D5.2

Micro-ROS benchmarks - Initial

Grant agreement no.	780785
Project acronym	OFERA (micro-ROS)
Project full title	Open Framework for Embedded Robot Applications
Deliverable number	D5.2
Deliverable name	Micro-ROS benchmarks - Initial
Date	April 2019
Dissemination level	public
Workpackage and task	5.2
Author	Tomasz Kołcon (PIAP)
Contributors	Mateusz Maciaś (PIAP), Alexandre Malki (PIAP), Tomasz Plaskota (PIAP)
Keywords	Microcontroller, NuttX, Benchmarking
Abstract	This deliverable summarizes preliminary works in Task 5.2: Benchmarking of whole stack.

Contents

1 Summary

This deliverable summarizes the works in Task 5.2: Benchmarking of whole stack. It is initial level of benchmarking that will be continued in Task 5.3, Benchmark tooling for application developers.

This document starts with explanations of whole stack benchmarking procedures. This is followed by descriptions of used hardware and software tools. Next, benchmarking setup will be presented. Subsequently, results are presented for each benchmark along with comments. At the end, conclusions are presented.

Term	Definition
ROS	Robot Operating System
MCB	Multi-connectors board
SWO	Single Wire Output
SWD	Serial Wire Debug
SWV	Serial Wire Viewer
JTAG	Joint Test Action Group (also name of interface)
ITM	Instrumentation (or Instruction) Trace Macrocell
ETB	Embedded Trace Buffer
MTB	Micro Trace Buffer
DWT	Debug, Watchpoint and Trace
TPIU	Trace Port Interface Unit
OS	Operating System
RTOS	Real Time Operating System
HW	HardWare
IP	Internet Protocol
TCP	Transmission Control Protocol
RAM	Random Access Memory
6LoWPAN	IPv6 over Low-Power Wireless Personal Area Networks.
I/O	Input/Output
ETM	Embedded Trace Macrocell

2 Methodology

The work is an extension of done work described in D5.1: Reference values for HW + OS and shows results of platform baseline benchmarking based on defined use cases. Now, information about memory usage and the program execution parameters, like time performance, will be provided.

2.1 Memory analysis

The different kind of pertinent analysis of used memory:

- heap leaks tracking
- heap accesses

- heap allocation sources
- heap fragmentation
- stack usage

2.1.1 Heap leaks tracking

- Memory leak tracking
  - checking where memory block are not freed
  - it will provide information about memory block that are definitely lost
  - the total amount of allocated
  - where the leaking blocks are allocated

2.1.2 Heap accesses

- How is a block used
  - how many time a byte is read/written in average
  - average age of a block
  - how many blocks were alive in the same time (maximum)
- Interpretations
  - it would be possible to know if memory blocks are use efficiently. Thus, showing information about memory holes/alignment issues and thus fragmentation statistics.
  - if the number of allocation/de-allocation is efficient

2.1.3 Heap allocation sources

- Heap allocation sources
  - showing which function (line) in the code is doing the most allocations
  - showing the back-trace to track the function hierarchy
- Interpretations
  - this information shall be combined and analyzed in conjunction with the heap access. This can provide a way to enhance the execution performance. It also helps to understand which layer of the application is doing allocations

2.1.4 Heap fragmentation

- The heap fragmentation could be an issue as it might increase the heap unnecessarily. Hence, the information that could help to statistically qualify it, would be
  - the average/standard deviation block size when allocation is called
  - histogram to classify the number of block by size
- Interpretations
  - as the size of allocated blocks are varying the chance of fragmentation is greater

2.1.5 Stack usage

- Stack usage is an important information to know if it's corrupting the heap
  - this should never happen in x86_64 virtual memory. But can give use some insight about what is going on.

2.1.6 Memory usage analysis conclusion

The memory analysis shall be used for memory performance. It will provide indications to mitigates memory bottlenecks/issues. But it should not be forgotten that, on different architecture with different OS, the software behaviors will be different. Hence some other issues could arise on one platform/architecture and not in another.

2.2 Performance and communication benchmarks

2.2.1 Execution performances

The execution performances tool is intend to retrieve execution time information in order to know where is the CPU spending most of its time. In order to retrieve these information, the tools will rely on PC sampling methodology.

2.2.2 Communication benchmarking

Determine how much time key operations take to execute and characteristics of communication, such as throughput and latency. The simplest way is to write application like ping-pong.

2.3 Power consumption benchmarks

The current deliverable does not provide power consumption measurements. It will be performed in the next deliverable. However, some results are available in deliverable D5.1 .

3 Tools

3.1 Hardware tools

3.1.1 Multi-connectors board

Multi-connectors board (MCB) was made to avoid cable clutter and to facilitate test automation. MCB is used to interconnect software tools to benchmarked board like Olimex STM32-E407. Of course, this board can be replaced by other setup composed of inexpensive separate tools.

Board features:

- JTAG sniffing
- read SWO data stream via built-in FT232 UART<->USB adapter
- shunt resistors set for current measurement
- trigger signal output for oscilloscope
- connector for Saleae logic analyzer
- connector for Sigrok logic analyzer
- JTAG and SWD standard connectors
- easy configuration on jumpers

