D4.2

Micro-ROS Client library Software Release Y2

Grant agreement no.	780785
Project acronym	OFERA (micro-ROS)
Project full title	Open Framework for Embedded Robot Applications
Deliverable number	D4.2
Deliverable name	Micro-ROS Client library
Date	December 2019
Dissemination level	public
Workpackage and task	4.2
Author	Borja Outerelo Gamarra (eProsima)
Contributors	Ralph Lange (Bosch)
Keywords	micro-ROS, robotics, ROS, microcontrollers, rclc, client library
Abstract	This document provides links to the released software and documentation for deliverable D4.2 *Micro-ROS client library Software Release Y2* of the Task 4.2 *micro-ROS client library* (*urcl*).

Contents

1 Summary

With the latest developments done in micro-ROS this year, OFERA consortium has decided to follow a modular approach for the client library. With this modular approach, the micro-ROS client library software release is a collection of multiple repositories. This decision has been taken following a decision paper, presented as an annexe in this document. In this year micro-ROS has adopted rcl as their user API. This user API replaces the last year rclc which was outdated and not maintained by OSRF. Regarding rclc, OSRF has provided permission to the micro-ROS project to take over it and develop micro-ROS specific modules in it. The first module that has been added to this new rclc approach is RCL-Executors from BOSCH.

Additionally, an essential package for users is micro-ROS-build. This ROS 2 package consists of a build system for micro-ROS. Allows the users to develop a micro-ROS application within a ROS 2 workspace. Other appealing packages are micro-ROS-demos and docker. micro-ROS-demos as it names stands are a set of demo applications using micro-ROS. Docker, its a repository providing a docker infrastructure for ease the first approach to micro-ROS, providing ready to use environments.

2 Acronyms and keywords

Term	Definition
OSRF	Open Source Robotics Foundation
RCLC	ROS Client Library for the C language.
RCLCPP	ROS Client Library for the Cpp language.
RCLPY	ROS Client Library for the pyhton language.
ROS2	Robot Operating System

3 Overview to Results

This document provides links to the released software and documentation for deliverable D4.2 *Micro-ROS client library Software Release Y2* of Task 4.2 *micro-ROS client library* (*urcl*).

4 Links to Software Repositories

The Micro-ROS client library package is a set of repositories:

LET-Executor: This repository contains real-time executors implementation.

- Git repository: https://github.com/micro-ROS/rclc
- Branch name: new_api_and_LET_executor
- Latest commit: 2103d7c

micro-ROS-demos: This repository has examples using rcl to get started quickly

- Git repository: https://github.com/micro-ROS/micro-ROS-demos

Branch	Latest commit	ROS 2 version
master	a8cff90	crystal
feature/dashing_migration	7d05075	dashing

micro-ROS-build: Micro ROS build-system repository. Using this repository, users can easily create and cross-compile applications for reference hardware.

- Git repository: https://github.com/micro-ROS/micro-ros-build

Branch	Latest commit	ROS 2 version
crystal	2138d50	crystal
dashing	b178c62	dashing
feature/multiboard	0f3ae1c	dashing + FreeRTOS

docker: This repository holds a set of docker files which generate docker images with ready to use environments.

- Git repository: https://github.com/micro-ROS/docker

Branch Name	Latest commit	ROS 2 version
crystal	3b800b2	crystal
dashing	37e28cf	dashing

5 Annex 1: GitHub micro-ROS website

Content of https://micro-ros.github.io/docs/concepts/client_library/ from 19th December 2019.

The client library provides the micro-ROS API for the user code, i.e. for application-level micro-ROS nodes. The overall goal is to provide all relevant, major ROS 2 concepts in suitable implementation for microcontrollers. Where possible, API compatibility with ROS 2 shall be achieved for ease of portability.

In this undertaking, to minimize the long-term maintenance cost, we strive to use existing data structures and algorithms from the ROS 2 stack or to bring necessary changes in the mainline stack. This raises a lot of question regarding the applicability of existing ROS 2 layers on microcontrollers in terms of runtime efficiency, portability to different RTOS, dynamic memory management, etc.

