D5.5

Micro-ROS benchmarking and validation tools Release - Beta

Grant agreement no.	780785
Project acronym	OFERA (micro-ROS)
Project full title	Open Framework for Embedded Robot Applications
Deliverable number	D5.5
Deliverable name	Micro-ROS benchmarking and validation tools Release - Beta
Date	December 2019
Dissemination level	public
Workpackage and task	5.3
Author	Tomasz Kołcon (Łukasiewicz-PIAP), Alexandre Malki (Łukasiewicz-PIAP)
Contributors	Mateusz Maciaś (Łukasiewicz-PIAP), Jan Staschulat (Bosch)
Keywords	Microcontroller, NuttX, Benchmarking
Abstract	This deliverable summarizes works in Task 5.3: Benchmark tooling for application developers.

Contents

1 Summary

This deliverable summarizes the works in Task 5.3: Benchmark tooling for application developers. It is a description of beta release of benchmarking tools that will be continued and described in D5.6. The tools developed in the scope of task 5.3 are used internally as well as released as open source packages.

This document starts with general description of the created tools. This is followed by a descriptions of hardware tools used for benchmarking. Next, tools setup will be presented. Subsequently, use examples are shown. At the end, conclusions and future plans are described.

Term	Definition
ROS	Robot Operating System
MCB	Multi-connectors board
SWO	Single Wire Output
SWD	Serial Wire Debug
JTAG	Joint Test Action Group (also name of interface)
ITM	Instrumentation (or Instruction) Trace Macrocell
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
JSON	JavaScript Object Notation
AST	Abstract Syntax Tree
TFA	Trace Framework Abstraction

2 General description

2.1 Introduction to Benchmarking

Developing a working and stable application from the scribbles to the final executing binary is long and hard task. During this process developers may come across stabilities issues, performances issues. In addition to these issues, some specified QoS might be difficult to quantify. Solving those problems without the proper tools might be frustrating, tedious tasks leading to reduce developers efficiency. An adapted benchmarking tool could overcome all those development obstacles and increase development time. There are different KPI (Keep Performance Indicators) that one might be interested into. In the framework of this micro-ROS, the KPI can be freely chosen by the developer. In this way, the benchmarking tool will remain flexible and allow the community to constantly add some support for a lot of different KPI. The benchmarking tool is also intended for embedded software benchmarking with low/none overhead intrusion.

The problems we want to tackle are:

- Out there, many benchmarking tools exist. Each of them targeting different KPIs
- Different platforms (Linux/Nuttx/Baremetal et.c.)
- Too few time/resources to code benchmarking tool for each KPI,
- Avoid code overhead: Keep code clarity,
- Avoid execution overhead: Do not want to make execution slower when benchmarking.

Current benchmarking documentation is always available on:

- https://micro-ros.github.io/docs/concepts/benchmarking [1].

2.2 Benchmarking tool framework

The benchmarking tool under development is providing a framework to allow developers to create their own benchmarking solution. Each part a developer wants to benchmark can be added as a plugin using the provided framework. In this way plugins can be shared and this improves reusability as much as possible.