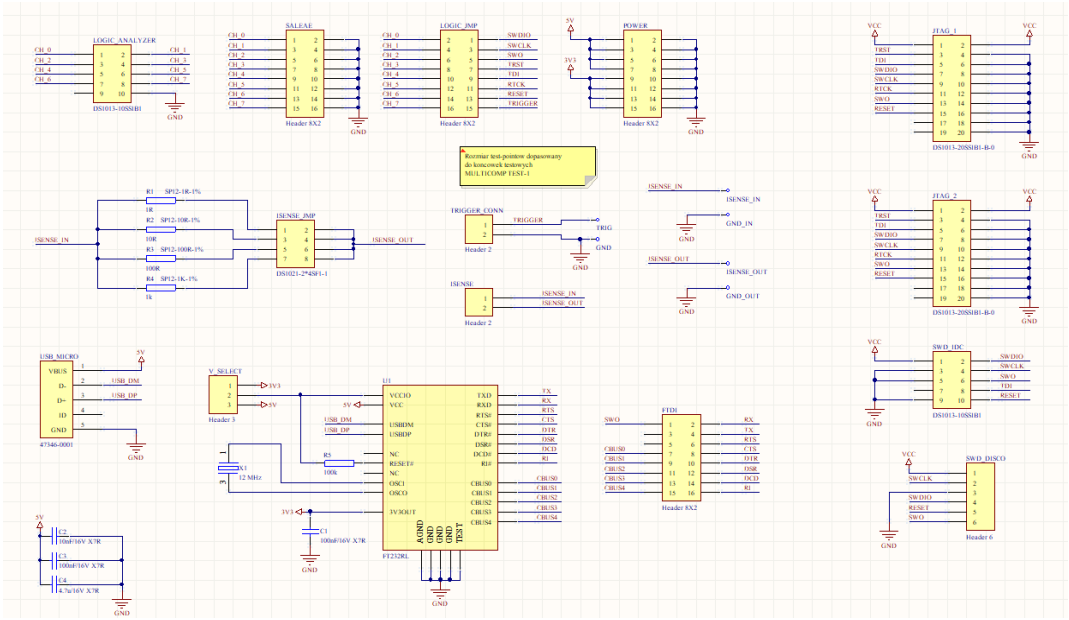


Figure 1: Multi-connectors board schematic

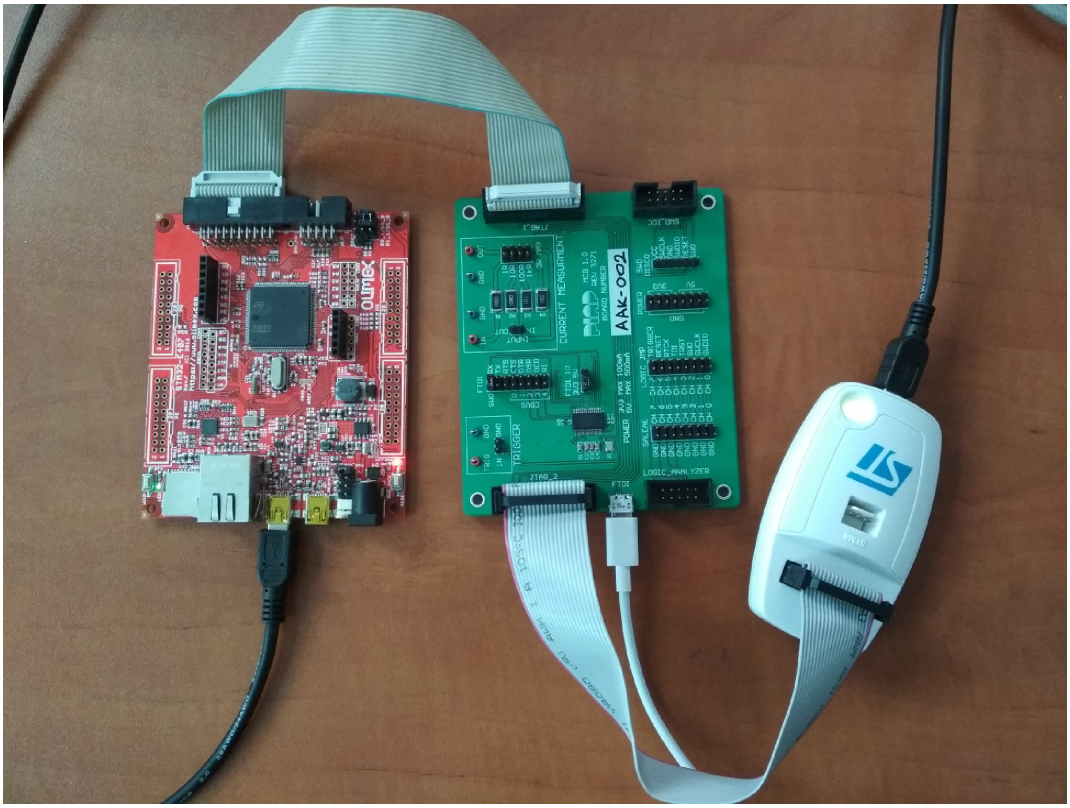


Figure 2: MCB (prototype) set with Olimex STM32-E407

MCB will be released as open-source project.

3.1.2 JTAG adapter

JTAG adapter is used to connect software benchmarking tools to MCB. We used ST-LINK debugger, but it can be any JTAG adapter supported by OpenOCD.

Figure 3: St-link adapter

3.2 Software tools

3.2.1 Benchmarking tool

The benchmarking tool is intended for embedded software benchmarking with low/none overhead intrusion. The tool (will) be able to benchmark:

- Performance execution using PC sampling a regular frequency (1/16384 CPU clock).
- Memory:
  - Static analysis: Information of the text + bss usage.
  - Stack analysis per thread: Retrieve per thread, the amount of stack.
  - Heap analysis per thread: Retrieve per thread, the amount of dynamic allocation.

The performance execution benchmarking is intended to work with any applications running on a micro-controller with the ARM-CoreSight debug system implemented.

However the *Stack* and *Heap* memory analysis will first target the nuttx *kernel*.

The tools are available on the microROS' github page https://github.com/microROS/benchmarking [1]

3.2.2 Tools description

Two tools were developed to address benchmarking on embedded platform with the minimum overhead in code. The solution is to use the debugging system, called CoreSight, available on all the ARM architecture. This solution offer a way to generically to access information regarding the processor status registers and to print trace messages.

The tools are currently able to provide CPU usage information and heap memory analysis. Those two tools will be discussed in depth.

3.2.3 How to get advantage of the CPU CoreSight and output it over a serial line

3.2.3.1 CoreSight The Debug System is the whole mechanism enabling the on-chip trace and debug facilities. More information about the Debug System can be found in the following document:

- ARM Debug Interface V5 [2].
- ARMV7-M Architecture Reference Manual Section C1.6 [3].
- CoreSight Design Kit revision r0p1 Technical Reference Manual [4].

The following picture shows data flow between different Trace Units (**ITM DWT ETM**) and elements of the Debug System (CoreSight):

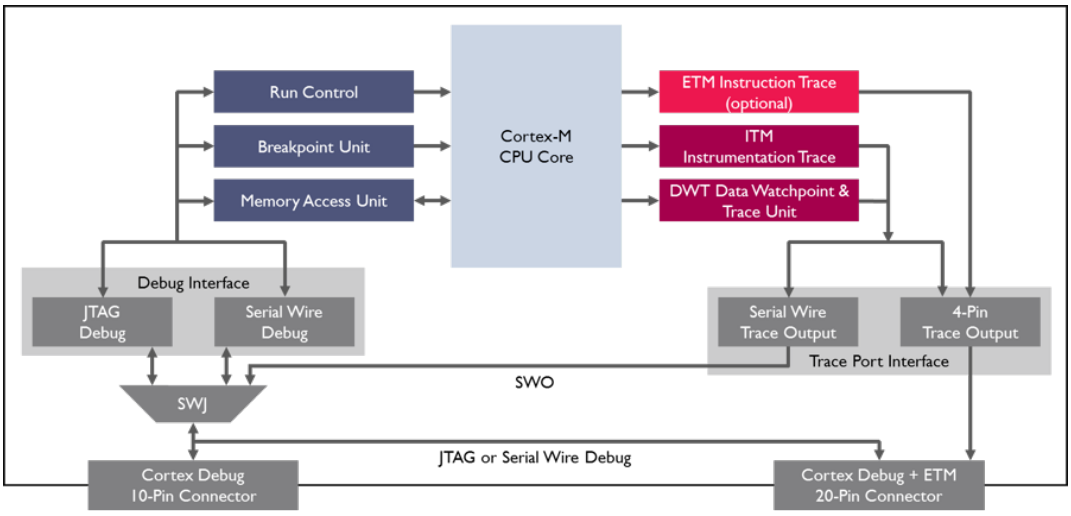


Figure 4: CoreSight - http://www2.keil.com/coresight/ [5]

As depicted by the above image, the **ETM** trace facility will not be used as it is optional, thus not available on all CPU.