5.1 Two-layered API

While C is still the dominating programming language for microcontrollers, there is a clear trend towards the use of higher-level languages, in particular C++. This trend is also driven by modern microcontrollers featuring several hundred kilobytes or even few megabytes of RAM.

Therefore, we plan to offer/support two APIs:

1. A **C API** based on the ROS 2 Support Client Library (rcl), enriched with modular packages for execution management, diagnostics, parameters, ...
2. A **C++ API** based on the ROS 2 rclcpp, which at first requires analyzing the fitness of rclcpp for use on microcontrollers, in particular regarding memory and CPU consumption as well as dynamic memory management.

The basis of discussion for this decision is documented in a dedicated decision paper.

5.2 Advanced Concepts

Advanced concepts developed in the context of the client library are documented separately. In general, these concepts are developed for the standard rclcpp first, before implementing a tailored C version. These are:

- Real-Time Executor
- System Modes
- Embedded Transform (TF)

6 Annex 2: Decision paper

Content of from 19th December 2019.

This document served as decision template for the design and implementation of the micro-ROS client library in March 2019. We discuss different options and existing starting points for this undertaking, decision criteria and analysis results regarding the existing assets.

6.1 Assumptions

From the past discussions and developments in 2018, we assume that a rosserial- or ros2arduino-like approach is not an option in the context of the ROS 2 Embedded SIG and the EU project OFERA, but that we strive for a solution based on rmw and rcl.

6.2 Major Options

1. rclc: Implement a new client library in the C programming language from scratch or from the few existing, unmaintained lines of code at https://github.com/ros2/rclc/.
2. rclcplus: Implement a new client library from scratch featuring basic C++ mechanisms (such as templates), but not requiring a full-fledged libstdc++ (but some very basic subset only).
3. rclcpp: Modify the rclcpp to be usable on MCUs and NuttX using an (almost) full-fledged libstdc++

6.3 Decision Criteria

- Runtime efficiency
  - Memory consumption
  - Heap fragmentation
  - CPU consumption
  - Flash footprint
- Supported programming concepts
  - Plain C or some C++
  - Support of dynamic memory management
  - Abstraction level
- Portability of user code
  - Supported ROS 2 API concepts
  - Accordance with rclcpp API
- Portability to other RTOS and MCUs
  - Dependencies on other libraries
  - Requirements to compilers
- Long-term maintenance
  - Portion of new code outside of rmw-rcl-rclcpp stack
  - Long-term commitment given by third party
- Development effort
  - Action items for implementation/port to NuttX
  - Potential action items for port to other RTOS and HW platforms
- Target users (“Clients”)
  - ROS 2 users with high-level abstraction and ROS 2 concepts in mind.
  - Embedded developers with no high level abstractions requirement.

6.4 Requirements to micro-ROS Client Library

In the EU project OFERA, a list of high-level requirements to the whole micro-ROS stack including the client library has been compiled in the Deliverable D1.7 Reference Scenarios and Technical System Requirements Definition. Import requirements immediately linked to the client library are:

- ROS 2 lifecycle: micro-ROS nodes should support the node lifecycle defined for ROS 2 nodes.
- Dynamic component parameters: micro-ROS shall provide mechanisms for dynamic management of component parameters, compatible with ROS mechanisms.
- Time precision: Clock synchronization between main micro-processor and MCU should be precise, with precision not less than 1ms.
- No-copy: Communication between nodes on the same MCU should be effective (no-copy).
- Memory usage: Relevant micro-ROS components (serialization, diagnostics, runtime configuration, RTOS abstractions, ...) shall fit on MCU with 192kB SRAM, together with existing application software.
- ROS standards: Compliance with ROS standards.
- Transferable: Moving a standard ROS 2 node to micro-ROS or the other way around should be straightforward and documented.

6.5 Technical Background and Open Questions

6.5.1 rclcpp Library

The rclcpp library features all ROS 2 concepts – including parameters and lifecycle node – and is maintained actively by the OSRF. Use of rclcpp would give best conditions for porting of ROS 2 user-code to micro-ROS.