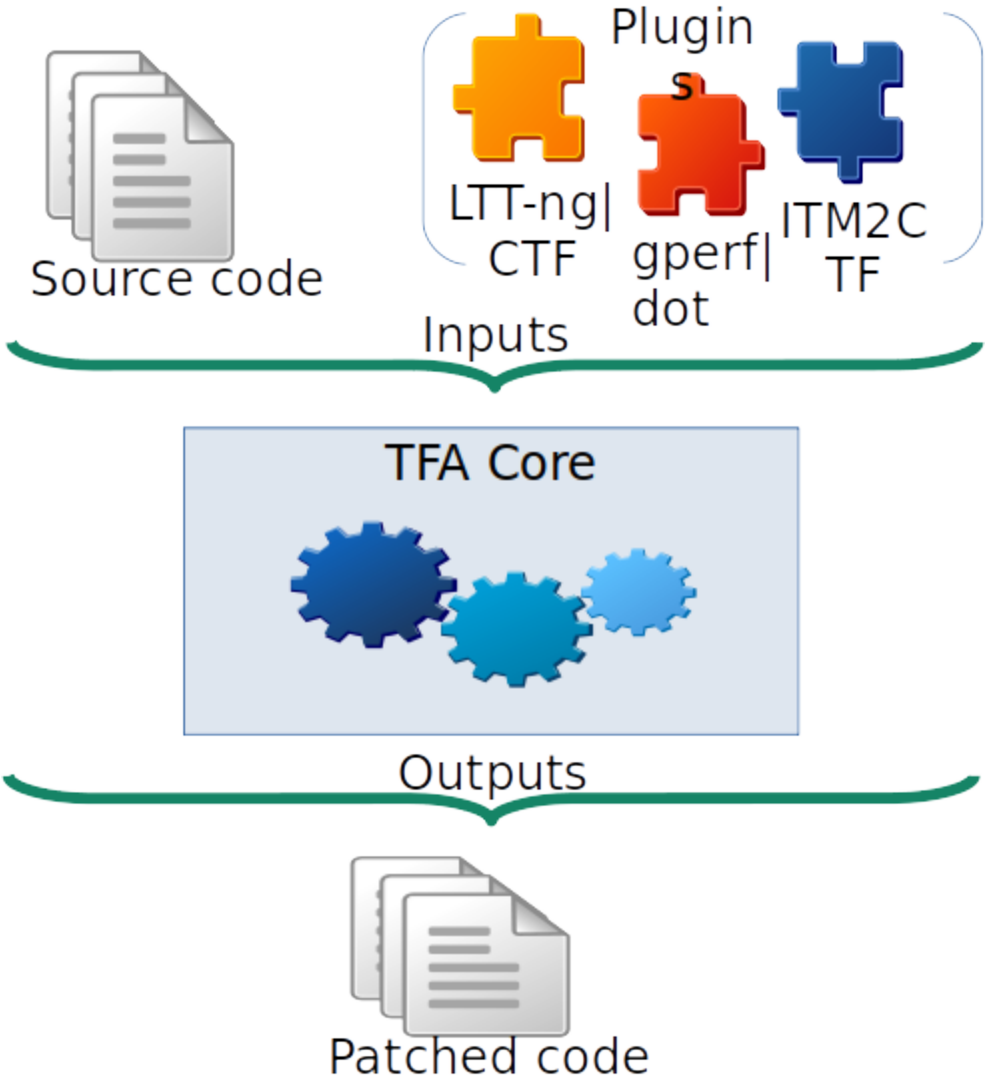

Figure 1: Shadow building

2.3 Trace Framework Abstraction

The Shadow builder alone only parses comments from the application and passes it along to the Trace Framework Abstraction (TFA) Core. The TFA core is aware of the plugins that are available, all the plugins' capabilities and platform target. The process goes as explained below:

- The line containing the functionality Benchmarking::XX::YY will be checked against all the available plugins.
- Plugins that are capable of handling the functionality will respond with a piece of code that later will be written withing the source code.
- Then the output file will be added in a folder corresponding to the platform type and benchmarking type.

Being generic is the key for this benchmarking tool. The plugins will, in contrary, bring the specific implementation needed to benchmark a specific platform. Every plugin will provide information as requested by the parser:

- Provide a list of supported platforms.
- Provide a list of functions that are handled.
- Provide snippets codes that will be added for benchmarking.
- Provide a list of patches and/or patch code
- Optional provide an end script to run and execute the benchmarks