3.2.3.2 Outputting over Serial line SWD is a low cost alternative to JTAG debugging. Indeed The Protocol only use 3 pins maximum **SWCLK/SWDIO/SWO**:

- **SWDIO** (Data In/Out).

- **SWCLK** (Clock) can be use to communicate with the processor.

The **SWDCLK** and **SWDIO** are the entry points of the host debugger to download and write code inside the processor flash memory. It can also be used to access the different memory bus to read/write them. This way, the host debugger can access all the memory mapped peripherals and RAM.

The **SWD** protocol is described in the **ARM Debug Interface (ADI) document section 4**. The details are not really interesting for our purpose, as this protocol is handled by the **SWD** Probe (In our case the STLINK V2) and by using **OpenOCD**. Depending on which platform is used, **SWD** pins will be different.

OpenOCD is at the heart of our tools. The bemchmarking tools instruct OpenOCD to program the CPU's CoreSight in a way that meets our ends.

3.2.3.3 Software Prerequisites The application was tested on Ubuntu 18.04.1 LTS. It shall work on other Linux distributions.

The list of packet needed for the application to work:

- autoconf
- autotools-dev
- binutils
- check
- doxygen
- gcc-arm-none-eabi
- git
- libtool
- libusb-1.0
- libftdi-dev
- make
- pkg-config
- texinfo

3.2.3.4 Hardware Prerequisites The benchmarking tool is using hardware debugger to reduce the overhead in code. Status about the different debugger/serial/board:

3.2.3.4.1 Debuggers List of debug probe supported:

Debugger Probe	Status
ST Link v2.1	Working

3.2.3.4.2 UART (SWO output) Every UART device that support the format 115200 8N1 should be working. MCB suports this.

3.2.3.4.3 Board List of boards:

Board	Status execution	Status memory
Olimex E407	Working	Working Heap (Stack and static not available yet)
STM32L152-Discovery	Not Tested	Not available

3.2.4 Compiling

To compile:

```
foo@bar:~$ ./autogen.sh # Will retrieve dependencies and compile them.
foo@bar:~$ ./configure
foo@bar:~$ make
```

The tool generates two executables: * pea (performance execution analysis) * mfa (memory footprint analysis) * msfa (memory stack footprint analysis **TBD**)

3.3 Memory footprint analysis

This tool will perform a memory analysis on an embedded platform.

3.3.1 Configuration file

Before execution the configuration file need to be changed/adapted depending on the use. The configuration file used by the application is located at **res/tests/memory_heap_config.ini**

More explanations about the fields are available in the template located at **res/configs/default_config.ini**

3.3.2 Memory footprint analysis (MFA)

The compiled application will be located at **apps/mfa**.

To execute it:

```
foo@bar:~$ ./apps/mfa
```

Before executing, the configuration file shall be filled appropriately and UART and SWD debugger shall be connected to the targeted embedded platform.

The memory footprint analysis tool currently focuses on analyzing the **HEAP** memory. The cornerstone stones of the tool are the ITM facility of the debug facility and the backtrace library located here:

https://github.com/red-rocket-computing/backtrace [6].

To achieve heap memory footprint analysis, Nuttx was patched. The patch consists in instrumenting the **malloc** in NuttX **nuttx/mm/mm_heap/mm_malloc.c**. Each call to **malloc** will output to the ITM facility a backtrace showing the hierarchy function call, the size of the allocations and the memory pointer. Here is an example:

```
alloc 152 0x20008F20 <--- Allocation of 152 bytes at address 0x20008F20
0x08001568 unknown
0x08000216 __start <--- Address calling from 0x080000216 +
function name __start
0x08002B2A os_start <--- Address calling from 0x080002B2A +
function name os_start
0x080050AE nxsig_initialize <--- Address calling from 0x080050AE +
function name
nxsig_initialize
0x08004FDE nxsig_alloc_pendingsignalblock <--- Address calling from 0x08004FDE +
function name nxsig_alloc_pendingsignalblock
0x08008B14 malloc <--- Address calling from 0x08008B14 +
function name malloc
```

Once the tools received the complete frame, it will use the calling address and find the corresponding symbols to the address provided. Finally, the MFA tool will output a JSON file as shown below:

```
{
    "0x0801024A nsh_parse_command": {
        "file": "/root/apps/nshlib/nsh_syscmds.c",
        "function": "cmd_uname",
        "line": 264,
        "count":    2,
        "allocated-blks":   [{
                "count":    2,
                "size": 24
            }],
      ....
    }
}
```

The previous json output shows that an allocation from the file located at **/root/apps/nshlib/nsh_syscmds.c** was made. The function cmd_uname made **2 allocations** of **24 bytes** at the line **264**.

The advantage of using ITM is to use a UART peripheral and leave it entirely to the application. Regarding the execution, it will add some overhead to the software. But this tool is only activated when the option is selected in the Nuttx configuration, as shown below:

From the main menu:

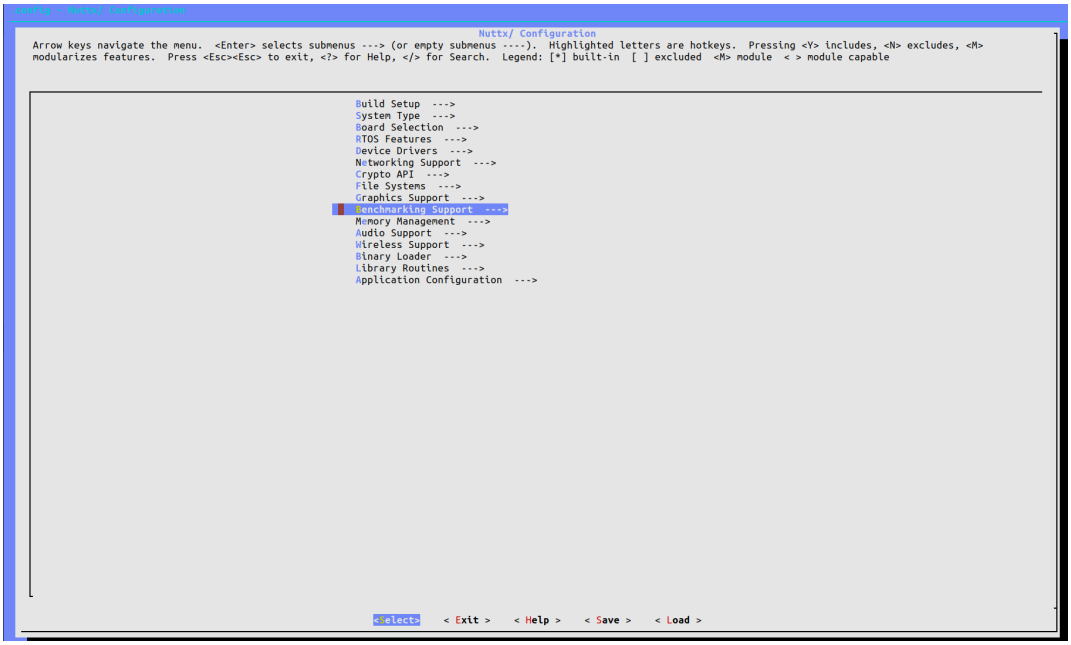


Figure 5: Menuconfig benchmarking main

In the Benchmarking menu:

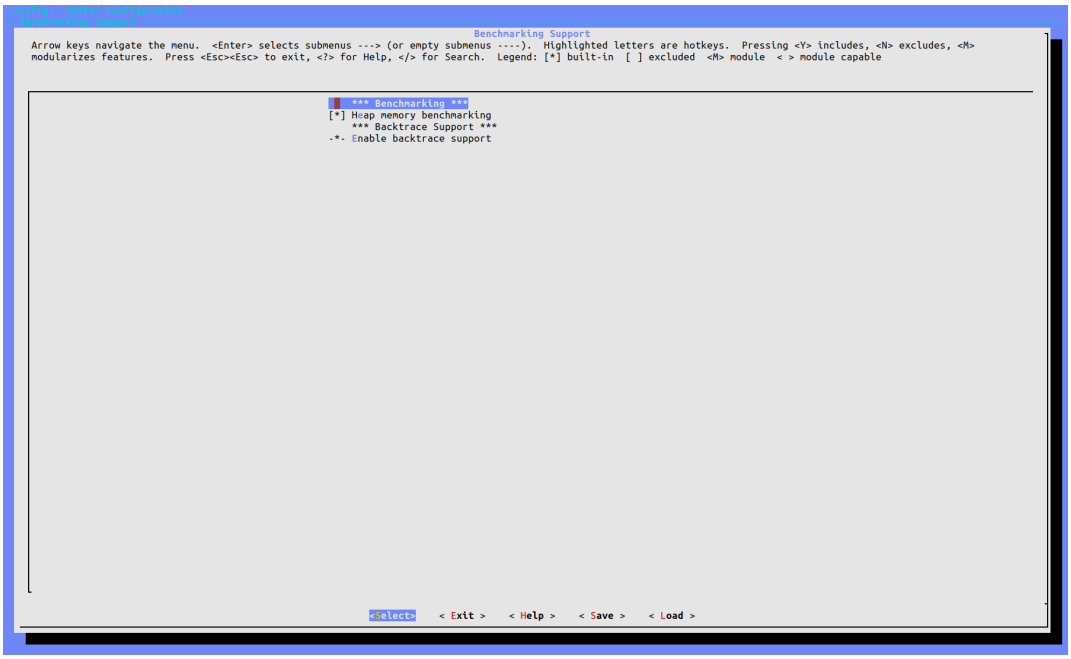


Figure 6: Menuconfig benchmarking bench

For our purpose, the results will be over-viewed and interpreted. Similar allocations and huge allocation (over 1 K bytes) will be listed and analyzed. Thanks to the tool, it is possible to know pieces of code responsible for allocations.

3.4 Performance execution analysis

This tool will perform new analysis execution analysis (CPU usage).

3.4.1 Configuration file

Before execution, the configuration file need to be changed/adapted depending on the use. The configuration file used by the application is located at **res/tests/execution_config.ini**

More explanations about the fields are available in the template located at **res/configs/default_config.ini**

3.4.2 Performance execution analysis (PEA)

The compiled application will be located at **apps/pea**.

To execute it:

```
foo@bar:~$ ./apps/pea
```

Before executing, the configuration file shall be filled appropriately and UART and SWD debugger shall be connected to the targeted embedded platform (e.g. using MCB). The PEA retrieves CPU usage information. The tool is using the CoreSight ARM IP. The PEA tool retrieves packets from Data Watchpoint and Trace (DWT). These packets are programmed by the PEA tool to embed the Program Counter (PC). The frequency at which these DWT packets are emitted is also programmed via the PEA tools.

Once the program is running, the DWT facility will emit a packet at the requested frequency with the PC to the SWO serial line. The tool decodes the packet received from the SWO serial line and sorts/stores the PC sample. Once the benchmarking the software is executed, it will produce a JSON file that will store the results. The results will provide number of "hits" corresponding to the number of times a function was called. In each function, the PC address is provided. The PC address correspond to the instruction within the function it belongs to. An example of the JSON output file:

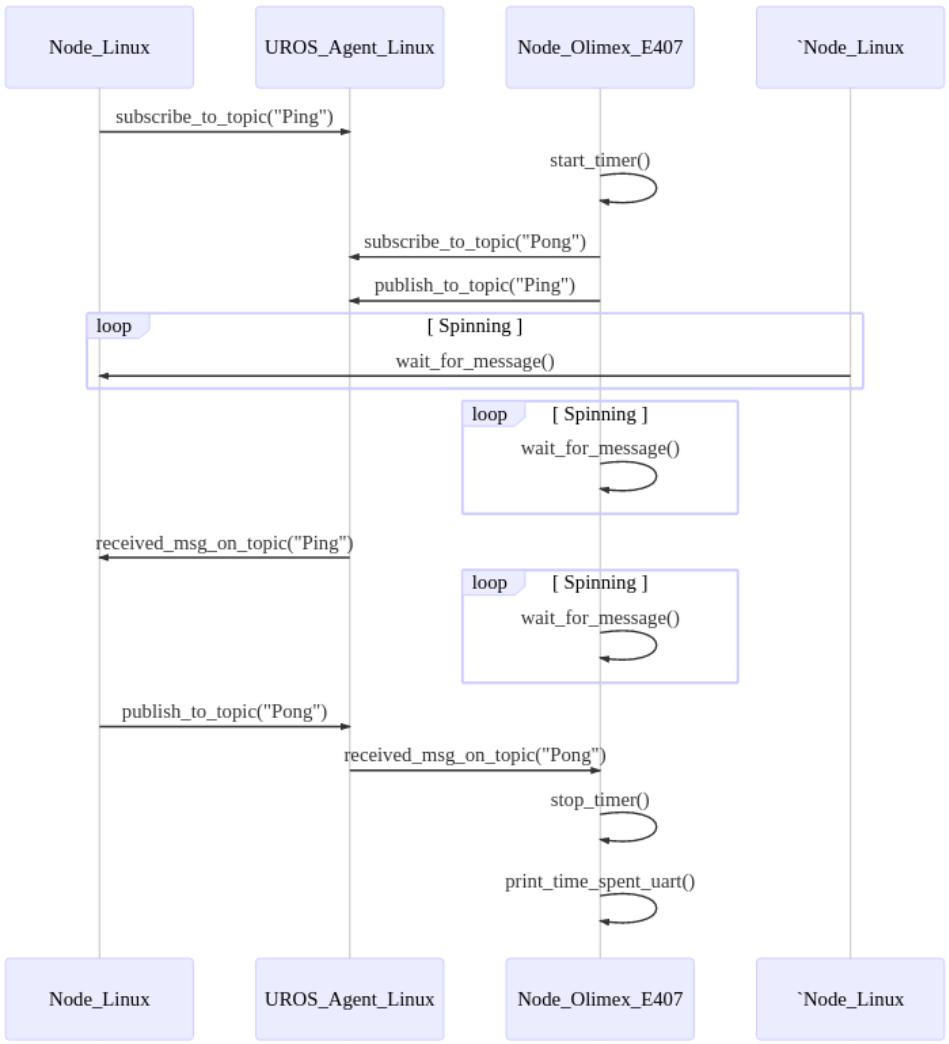
```
file: /root/nuttx/arch/arm/src/chip/stm32_start.c
function: __start
hits: 2 # Number of PC samples gotten within this function.
details: [
    { hits: 1, line: 298, address: 80001b6},
    { hits: 1, line: 296, address: 80001bc},

]
file: /root/nuttx/libs/libbacktrace/backtrace/backtrace.c
function: unwind_search_index
hits: 1
details: [
    { hits: 1, line: 42, address: 800202a},
```

```
]
file: /root/nuttx/libs/libc/string/lib_memset.c
function: memset
hits: 1
details: [
    { hits: 1, line: 183, address: 80085ec},

]
file: /root/nuttx/arch/arm/src/chip/stm32_idle.c
function: up_idle
hits: 691 # Number of PC samples gotten within this function.
details: [
    { hits: 135, line: 433, address: 80094c0},
    { hits: 70, line: 433, address: 80094c2},
    { hits: 65, line: 448, address: 80094c4},
    { hits: 142, line: 448, address: 80094c6},
    { hits: 139, line: 448, address: 80094c8},
    { hits: 140, line: 448, address: 80094ca},

]
file: /root/nuttx/sched/sched/sched_processtimer.c
function: sched_process_timer
hits: 1
details: [
    { hits: 1, line: 224, address: 804622e},

]
```