Questions:

- Rclcpp is optimized for dynamic creation/destroying of subscriptions, publishers, timers, etc.
  - How can a static variant be implemented?
  - Is it possible to separate between an initialization and a run phase?
- Rclcpp comes with a fine-grained, complex structure of interfaces and data types. For example, a node is composed from eight interfaces.
  - How much CPU and memory overhead is caused by this architecture?
  - Does this architecture even allow to configure different static variants?
- Extensive use of advanced C++ concepts and dynamic memory management (std::vector, std::unique_lock, std::atomic_bool, std::shared_ptr)
  - Is it possible to provide abstractions and substitute types for an RTOS like NuttX?
- How much memory does rclcpp consume at runtime?

6.5.2 rclc

Only very basic concepts (node, subscription and publisher) implemented so far. Whole library has just 500 LoC in sum.

Has OSRF plans for it in the midterm? They have mentioned it on the roadmap, ROS 2 roadmap.

6.6 Dynamic Memory Management in ROS 2

Dynamic memory allocation and deallocation fragments heap, which causes indefinite computing times (for those operations) and may cause unpredictable crashes.

rmw and rcl make intensive use of dynamic memory management. The PIMPL technique is used on both layers, so the types being allocated on the heap are even not visible at the API level (i.e. not defined in the provided header files).

rcutils defines allocator struct (rcutils_allocator_t) and helper-functions (allocator.h/allocator.c) to pass own allocator to rcl functions. At many places, rcl calls rcutils_get_default_allocator, which probably requires some (minor) refactoring to allow consistent use of custom allocators throughout whole rcl and rmw.

For the rmw layer, an 'API for pre-allocating messages in the middleware' is currently discussed.

A list of issues on dynamic memory management and real-time is also discussed in the context of Autoware at AutowareAuto/AutowareAuto#65, but without the specific requirements by microcontrollers.

Questions:

- Is it possible to avoid a large heap at all by providing tiny heaps for each concept (node, subscription, publisher) on the corresponding data types on top-most layer? Can each allocation on the lower layers be clearly assigned/related to one instance on the top-most layer?
- Is a two-phase approach – allow dynamic allocation in some initialization phase but not in a later run phase – possible? How much effort is it to implement? Would such an approach be acceptable for some safety-certified implementation?

As a first experiment, we implemented a simple node and subscriber directly against the rcl in the C programming language and counted the allocations and frees. The code can be found at https://github.com/micro-ROS/micro-ROS_experiments/tree/experiment/measure_allocations/rcl_int32_subscriber.

In detail, we counted the calls of the standard C memory functions (malloc, realloc, free) and the calls of the functions of the default allocator (which uses the standard C memory functions) in rcutils/allocator.c

With the Micro XRCE-DDS middleware, we obtained the following numbers:

	STANDARD C MEMORY FUNC.			RCUTILS/ALLOCATOR.C FUNCTIONS			
CODE LINES	MALLOC	REALLOC	FREE	MALLOC	CALLOC	REALLOC	FREE
start	0	0	0	0	0	0	0
zero init options creation	0	0	0	0	0	0	0
init options init	1	0	0	1	0	0	0
zero context creation	1	0	0	1	0	0	0
rcl init	11	0	4	9	2	0	4
UDP mode => ip: 127.0.0.1 - port: 8888							
node init	19	0	4	17	2	0	4
subscription init	26	0	7	24	2	0	7
wait-set init	28	2	7	26	2	2	7
subscription added to wait-set	28	2	7	26	2	2	7
rcl_wait returned with timeout	28	2	7	26	2	2	7
rcl_wait returned with message	28	2	7	26	2	2	7
Fetched message by rcl_take	28	2	7	26	2	2	7
Message data is 159							
Printed message to std out	28	2	7	26	2	2	7
Finalized subscription	28	2	11	26	2	2	11
Finalized node	28	2	18	26	2	2	18

Figure 1: Allocations with Micro XRCE-DDS

For curiosity, the numbers for Fast-RTPS:

	STANDARD C MEMORY FUNC.			RCUTILS/ALLOCATOR.C FUNCTIONS			
CODE LINES	MALLOC	REALLOC	FREE	MALLOC	CALLOC	REALLOC	FREE
start	0	0	0	0	0	0	0
zero init options creation	0	0	0	0	0	0	0
init options init	697	0	326	1	0	0	0
zero context creation	697	0	326	1	0	0	0
rcl init	793	0	416	9	2	0	4
node init	3838	0	787	16	2	0	4
subscription init	4483	0	1059	22	2	0	7
wait-set init	4486	2	1059	25	2	2	7
subscription added to wait-set	4486	2	1059	25	2	2	7
rcl_wait returned with timeout	4486	2	1059	25	2	2	7

Figure 2: Allocations with Fast-RTPS

6.6.1 Current work on ROS 2

Currently there is an active interest in making ROS 2 a real "real time" platform. These interest have strive to a set of developments regarding the amount of dynamic memory used along the full stack. From eProsima side they are making a big effort changing Fast RTPS dynamic memory to a static system. This eProsima approach aligns with the idea of two-steps: 1) reserve all memory needed and then 2) work with the pre-allocated memory and avoid new allocations.

To develop this mechanism currently there are to main changes, all regarding STL containers:

- Use of https://github.com/foonathan/memory system, where similar to the point previously listed in this section questions, there are heaps holding sets of container elements. Then, all the operations on the containers are done without relaying in new allocations.

- Own vector specialization using two steps, a first reserve of the memory and then make some checks on the traditional API. https://github.com/eProsima/Fast-RTPS/pull/386
- Reuse of container entities.

The duality using foonathan and own vector is due to the fact that foonathan implementation is great for node based containers but not as good for continuous memory ones. A design document of Fast RTPS approach: https://github.com/eProsima/Fast-RTPS/issues/344

6.7 Embedded C++ and the C++ library

Libstdc++ makes use of dynamic memory allocation and provides features which may not be available on microcontrollers, i.e. not portable to relevant RTOS. Also, its resource consumption (in particular code size) might be relevant. Nevertheless, C++ may be used on microcontrollers.

First, it is possible to use a subset only, which does not require libstdc++ at all. A tiny demo with rclc is provided at https://github.com/micro-ROS/rclc/blob/example/c_with_templates/rclc/include/rclc/rclc.h#L24 and https://github.com/micro-ROS/micro-ROS_experiments/blob/experiment/c_with_templates/c_with_templates/main.cpp.

Further links:

- How to write a C++ program without libstdc++ (http://ptspts.blogspot.com)
- g++ without libstdc++ – can it be done? (https://stackoverflow.com/questions/3714167/)
- Bit Bashing: C++ On Embedded Systems discusses different language features of C++ and provides guidelines which to enable and which to disable for embedded programming.

6.7.1 Libcxx

During early stages of NuttX support, eProsima did a small POC using libcxx support provided by NuttX. http://nuttx.org/doku.php?id=wiki:nshhowtos:llvm-stdcxx. The POC was done using the Assis branch of libcxx however some changes on the makefile were required to be able to compile NuttX.

Another options are stripped-down libstdc++ variants with reduced feature sets optimized for embedded applications:

- https://github.com/arobenko/embxx
- https://cxx.uclibc.org/
- https://www.etlcpp.com/
- http://libmicxx.sourceforge.net/

Further links:

- Michael Caisse on 'Modern C++ in an Embedded World' at C++ Now 2018
- Bare Metal C++ – note that I (Ingo Lütkebohle) am not advocating for bare metal (rather for NuttX), but it's interesting to see what's possible

6.7.2 C++ abstractions with platform dependencies

Some of the Cortex-MX we use does not have support for atomic operations on 64bits atomic variables:

- https://stackoverflow.com/questions/35776372/atomic-int64-t-on-arm-cortex-m3#35777259
- https://answers.launchpad.net/gcc-arm-embedded/+question/616213

GCC implementation:

- https://gcc.gnu.org/wiki/Atomic/GCCMM?action=AttachFile&do=view&target=libatomic.c

We have work around this issue implementing the atomic operations on 64Bits as regular memory read-/writes.