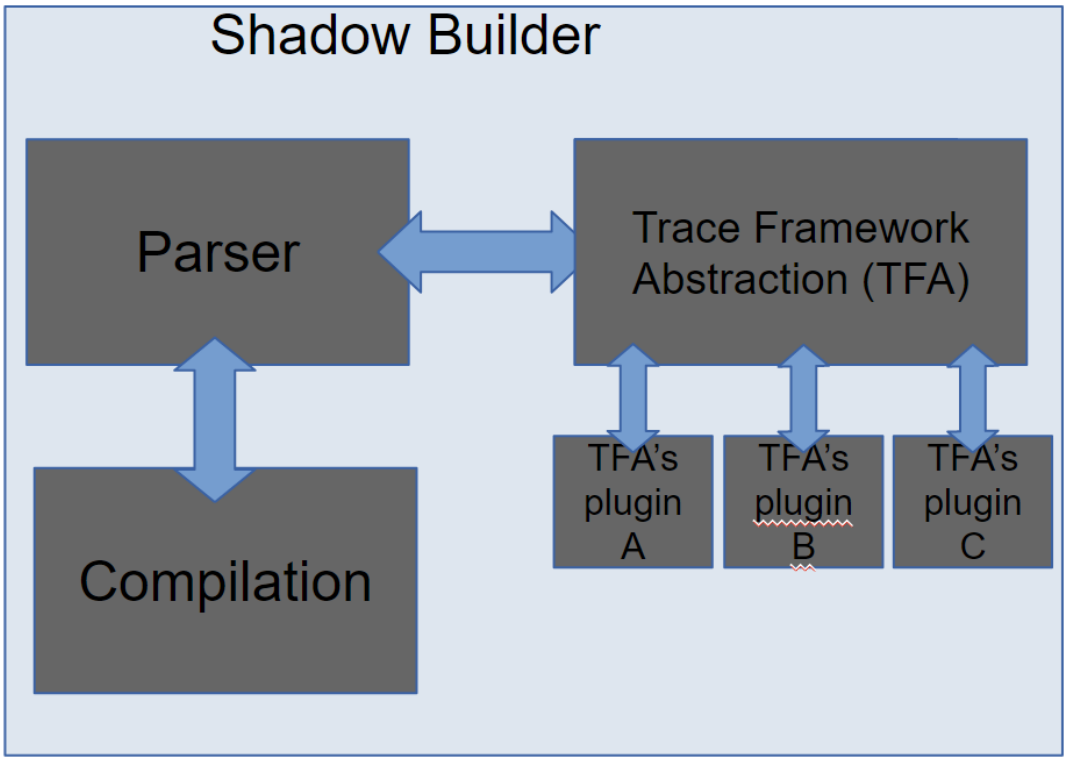

Figure 2: Shadow building

2.4 Shadow Builder

This section will introduce some concept related to the Shadow Builder (SB).

The Shadow Builder is a tool that will transparently instrument the code to benchmark. The tools will be able to output an "instrumented code" that will be later be compiled as a normal code. The following steps describe what the shadow builder process flow:

- Get configuration file from the user (Benchmarking Configuration File).
- Get appropriate sources.
- Execute Trace Framework Abstraction Configuration file.
- Parse the sources file needed then injects the code.
- Compile the targeted binary for different platform.

- If needed, depending what type benchmark is undertaken, compile another target binary benchmarking.

The SB (Shadow Builder) is meant to be as transparent as possible for the user. And if the benchmarking is not activated, it should be bypassed. The SB is in charge of getting the path/git repository to the source code that needs to be benchmarking. The sources are specified by the user in the benchmarking configuration file. In order to inject code, there are some tools that allow this. Clang AST tool will allow to inject some code.