The above results show that most of the time was spent in the ***up_idle*** which is the idle task on NuttX. Indeed, the total number of hits for this session is 696 hits. Hence, the total amount of time in the up_idle is 691 out of 696 hits. Which correspond **~99.2%** of the time. Therefore the amount of CPU usage is very low. **~0.7%**.

3.5 Ping Pong communication

The following diagram depict how benchmarking will be performed.



The benchmarking would be done using different means of communication (UDP/serial/6lowpan).

3.6 Software "Benchmarked"

The software benchmarked are the examples taken from the NuttX Applications. The uros_agent will be running in the Linux docker.

For memory and performance execution, the application "Benchmarked" are the Nuttx examples applications: the publisher and subscriber. The application will use the UDP and serial as a communication medium. The two applications will be benchmarked for execution and heap memory performance analysis.

To benchmark communication performance, the application used is the Ping_pong application communicating over UDP to the uros_agent.

3.6.1 Restriction

Currently the serial subscriber is not "benchmarked" as the uros_agent does not bridge the serial to UDP/another serial. Hence, it will only be benchmarked using the UDP. # Results

This chapter gather results gotten from benchmarking according to the deliverable i.e. :

1. Resource usage: Memory usage and execution performances.
2. “Realtime-ness”: Being able to determine how deterministic is the micro-ROS stack.
3. Communication performances: UDP communication time.

3.7 Current development status of the memory tool

Currently tools are developed aiming embedded processors (ARM cortex M4) with the CoreSight IP implemented. The current tools allow the user to get information such as CPU usage, and heap memory usage.

3.8 Future works planned for the memory tool

In the future, more tools are to come for memory analysis:

- Stack usage per tasks
- Heap usage per tasks and kernel vs user-space
- Static memory usage (ROM) per layer

4 Benchmark results for Olimex board

4.1 Resource usage and communication performances

This section deals with the memory usage, execution performances and communication performances measurement and analysis. The output benchmark results are the fruit of the applied methodologies.

4.1.1 Memory analysis

The current version of the tools analyses the heap memory usage. Currently it does not distinguish user allocated memory and kernel allocated memory. The results will show the most relevant functions. In this document, the focus will be put on big allocations (> 1024 bytes) and repetitive allocations.

4.1.1.1 Memory heap first case: publisher example over serial The JSON output file results_data/mem_publisher_serial.json displays allocation that were made during the whole benchmarking session since platform boot-up. The publisher is sending 1000 messages to the uros_agent and wait 500ms.

The most relevant information are displayed below:

Line	Function	File:line	Count	Size
1	uxr_serialize_OBJK_ Endpoint_QosBinary	microxrcedds/Micro-XRCE-DDS-Client/src/ c/core/serialization/ xrce_protocol.c:960	1001	40
2	rcl_wait_set_init	ros2/rcl/rcl/src/rcl/wait.c:132	1001	104
3	rcl_wait_set_resize_timers	ros2/rcl/rcl/src/rcl/wait.c:443	1000	24
4	thread_create	nuttx/sched/task/ task_create.c:135	1	65016
5	task_spawn_exec	nuttx/sched/ task/task_spawn.c:142	1	2072

More details on those lines:

1. The first line is allocating memory for needed by the message serialization of the **microxrcedds protocol**.
2. The second line allocates memory for the **rcl_wait_set_t** implementation.
3. The third line re-allocates memory used to hold the **timer object**.
4. The fourth line shows that ~64Kbyte of memory ram is allocated upon **task create**. This allocates memory for the TCB (Thread Control Block) and stack to run the **publisher** application.
5. The fifth line correspond to the allocation of ~2Kbytes of memory ram upon **task_create**. This allocates memory for TCB (Thread Control Block) and stack to run the startup task.

4.1.1.2 Memory heap second case: subscriber example over serial The JSON output file results_data/mem_subscriber_serial.json displays allocation that were made during the whole benchmarking session since platform boot-up.

The most relevant information are displayed below:

Line	Function	File:line	Count
1	thread_create	nuttx/sched/task/task_create.c:135	1
2	task_spawn_exec	nuttx/sched/init/os_bringup.c:461	1

More details on those lines: case this is the creation of the new subscriber 1. The first line shows that ~64Kbyte of memory ram is allocated upon **task create**. This allocates memory for the TCB (Thread Control Block) and stack to run the **subscriber** application. 2. The second line correspond to the allocation of ~2Kbytes of memory ram upon task_create. This allocates memory for TCB (Thread Control Block) and stack to run the **startup task**.

Benchmarking was performed using only the subscriber (no publisher connected to the agent) thus no message were received. However, it is expected that UDP subscriber allocates more data when a message will be received.