A discrimination of the usage of the workaround is still to be done. This should be used ONLY on those architectures not supporting that kind of atomic operations.

See the modified rcl version at https://github.com/micro-ROS/rcl/commit/cdb0cca50d49c5b5576bf88d2bb7a1d57ae

6.8 Decision in the OFERA Project

- In a face-to-face meeting in Bucharest in March 2019, the partners from the OFERA project decided to take a double-tracked approach as follows:

1. Use rcl as C-based Client Library for micro-ROS by enriching it with small, modular libraries for parameters, graph, logging, clock, timers, execution management, lifecycle and system modes, TF, diagnostics, and power management.
2. Analyze fitness of rclcpp for use on microcontrollers, in particular regarding memory and CPU consumption as well as dynamic memory management.

In this meeting, we decided explicitly against a separate client library in the style of rclc.

6.8.1 Responsibilities for extensions to rcl

Parameters (eProsima)

- Optimized implementation planned, where MCU client queries agent specifically for parameter values rather than all values being sent to the node on the MCU

Graph (eProsima)

- Similar to parameters

Logging (eProsima)

- Optimized implementation for MCU

Time / Clock and Timers (eProsima, Bosch, Acutronic Robotics)

- Bosch will analyze rcl time and clock interface
- Synchronization with microprocessor – message types already available in Micro XRCE-DDS
- Adapter for RTOS required – part of abstraction layer

Executor (Bosch)

- Very simple mechanism will be developed in the micro-ROS Turtlebot demo until August 2019.

Lifecycle / System modes (Bosch)

- To be developed in a second step in the micro-ROS Turtlebot demo.

TF (Bosch)

- To be developed in a third step in the micro-ROS Turtlebot demo.

Diagnostics (Bosch)

- To be developed in the second step in the micro-ROS Turtlebot demo.
- Liveliness of node: Introduce mechanism in Micro XRCE-DDS similar to standard DDS?
- Make PR for rmw extension with abstract interface to be informed about liveliness of other nodes?!

Power management (Acutronic Robotics)

- Highly dependent on RTOS, it would require implementation for each one

7 Annex 3: micro-ROS-build

Content of from 19th December 2019.

7.1 Installation and Building

This package contains submodules. Please clone it using

```
git clone --recursive ...
```

Building this package requires that you have ROS 2 Crystal installed first.

7.2 micro_ros_setup

Support scripts for creating Micro-ROS agent and client workspaces, and cleanly doing cross-compilation.

See micro_ros_setup README for usage instructions.

7.3 micro_ros_cmake

This is a Work-in-Progress package that aims to add CMake macros and vendor packages for building micro-ROS. This should make it even easier to create applications.

See micro_ros_cmake README for more info.

7.4 Purpose of the project

The software is not ready for production use. It has neither been developed nor tested for a specific use case. However, the license conditions of the applicable Open Source licenses allow you to adapt the software to your needs. Before using it in a safety relevant setting, make sure that the software fulfills your requirements and adjust it according to any applicable safety standards (e.g. ISO 26262).

7.5 License

micro-ros-build is open-sourced under the Apache-2.0 license. See the LICENSE file for details.

micro-ros-build does not include other open source components, but uses some at compile time. See the file 3rd-party-licenses.txt for a detailed description.

7.6 Known issues/limitations

- There are currently sometimes compile issues when building the firmware for the first time. When building it a second time, this disappear. It it not known why this happens, we're investigating it.

8 Annex 4: Docker

Content of from 19th December 2019.

This repository contains docker-related material to setup, configure and develop with micro-ROS These set of Dockerfiles provide ready-to-use environments and applications. The Docker images are available in dockerhub.