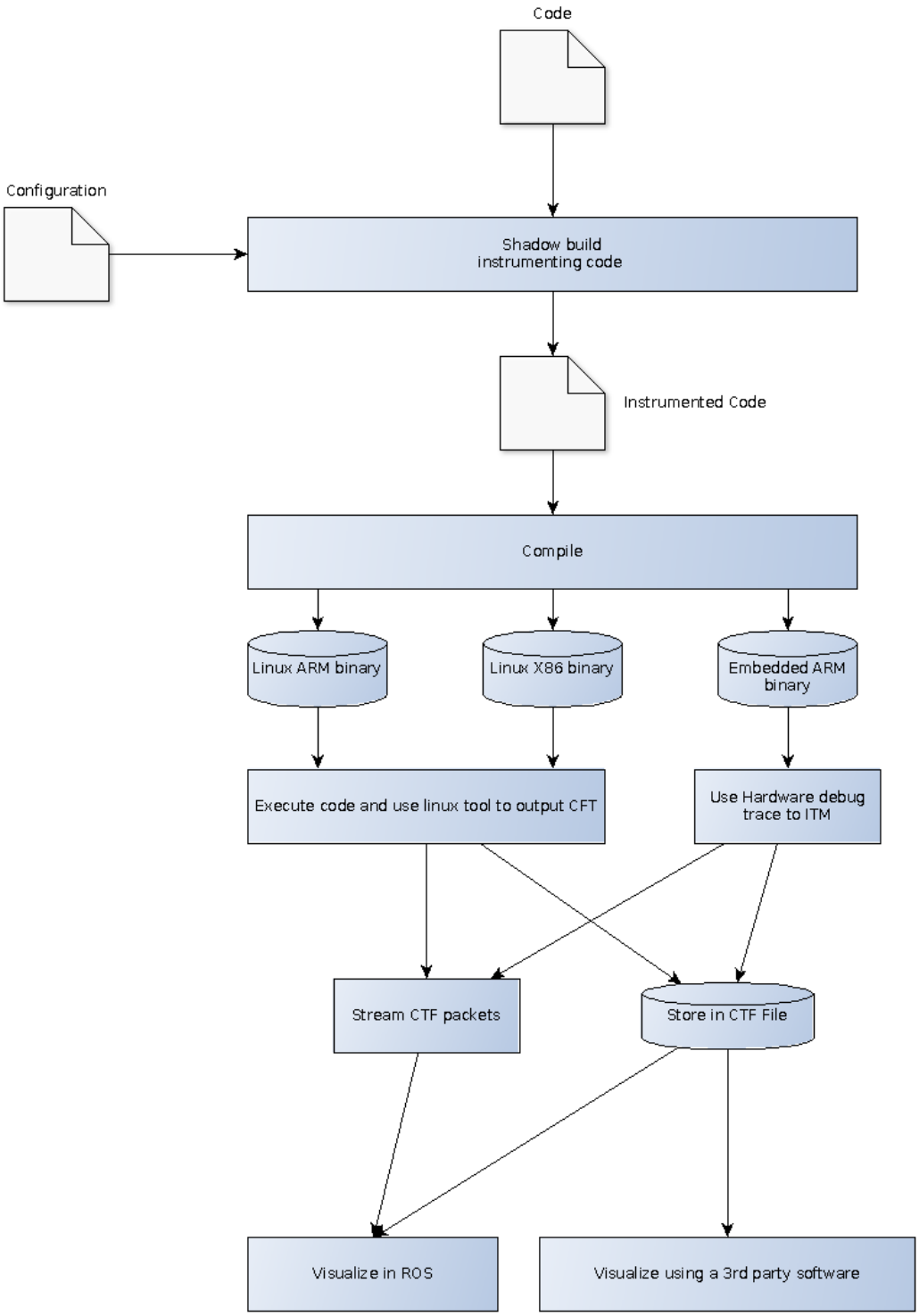

Figure 3: Shadow building

2.5 Binary generation for instrumented code

The binary generation is the process of compiling the source code. In order to benchmark, previously to compile the source code, it is necessary to instrument the code. The code will be instrumented

in a transparent way for the programmer/user. Therefore, a configuration file provided by the programmer will be parsed and code injected as described in a configuration file.