4.1.1.3 Memory heap third case: publisher example over UDP

The JSON output file results_data/mem_publisher_udp.json displays allocation that were made during the whole benchmarking session since platform boot-up. The publisher is sending 1000 messages to the uros_agent and wait 500ms.

In comparison to the serial version of the publisher, it is expected that the number of allocations to be higher. The most relevant information are displayed below:

Line	Function	File:line	Count	Size
1	rcl_wait_set_ resize_timers	ros2/rcl/rcl/src/rcl/ wait.c:443	1001	24
2	nsh_netinit	apps/nshlib/nsh_netinit.c:842	1	1592
3	udp_pollsetup	nuttx/net/udp/udp_netpoll.c:184	1022	40
4	rmw_allocate	ros2/rmw/rmw/src/allocators.c:30	1001	30
5	up_create_stack	nuttx/arch/arm/src/common/up_createstack.c:194	1	65016
6	rcl_wait_set_init	ros2/rcl/rcl/src/rcl/wait.c:132	1001	104
7	work_hpstart	nuttx/sched/wqueue/kwork_hpthread.c:198	1	2072
8	work_lpstart	nuttx/sched/wqueue/kwork_hpthread.c:194	1	2072

More details on those lines: 1. The first line (re)-allocates memory used to hold the **timer object** used by rclc_spin_node_once. 2. The second line correspond to the allocation of memory needed for the **network structure in nuttx**. 3. The third line is an allocation of an **UDP poll structure**. The Nuttx's network subsystem is using the poll structure to monitor event on an UDP/IP socket. And upon write to the UDP socket, this structure is created. Thus the number of of allocation, 1022 correspond to the number of the number of time the publisher sends data (re-send in the case of a failure) on every iterations. 4. The fourth line shows that the **ros middleware** is allocating some space on every iterations. After tracing back to the source of call, the allocation is made by the **rclc_spin_node_once**. The latest is called on every iteration of the **publisher_main** function hence the 1001 allocations. 5. The fifth line shows that ~64Kbyte of memory ram is allocated upon **task create**. This allocates memory for the TCB (Thread Control Block) and stack to run the **publisher** application. 6. The sixth line of the table is the (re)-allocation of the the structure **rcl_wait_set_implementation**. This (re)-allocation is done upon each iteration of the loop in **publisher_main**. 7. The seventh line correspond to the allocation of ~2Kbytes of memory ram upon **task_create**. This allocates memory for TCB (Thread Control Block) and stack for **high priority work queue**. 8. The eighth line correspond to the allocation of ~2Kbytes of memory ram upon **task_create**. This allocates memory for TCB (Thread Control Block) and stack for a **low priority work queue**.

4.1.1.4 Memory heap fourth case: subscriber example over UDP

The JSON output file results_data/mem_subscriber_udp.json displays allocation that were made during the whole benchmarking session since platform boot-up. **The subscriber** is looping 50 times and exiting upon reception. Each loop will receive one message and wait 500ms.

We cannot make a direct comparison with the serial subscriber. Indeed in this benchamarking session, the subscriber was able to get messages from a publisher through an agent.

Line	Function	File:line	Count	Size
1	rcl_wait_set_ resize_subscriptions	ros2/rcl/rcl/src/rcl/ wait.c:378	51	24
2	rcl_wait_set_ resize_subscriptions	ros2/rcl/rcl/src/rcl/ wait.c:378	51	24
3	rcl_wait_set_init	ros2/rcl/rcl/src/rcl/wait.c:132	51	104
4	nsh_netinit	apps/nshlib/nsh_netinit.c:842	1	1592
5	udp_pollsetup	nuttx/net/udp/udp_netpoll.c:184	182	40
6	work_hpstart	nuttx/sched/wqueue/ kwork_hpthread.c:198	1	2072
7	work_lpstart	nuttx/sched/wqueue/ kwork_hpthread.c:194	1	2072
8	up_create_stack	nuttx/arch/arm/src/common/ up_createstack.c:194	1	65016

More details on those lines: 1. The first line is a (re)-allocation. It (re)-initialize the subscription space. This happens on every loop. So the number of allocations, in this case 51, corresponds to the number of iterations in the loop in the **main** function of the **subscriber**. The call is made from rcl_wait_set_resize_subscriptions by the macro **SET_RESIZE_RMW_REALLOC(...)**. 2. The second line is similar to the first, except that the call is made from rcl_wait_set_resize_subscriptions by the macro **SET_RESIZE(...)**. 3. The third line of the table is the (re)-allocation of the the structure **rcl_wait_set_implementation**. This (re)-allocation is done upon each iteration of the loop in **subscriber_main**. 4. The fourth line correspond to the allocation of memory needed for the network structure in nuttx. 5. The fifth line is an allocation of an **UDP poll structure**. The Nuttx's network subsystem is using the poll structure to monitor event on an UDP/IP socket. And upon write to the UDP socket, this structure is created. Thus the number of of allocation, 182 correspond to the number of the number of time the sends data (re-send in the case of a failure) on every iterations. 6. The sixth line correspond to the allocation of ~2Kbytes of memory ram upon **task_create**. This allocates memory for TCB (Thread Control Block) and stack for **high priority work queue**. 7. The sevent line correspond to the allocation of ~2Kbytes of memory ram upon **task_create**. This allocates memory for TCB (Thread Control Block) and stack for a **low priority work queue**. 8. The eighth line shows that ~64Kbyte of memory ram is allocated upon task create. This allocates memory for the TCB (Thread Control Block) and stack to run the **subscriber** application.

4.1.2 Performance analysis

In this section, the execution results will be taken from the PEA tools. Currently, the implementation only retrieves a PC sample, and does not trace where was the call from. The sampling rate is set to **1/16384** of the CPU clock. As the CPU clock is 168 MHz, the sampling frequency is ~10MHz. And packet are emitted ad the frequency of 168MHz/256M **~0,65625MHz**.