Avaiable images are listed below:

Image	Description	Status
base	Base image with ROSDISTRO installation + micro-ROS specific build system. Used as base of any other micro-ROS images	docker build automated
micro-ros-agent	Image containing a pre-compiled micro-ROS-Agent, ready to use as standalone application	docker build automated
micro-ros-demos	Contains precompiled micro-ROS-demos, ready to use to view micro-ROS funcionality	docker build automated
micro-ros-firmware	Contains a firmware ws ready to configure and build micro-ROS	docker build automated
micro-ros-olimex-nuttx	Contains a ready to flash example for Olimex STM32 E407	docker build automated

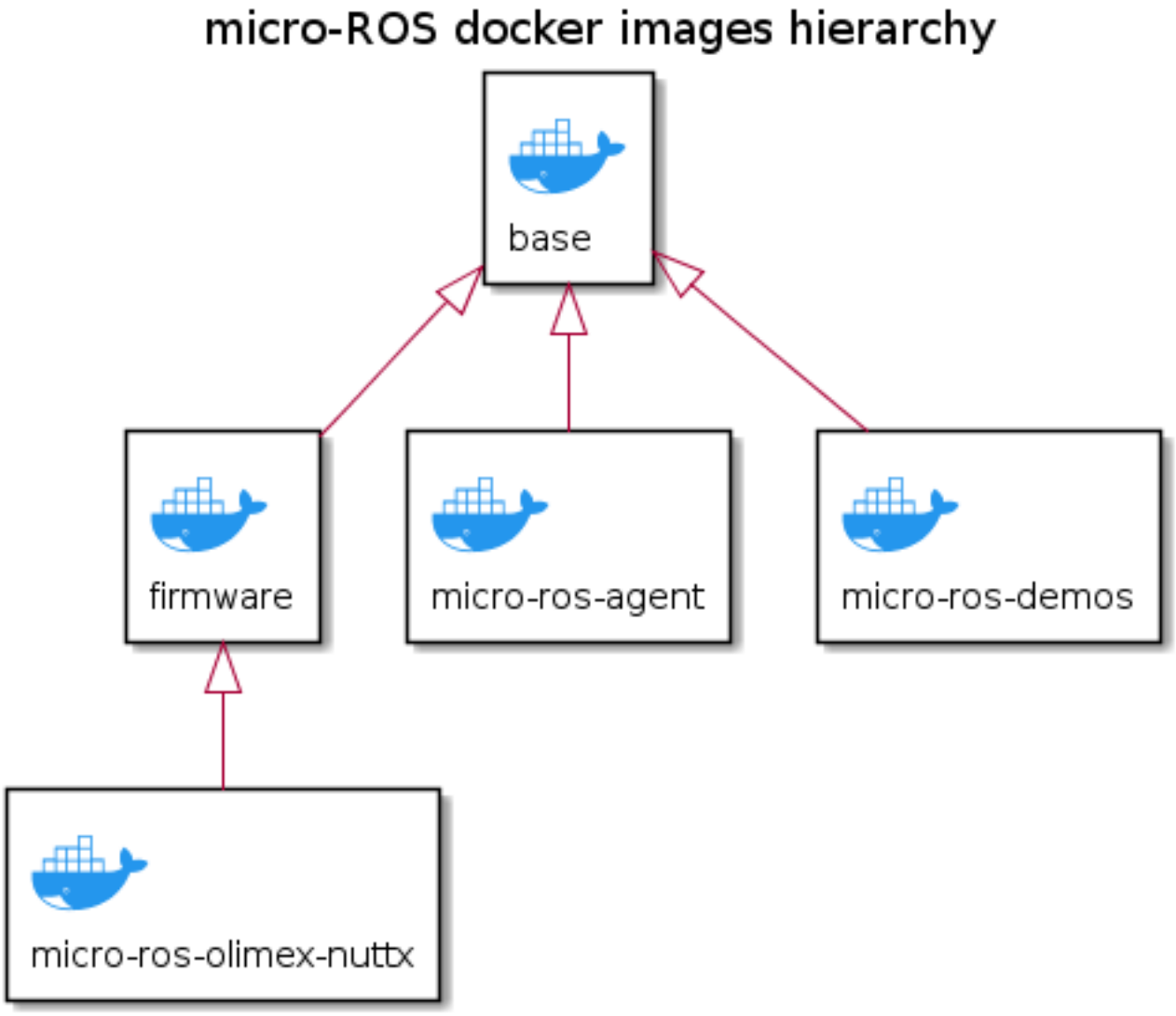

Figure 3: Image hierarchy

8.1 Pre-requisite

You need to have docker in your system. For installing docker, refer to the official documentation at https://www.docker.com/.