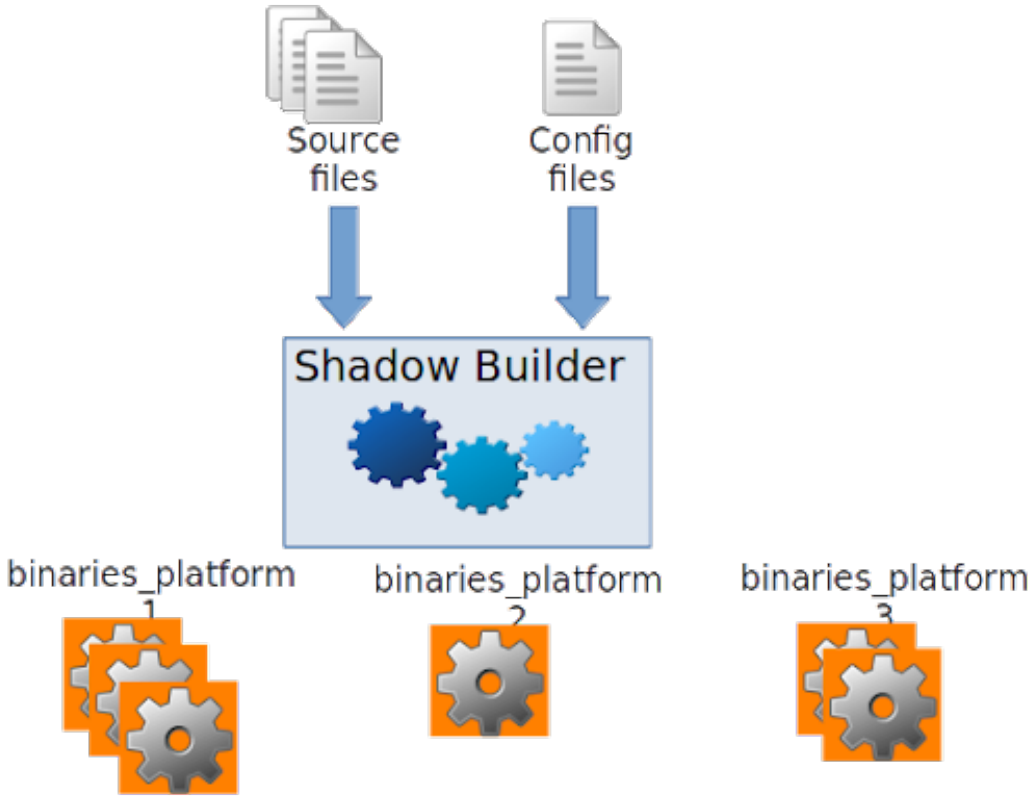

Figure 4: Shadow building

2.5.1 Receiving inputs

The binary generation's pipeline receives two inputs to work with:

- Configuration Benchmarking file.
- Source code to benchmark.

In short, the configuration describes:

- What is benchmarked (sources).
- Where to benchmark.
- What type of benchmark.
- Optionally against what base line to compare (base line source)

2.5.2 Parse and Check

Once the input received the Shadow Builder parses the configuration file. From the configuration file, the Shadow Builder gets:

- The different benchmarking to be achieved.
- The targeted platforms.

In addition to parsing, the Shadow Builder is in charge of checking capabilities and consistency within the configuration file and the different TFA's plugins registered in the TFA module.

2.5.3 TFA Execution

Once parsed and checked against the TFA module capabilities, the Shadow Builder will be in charge of translating configuration into source code. The translated sources will also be achieved in cooperation with the TFA module. At the end of this step, the TFA will generate the new forged source code ready for compilation. In addition to patched source code, the TFA will generate scripts that will the benchmarks.

2.5.4 Compilation

The compilation will happen for every kind of benchmarks and platforms targeted. Depending on the kind of benchmark that is being executed, there will be one or more binaries per benchmarks session. The number of binary generated also depends on what plugins are provided by the user to the shadow builder. The Shadow Builder will retrieve capabilities of the plugins and request from the developer, match them and generated software according to the matches.

2.5.5 Step to start benchmarking

The Shadow Builder will be executed as follow:

- Software sources are passed to the Shadow Builder.
- The source are passed and upon comments containing /Benchmarking::XX::YY/ (a tag) the code line is passed to the Trace Framework Abstraction module. Using comments is preferable → No includes needed.
- All plugins that registered to the TFA the Benchmarking::XX::YY functionality will return a piece of code that will be added to the source.
- Once all parsed, the shadow builder will compile for all the different platforms requested either by plugins or by user configuration.

2.6 Executor Benchmarking Suite

In a bilateral open-source cooperation, Nobleo Technology from Eindhoven (The Netherlands) and Bosch developed a small tool suite for benchmarking of ROS 2 Executor implementations. With a few number of parameters, like number of topics, message size, publication frequency and setup topology, a test-scenario is generated. In a second step, the actual testbench is created consisting of ROS2 nodes, publishers, subscribers according to the specified test-scenario. Finally, the latency is measured using performance tracing.

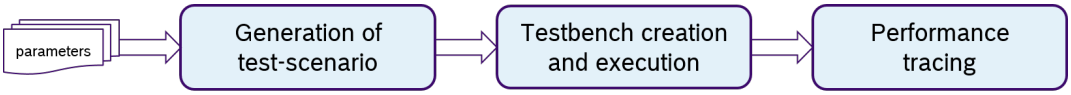

Figure 5: Overview Testbench

Several topology types represent different communication use-cases. For example, from very simple ones (types A-B), in which there is only publisher and multiple subscribers to more complex ones (E,F), in which multiple topics are published and received. On particular interest was to evaluate the performance overhead of RCL, if all publishers and subscribers are in their own ROS2 node or if all communication is in one ROS2 node. This is represented in topology F.