4.1.2.1 Performance First case: publisher example over serial The results of the execution performance regarding the publisher over serial are shown below:

```
file: /root/nuttx/sched/init/os_start.c
function: os_start
```

```
hits: 1692
details: [
    { hits: 678, line: 867, address: 8001d7e},
    { hits: 1014, line: 867, address: 8001d82},

]
file: /root/nuttx/arch/arm/src/chip/stm32_idle.c
function: up_idle
hits: 3304
details: [
    { hits: 657, line: 433, address: 8007298},
    { hits: 337, line: 433, address: 800729a},
    { hits: 338, line: 448, address: 800729c},
    { hits: 660, line: 448, address: 800729e},
    { hits: 653, line: 448, address: 80072a0},
    { hits: 659, line: 448, address: 80072a2},

]
file: /root/nuttx/arch/arm/src/chip/stm32_otgfsdev.c
function: stm32_ep0out_setup
hits: 1
details: [
    { hits: 1, line: 2564, address: 8008870},

]
file: /root/nuttx/fs/vfs/fs_poll.c
function: poll_teardown
hits: 13
details: [
    { hits: 2, line: 244, address: 8025ac8},
    { hits: 1, line: 251, address: 8025ade},
    { hits: 1, line: 251, address: 8025ae0},
    { hits: 1, line: 256, address: 8025afc},
    { hits: 1, line: 256, address: 8025b02},
    { hits: 1, line: 258, address: 8025b0e},
    { hits: 1, line: 258, address: 8025b1a},
    { hits: 1, line: 258, address: 8025b1e},
    { hits: 1, line: 260, address: 8025b24},
    { hits: 1, line: 297, address: 8025b8a},
    { hits: 1, line: 251, address: 8025b9a},
    { hits: 1, line: 301, address: 8025ba4},

]
file: /root/nuttx/fs/vfs/fs_read.c
function: read
hits: 1
details: [
```

```
    { hits: 1, line: 225, address: 8025e4c},

]
file: /build/gcc-arm-none-eabi-iopiMw/gcc-arm-none-eabi-6.3.1+svn253039/build/
arm-none-eabi/thumb/v7e-m/libgcc/../../../../../src/libgcc/libgcc2.c
function: __udivmoddi4
hits: 14
details: [
    { hits: 1, line: 1012, address: 8029920},
    { hits: 1, line: 1060, address: 802992e},
    { hits: 2, line1692: 1074, address: 8029940},
    { hits: 1, line: 1074, address: 8029950},
    { hits: 1, line: 1078, address: 802995c},
    { hits: 2, line: 1078, ad]dress: 8029974},
    { hits: 2, line: 1078, address: 8029990},
    { hits: 1, line: 1078, address: 802999c},
    { hits: 3, line: 1129, address: 80299d0},

]
file: /root/nuttx/sched/semaphore/sem_wait.c
function: nxsem_wait
hits: 25
details: [
    { hits: 1, line: 87, address: 802ae74},
    { hits: 3, line1692: 88, address: 802ae7c},
    { hits: 2, line: 105, address: 802ae94},
    { hits: 1, line: 109, address: 802ae9c},
    { hits: 1, line: 109, address: 802aea0},
    { hits: 1, line: 109, address: 802aea2},
    { hits: 1, line: 113, address: 802aea8},
    { hits: 1, line: 113, address: 802aeae},
    { hits: 2, line: 113, address: 802aeb4},
    { hits: 1, line: 115, address: 802aeb6},
    { hits: 1, line: 115, address: 802aeba},
    { hits: 1, line: 116, address: 802aebc},
    { hits: 2, line: 116, address: 802aec0},
    { hits: 1, line: 135, address: 802aece},
    { hits: 1, line: 139, address: 802aed8},
    { hits: 1, line: 168, address: 802aee8},
    { hits: 1, line: 168, address: 802aeea},
    { hits: 1, line: 211, address: 802af18},
    { hits: 2, line: 211, address: 802af1e},

]
file: /root/nuttx1692/sched/semaphore/sem_tickwait.c
function: nxsem_tickwait
hits: 12
```

```
details: [
    { hits: 1, line: 88, address: 802b00e},
    { hits: 1, line: 102, address: 802b016},
    { hits: 1, line: 102, address: 802b01e},
    { hits: 1, line: 116, address: 802b038},
    { hits: 1, line: 120, address: 802b03c},
    { hits: 1, line: 152, address: 802b070},
    { hits: 1, line: 152, address: 802b080},
    { hits: 1, line: 157, address: 802b08c},
    { hits: 1, line: 165, address: 802b09a},
    { hits: 1, line: 165, address: 802b0a2},
    { hits: 2, line: 178, address: 802b0c6},

]
file: /root/nuttx/libs/libc/stdio/lib_libvsprintf.c
function: lib_vsprintf
hits: 1
details: [
    { hits: 1, line: 1170, address: 802d9ce},

]
file: /root/nuttx/libs/libc/stdio/lib_libfwrite.c
function: lib_fwrite
hits: 1
details: [
    { hits: 1, line: 71, address: 802e416},

]
file: /root/nuttx/drivers/usbdev/cdcacm.c
function: cdcacm_sndpacket
hits: 1
details: [
    { hits: 1, line: 453, address: 8030976},

]
file: /root/nuttx/drivers/usbdev/cdcacm.c
function: cdcuart_txint
hits: 1
details: [
    { hits: 1, line: 2804, address: 8031a54},

]
```

In total there are 5016 samples (hits) received. The serial communication is very slow and mostly handled over DMA. Hence the most of the time, the CPU is idling. This is demonstrated by the above result: * 1642 hits in the os_start line 867 which correspond to a function called in the idle_task. * 3304 hits caught in the up_idle.

Therefore, the total amount of hits in idle are 4946.

Function	Hits	Percentage of the total time
idle_task	4946	~99%

As explained above, the most of the transaction are done over DMA. And as UART communication is very slow, the CPU is idling most of the time.

4.1.2.2 Performance Second case: subscriber example over serial Test were performed using only the subscriber (no publisher) thus no messages were received.
Hence, the performance benchmarking could not be performed.

4.1.2.3 Performance Third case: publisher example over UDP The results of the execution performance regarding the UDP publisher are shown below:

```
file: /root/nuttx/arch/arm/src/chip/stm32_idle.c
function: up_idle
hits: 1792
details: [
    { hits: 360, line: 433, address: 800758c},
    { hits: 177, line: 433, address: 800758e},
    { hits: 177, line: 448, address: 8007590},
    { hits: 360, line: 448, address: 8007592},
    { hits: 357, line: 448, address: 8007594},
    { hits: 361, line: 448, address: 8007596},

]
file: /root/nuttx/arch/arm/src/chip/stm32_eth.c
function: stm32_phyread
hits: 935
details: [
    { hits: 1, line: 2980, address: 800b046},
    { hits: 1, line: 2980, address: 800b048},
    { hits: 2, line: 2986, address: 800b050},
    { hits: 1, line: 2993, address: 800b05a},
    { hits: 1, line: 2994, address: 800b068},
    { hits: 1, line: 2994, address: 800b06a},
    { hits: 2, line: 2995, address: 800b074},
    { hits: 72, line: 3003, address: 800b088},
    { hits: 207, line: 3003, address: 800b08a},
    { hits: 40, line: 3003, address: 800b08c},
    { hits: 40, line: 3003, address: 800b090},
    { hits: 81, line: 3003, address: 800b092},
    { hits: 1, line: 3005, address: 800b094},
```

```
    { hits: 1, line: 3005, address: 800b096},
    { hits: 1, line: 3006, address: 800b0a0},
    { hits: 81, line: 3001, address: 800b0a2},
    { hits: 37, line: 3001, address: 800b0a4},
    { hits: 46, line: 3001, address: 800b0a6},
    { hits: 44, line: 3001, address: 800b0a8},
    { hits: 154, line: 3001, address: 800b0aa},
    { hits: 41, line: 3001, address: 800b0ac},
    { hits: 79, line: 3001, address: 800b0ae},
    { hits: 1, line: 3014, address: 800b0bc},

]
file: /root/nuttx/arch/arm/src/chip/stm32_eth.c
function: stm32_phyinit
hits: 5
details: [
    { hits: 1, line: 3218, address: 800b1b8},
    { hits: 1, line: 3219, address: 800b1c4},
    { hits: 2, line: 3224, address: 800b1d0},
    { hits: 1, line: 3216, address: 800b1e2},

]
file: /root/nuttx/sched/sched/sched_roundrobin.c
function: sched_roundrobin_process
hits: 1
details: [
    { hits: 1, line: 168, address: 8042068},

]
```

The output of the performances demonstrates that the most of the time was spent in the the stm32_phyread at the line 3005 (207 hits).