8.2 Usage

To get an image, you use `docker pull` command:

- e.g. `docker pull microros/base`

You can select the tag to use appending `:tag` to the image name

- e.g. `docker pull microros/base:dashing`

Once you have the image locally, to start it use `docker run`

- e.g. `docker run -it microros/micro-ros-agent`

`-it` allocates a pseudo-TTY for you and keeps stdin listening. Another used command is `-v` to map local files with docker container ones. `-v` is useful in case you may want to flash boards from within a Docker container.

8.2.0.1 base image

It is the base for the rest of the containers. It contains the necessary micro-ROS setup tools and dependencies. From this image, you can start any development targeting micro-ROS.

8.2.0.2 firmware image

This image as the base one, it is used as a starting point to those images using firmware. This image contains a firmware workspace setup and dependencies.

8.2.0.3 micro-ros-agent

This image purpose serves as a stand-alone application. It includes installation of the ROS 2 version selected by the tag along with a micro-ROS-Agent. The entry point of this image is directly the micro-ROS-Agent, so upon execution of `docker run` you will be facing micro-ROS-Agent command line input.

- e.g. `docker run -it --net=host microros/micro-ros-agent udp 9999`

Will start micro-ROS-Agent listening UDP messages on port 9999

8.2.0.4 micro-ros-demos

micro-ros-demos is one of the example images. With this image, you can launch example applications using micro-ROS (Compiled for Linux boxes) This image entry point has a ROS 2 environment set up with micro-ROS examples. You can run regular ros2 tool to launch the examples.

- eg: `docker run -it --net=hot microros/micro-ros-demos ros2`

The currently avaiable examples are:

- complex_msg_publisher_c & complex_msg_publisher_cpp
- complex_msg_subscriber_c & complex_msg_subscriber_cpp
- int32_publisher_c & int32_publisher_cpp
- int32_subscriber_c & int32_subscriber_cpp
- string_publisher_c & string_publisher_cpp
- string_subscriber_c & string_subscriber_cpp
- rad0_actuator_c, rad0_altitude_sensor_c, rad0_control_cpp & rad0_display_c

8.2.0.5 micro-ros-olimex-nuttx

This image provides you with a ready-to-flash firmware for Olimex-stm32-e407 with demos embedded on it. To flash your device you need to map your host machine devices to the Docker container

- e.g. `docker run -it --privileged -v /dev/bus/usb:/dev/bus/usb microros/micro-ros-olimex-nutt`

Once inside the container you can flash the board runing `scripts/flash.sh` from `firmware/NuttX` directory.

There you can find a publisher and a subscriber examples. Both examples use serial transport to communicate with a micro-ROS-Agent, so you should start one with the same transport (You can use the micro-ros-agent imagen to do so). Once a client-agent communication is stablished you can use `ros2` tools to view the publications from the Olimex or to publish messages to it.

8.3 Automated builds

These Docker files are used for automatically create images on Docker Hub. These builds are tagged with the ROS 2 version they will be compatible with: e.g. crystal, dashing... The latest tag will always be the latest release of ROS 2.

These automatic builds has direct relationship with the content of the micro-ROS repositories:

Image	Triggers
base	https://github.com/micro-ROS/micro-ROS-build
firmware	https://github.com/micro-ROS/apps https://github.com/micro-ROS/NuttX
micro-ros-agent	
micro-ros-demos	https://github.com/micro-ROS/micro-ROS-demos
micro-ros-olimex-nuttx	

Apart from GitHub repositories changes, a build could be triggered whenever the base image is updated on Docker Hub. Base images are specified in the FROM: directive in the Dockerfile.