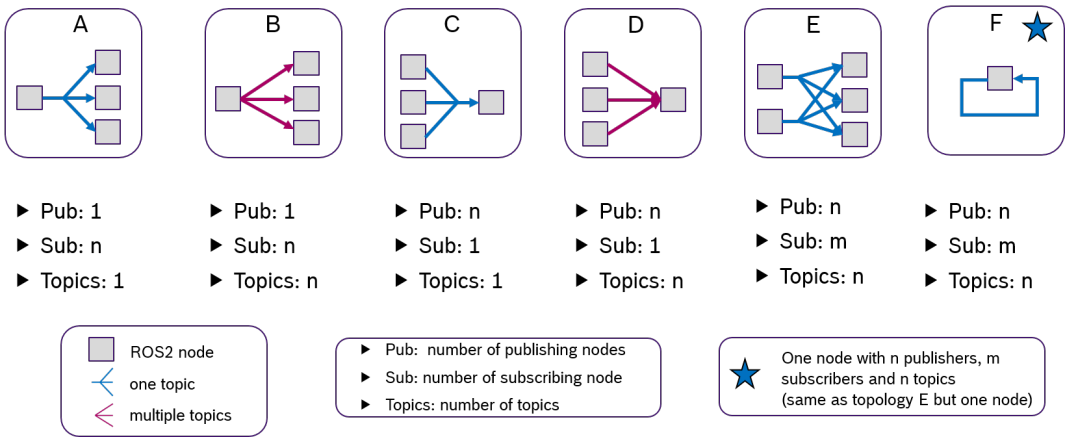

Figure 6: Topology of test scenarios

The tool suite has been published at this Nobelo repository and this Micro-ROS repository. Together with the runtime tracing tools developed in OFERA for ROS 2 and micro-ROS, it has been used to identify performance issues in the default rclcpp Executor as well as to evaluate the performance of the LET Executor for micro-ROS. On the basis of the analysis results on the default rclcpp Executor, Nobleo Technology even developed an improved, static version executor and contributed it to the core rclcpp repo in the Pull request 873.

2.7 Network emulation tool

In order to benchmark the tool in different "network conditions", the use of a tool from Linux called Netem[2] that is open-source.

3 Hardware used

The idea is to use open-source tools to automate the benchmarks. There is possibility to use separate small and simple tools or use one specialized board designed by Łukasiewicz-PIAP called MCB (Multi Connectors Board).

3.1 MCB

MCB board is intended to automate benchmarks execution and to avoid cable clutter. Full functionality can be obtained for ARM processors equipped with CoreSight on-chip trace and debug features.

Main features of the current version of board are listed below:

- Allows sniffing SWO trace line
- Automates current measurements
  - Contains set of shunt resistors (many possible ranges)
  - Newest version of the board contains module to measure current on it
  - All versions allow to trigger external oscilloscope for very accurate observation of current changes
- Additional connectors for Saleae/Sigrok logic analyzers
  - JTAG/SWD sniffing
  - SWO trace output decoding
- All trace pins decoding (D0, D1, D2, D3 and CLK)
- Easy configuration on jumpers

According to the previous version FT232 was replaced by FT2232.

Board project sources are available on: https://github.com/micro-ROS/mcb [3].

Like in the work described in D5.2 JTAG adapter is used to connect software benchmarking tools to MCB. We use ST-LINK debugger, but it can be any JTAG adapter supported by OpenOCD.

The basic platform for micro-ROS benchmarking tools developing is Olimex STM32-E407.

4 Tools setup

4.1 Sources

Sources are available here: https://github.com/microROS/benchmarking [4].

4.2 How to setup

4.2.1 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

```
foo@bar:~$ sudo apt install \
    autoconf \
    autotools-dev \
    binutils \
    check \
    doxygen \
    gcc-arm-none-eabi \
    git \
    libtool \
    libusb-1.0 \
    libftdi-dev \
    make \
    pkg-config \
    texinfo
```

4.2.2 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)

4.3 How to use

4.3.1 Performance execution analysis

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

4.3.1.1 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**

4.3.1.2 Execution

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.

