION: _printf_clone(default) and _trace_print (trace profiler based).
Predefined values for SINK_BULK: _bulk_hexdump(default)

Example how to use bulk sink with vcsHooks:

```
void _vcs_save_file(void * __restrict data, size_t size) {
      saveMemoryToFile((uint32_t)data, size, "buf.out");
}
#define SINK_BULK _vcs_save_file
#include <logMsg.h>

// ...
```

3.3.4. FAQ

Code compiled successfully with `PROFILE_INFO=<type>` *but I see no result. Where should I look?*

Run `make clean`. Sometimes it helps when make doesn't rebuild objects from previous project compilations

Code and/or data too big after function profiling the code and doesn't fit to memory

- First, reduce the buffer size `FP_<CORE>_BUFFER_SIZE=size` and/or enable profiling only on the core you need to analyze `FP_CORE_PROFILE=⋯`.
- Reduce number of profiled functions with FP_INSTRUMENT_ONLY, FP_<CORE>_EXCLUDE_FILE_LIST, FP_<CORE>_EXCLUDE_FUNCTION_LIST, see configuration section for more details. Also consider `__attribute__((no_instrument_function))`
- Try to move as much as possible code and data to DDR
- Other potential solutions would be to use sample profiling instead of function profiling, make sure you build in release mode, and remove code that is not under test therefore not needed in this scenario

Can I function profile a project that enables/disables DDR

- Yes. But make sure you take out the profiler buffers from DDR in other sections
  - e.g.: `FP_LOS_DATA_SECTION=.cmx.data`

What happens with the profiled application if I stop the timer block?

- Nothing. It will simply not work. It is required for the timer block to have clock all the time.

I have the message "unable to install thread switch extension hook" and no thread switch event is detected

- The profiler needs to add the thread switch extension in RTEMS. Please check the define `CONFIGURE_MAXIMUM_USER_EXTENSIONS`. It should be defined and at least equal to 1 or greater if you have your own extensions

I have link errors after `make clean all PROFILE_INFO=⋯` *but works without clean* `make all && make all PROFILE_INFO=⋯`

- The profiler sets the output directory to a different place for safety. Unfortunately some projects are using the ./output/ hardcoded in Makefile instead of $DirAppOutput. If this is the case, either fix the project's Makefile, or set `PROFILER_OUTPUT_DIR=output`

I need to disable/enable profiling

- Programmatically call `disableProfiler()` and/or `enableProfiler()`