This correspond to this snipped of code:

```
  /* Wait for the transfer to complete */
  for (timeout = 0; timeout < PHY_READ_TIMEOUT; timeout++)
    {
      if ((stm32_getreg(STM32_ETH_MACMIIAR) & ETH_MACMIIAR_MB) == 0)
        {
          *value = (uint16_t)stm32_getreg(STM32_ETH_MACMIIDR);
          return OK;
        }
    }
```

Thus, it is clear that the application is spending most of its time waiting for initiated transfer to be completed.

Function	Hits	Percentage of the total time
idle_task	1792	~65%
phy_read	935	~34%

The time spent in the other functions are insignificants. The above results show that time was mostly spent in the phy_read to wait for the frame to be sent.

4.1.2.4 Performance Fourth case: subscriber example over UDP The results of the execution performance regarding the UDP subscriber are shown below:

```
file: /root/nuttx/arch/arm/src/chip/stm32_idle.c
function: up_idle
hits: 288
details: [
    { hits: 56, line: 433, address: 800758c},
    { hits: 30, line: 433, address: 800758e},
    { hits: 29, line: 448, address: 8007590},
    { hits: 58, line: 448, address: 8007592},
    { hits: 56, line: 448, address: 8007594},
    { hits: 59, line: 448, address: 8007596},

]
file: /root/nuttx/arch/arm/src/chip/stm32_eth.c
function: stm32_phyread
hits: 1212
details: [
    { hits: 1, line: 2980, address: 800b046},
    { hits: 1, line: 2980, address: 800b04a},
    { hits: 1, line: 2986, address: 800b050},
    { hits: 1, line: 2987, address: 800b058},
    { hits: 1, line: 2993, address: 800b05a},
    { hits: 1, line: 2994, address: 800b066},
    { hits: 1, line: 2994, address: 800b068},
    { hits: 2, line: 2994, address: 800b06e},
    { hits: 1, line: 2994, address: 800b072},
    { hits: 1, line: 2997, address: 800b07e},
    { hits: 1, line: 2997, address: 800b080},
    { hits: 103, line: 3003, address: 800b088},
    { hits: 259, line: 3003, address: 800b08a},
    { hits: 57, line: 3003, address: 800b08c},
    { hits: 51, line: 3003, address: 800b090},
    { hits: 105, line: 3003, address: 800b092},
    { hits: 1, line: 3005, address: 800b096},
    { hits: 100, line: 3001, address: 800b0a2},
    { hits: 53, line: 3001, address: 800b0a4},
```

```
    { hits: 54, line: 3001, address: 800b0a6},
    { hits: 54, line: 3001, address: 800b0a8},
    { hits: 206, line: 3001, address: 800b0aa},
    { hits: 50, line: 3001, address: 800b0ac},
    { hits: 106, line: 3001, address: 800b0ae},
    { hits: 1, line: 3014, address: 800b0bc},

]
file: /root/nuttx/arch/arm/src/chip/stm32_eth.c
function: stm32_phyinit
hits: 7
details: [
    { hits: 1, line: 3218, address: 800b1b4},
    { hits: 1, line: 3218, address: 800b1bc},
    { hits: 2, line: 3219, address: 800b1c8},
    { hits: 1, line: 3224, address: 800b1d0},
    { hits: 1, line: 3216, address: 800b1d8},
    { hits: 1, line: 3216, address: 800b1de},

]
```

The output of the performances demonstrates that the most of the time was spent in the the stm32_phyread at the line 3005 (207 hits).

This correspond to this snipped of code:

```
  /* Wait for the transfer to complete */
  for (timeout = 0; timeout < PHY_READ_TIMEOUT; timeout++)
    {
      if ((stm32_getreg(STM32_ETH_MACMIIAR) & ETH_MACMIIAR_MB) == 0)
        {
          *value = (uint16_t)stm32_getreg(STM32_ETH_MACMIIDR);
          return OK;
        }
    }
```

Once a frame was received from the Ethernet, the kernel initiate the frame transfer and wait for the whole frame to be transfered.

Function	Hits	Percentage of the total time
idle_task	288	~15%
phy_read	1212	~84%

The time spent in the other functions are insignificant.

The above results show that time was mostly spent in the phy_read to transfer the received frame.

4.1.3 Ping Pong benchmarking

The ping pong benchmarking result currently cannot be compared to other benchmarking. Indeed the only medium of communication used is UDP.

However, the results can provide insights for next benchmarkings. The ping_pong results shows that the amount of time for a round trip message. The application showed a delay of 430ms for the whole round trip.

```
Total time 0 seconds 430000000 nanoseconds
```

5 Conclusion

In conclusion, the results shows that a lot of allocation are done and redone upon each publications/subscriptions. This could be reduce by pre-allocating memory before a loop if known in advance. In addition, as expected the use of a UDP is greedier in resources than the serial. Both CPU time wise and memory wise. Indeed as the speed of the Ethernet communication is fast and CPU is busier using Ethernet than using serial communication. Also the memory use is greater, it needs additional memory for the whole UDP/IP stack and some additional functionalities that are needed such as the priorities work queues and polling structures. The use of the communication medium has a huge impact on the resources. Hence, the medium shall be carefully selected depending on the micro-controller capabilities and application needs.

References

[1] A. Malki, 'micro-ROS benchmarking tools'. [Online]. Available: https://github.com/micro-ROS/benchmarking

[2] 'ARM Debug Interface V5 (ADI)'. [Online]. Available: https://static.docs.arm.com/ihi0031/c/IHI0031C_debug_interface_as.pdf

[3] 'ARMV7-M Architecture Reference Manual Section C1.6'. [Online]. Available: https://web.eecs.umich.edu/~prabal/teaching/eecs373-f10/readings/ARMv7-M_ARM.pdf

[4] 'CoreSight Design Kit revision r0p1 Technical Reference Manual'. [Online]. Available: http://infocenter.arm.com/help/topic/com.arm.doc.ddi0314h/DDI0314H_coresight_components_trm.pdf

[5] 'Keil CoreSight webpage'. [Online]. Available: http://www2.keil.com/coresight/

[6] 'The backtrace library'. [Online]. Available: https://github.com/red-rocket-computing/backtrace