4.3.2 Memory footprint analysis

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

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**

Execution

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

To execute it:

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

4.3.3 Network emulation tool

The commands that were use are:

- To introduce a network delay (latency) the command to use is

```
$  tc qdisc change dev eth0 root netem delay Xms # Where X is the delay in
milliseconds
```

- To introduce packet losses:

```
$   tc qdisc add dev eth0 root netem loss Y% # Where Y is the pourcentage of
packet that will be lost
```

- To introduce packet corruption:

```
$  tc qdisc change dev eth0 root netem corrupt Z% # Where Z is the pourcentage
of byte corrupted in a packet.
```

- To measure the packets that are travelling on the wire, the tshark was used:

```
$  tshark -i interface -w output.pcap
```

- To extract general information form the packet

```
$  capinfo output.pcap
```

5 Tools usage examples

5.1 Performance execution analysis

Using Micro-XRCE-DDS over Ethernet Performance analysis example.

The output of the performance benchmarking tools is shown below:

```
file: /uros_ws/firmware/NuttX/sched/init/nx_start.c
function: nx_start
hits: 618
details: [
    { hits: 311, line: 876, address: 8001120},
    { hits: 307, line: 876, address: 8001124},

]
file: /uros_ws/firmware/NuttX/arch/arm/src/chip/stm32_idle.c
function: up_idle
hits: 311
details: [
    { hits: 311, line: 433, address: 80039fc},

]
file: /uros_ws/firmware/NuttX/arch/arm/src/chip/stm32_eth.c
function: stm32_phyread
hits: 618
details: [
    { hits: 2, line: 2980, address: 80056a8},
    { hits: 1, line: 2986, address: 80056aa},
    { hits: 1, line: 2987, address: 80056b4},
    { hits: 1, line: 2993, address: 80056c0},
    { hits: 1, line: 2997, address: 80056c6},
    { hits: 1, line: 3001, address: 80056ca},
    { hits: 58, line: 3001, address: 80056cc},
```

```
    { hits: 34, line: 3001, address: 80056ce},
    { hits: 68, line: 3001, address: 80056d0},
    { hits: 163, line: 3003, address: 80056d8},
    { hits: 31, line: 3003, address: 80056da},
    { hits: 68, line: 3003, address: 80056de},
    { hits: 1, line: 3005, address: 80056e2},
    { hits: 1, line: 3005, address: 80056e4},
    { hits: 1, line: 3014, address: 80056e8},
    { hits: 57, line: 3001, address: 80056ea},
    { hits: 33, line: 3001, address: 80056ec},
    { hits: 32, line: 3001, address: 80056ee},
    { hits: 64, line: 3001, address: 80056f0},

]
```

5.2 Memory analysis

Using Micro-XRCE-DDS over Serial Memory analysis

Size of allocation that are beyond 1024 bytes. It corresponds to a little less 1/100 of the total amount of the reference board's SRAM (196 KBytes):

Line	Function	File	Count	Size (bytes)
1	task_spawn	/uros_ws/firmware/NuttX/ sched/task/task_spawn.c	1	65016
2	emutls_alloc	/build/gcc-arm-none-eabi…/ ../src/libgcc/emutls.c	1	1048
3	exception_common	/uros_ws/firmware/NuttX/ …v7-m/gnu/up_exception.S	1	2072
4	work_lpstart	/uros_ws/firmware/NuttX/ …wqueue/kwork_lpthread.c	1	2072

5.3 Network emulation tool

Network delay (latency) 500ms:

```
$  tc qdisc change dev eth0 root netem delay 500ms
```

6 Conclusions

Tools described here were developed to improve quality micro-ROS. The other goals were to make development and benchmarking more convenient. Thanks to the very general approach, it will also be possible to use them in other projects based so far on C/C++. We hope that the potentially large

possibilities and simplicity of their use will allow for extensive use in the future. Work will continue and finally described in D5.6. The tools created in the project will continue to be developed by the community.

References

[1] A. M. OFERA, 'Benchmarking concepts'. [Online]. Available: https://micro-ros.github.io/docs/concepts/benchmarking/

[2] S. Hemminger, 'Netem tool'. [Online]. Available: https://wiki.linuxfoundation.org/networking/netem

[3] Łukasiewcz-PIAP, 'MCB'. [Online]. Available: https://github.com/micro-ROS/mcb

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