D4.16

Micro-ROS Client library Software Release Y4

Grant agreement no.	780785
Project acronym	OFERA (micro-ROS)
Project full title	Open Framework for Embedded Robot Applications
Deliverable number	D4.16
Deliverable name	Micro-ROS Client library
Date	December 2021
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 D4.16 *Micro-ROS client library Software Release Y4* of the Task 4.2 *micro-ROS client library (urcl).*

Contents

1 Summary

By virtue of the decision of the OFERA consortium to follow a modular approach for the client library, 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. micro-ROS has adopted rcl plus rclc as their user API.

Regarding rclc, the OSRF (now Open Robotics) 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. During this years, rclc has become a more relevant API with features such as services, clients, parameters servers, ...

Additionally, an essential package for users is `micro_ros_setup`. This ROS 2 package consists of a build system for micro-ROS. It allows 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` is a repository providing a docker infrastructure for easing 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 D4.16 *Micro-ROS client library Software Release Y4* of Task 4.2 *micro-ROS client library (urcl).*

3.1 Links to Software Repositories

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

RCLC: This repository contains RCLC user layer and the real-time executor implementation.

- Git repository: https://github.com/ros2/rclc

Branch	Latest commit	ROS 2 version
foxy	2f129f4	foxy
galactic	85a914b	galactic
master	f8de341	rolling

micro-ROS-demos: This repository has examples using rcl and rclc to easily get started with the task of developing a micro-ROS application.

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

Branch	Latest commit	ROS 2 version
foxy	c67c381	foxy
galactic	4ab3395	galactic
main	02abbe7	rolling

micro_ros_setup: 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_setup.git

Branch	Latest commit	ROS 2 version
main	0c61c67	rolling
galactic	92599b1	galactic
foxy	e920c89	foxy

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
foxy	bd56763	foxy
galactic	102b9f1	galactic
main	a0964e8	rolling

4 Annex 1: GitHub micro-ROS website

Content of https://micro.ros.org/docs/concepts/client_library from 26th November 2021.

4.1 Introduction to Client Library

The Client Library provides the micro-ROS API for the user code, i.e., for application-level micro-ROS nodes. The goal is to provide all relevant, major ROS 2 concepts in an optimized implementation for microcontrollers. At the same time, we strive to make the API as compatible as possible to standard ROS 2, to facilitate porting of code.

To minimize the long-term maintenance cost, we use existing data structures and algorithms from the ROS 2 stack and bring necessary changes in the mainline stack as far as possible. That's why the micro-ROS client library is built up from standard ROS 2 Client Support Library (rcl) and a new ROS 2 Client Library package (rclc). Together, as depicted below, rcl + rclc form a feature-complete client library in the C programming language.

Important features and properties:

- Use of rcl data structures when possible to avoid runtime overhead by wrappers.
- Convenience functions for common tasks (e.g., creation of a publisher, finalization of a subscription) provided by rclc.
- Dedicated Executor for fine-grained control over triggering and processing order of callbacks.
- Specialized implementations for graphs, lifecycle nodes, diagnostics, etc.

4.2 Feature List

The micro-ROS Client Library, formed by standard ROS 2 Client Support Library (rcl) and a new ROS 2 Client Library package (rclc), is going to feature all major ROS concepts such as

- Nodes
- Publishers/subscriptions
- Services/clients
- Graph
- Executor
- Lifecycle
- Parameters

Most features are already available in the Foxy release. Please see our Feature Overview page for details on the status. To learn developing your own application nodes with rcl + rclc, please head to the corresponding programming tutorial.

5 Annex 2: Decision paper

Content of https://micro.ros.org/download/client_library_decision_paper_2019.pdf from 26th November 2021.

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.

5.1 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.

5.2 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++

5.3 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.

5.4 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.

5.5 Technical Background and Open Questions

5.5.1 rclcpp Library

The rclcpp library features all ROS 2 concepts – including parameters and lifecycle node – and is maintained actively by the Open Robotics Foundation. 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?

5.5.2 rclc

While only very basic concepts (node, subscription and publisher) had been implemented in this library up to last year (see D4.2 for reference), this year the OFERA consortium partner BOSCH has been devoting an intense development activity to migrate high-level concepts such as executors, system modes, services and more to this API. The executor is now feature complete; i.e. supporting all event types as the ROS 2 rclcpp executor (subscriptions, timers, services, clients, guard conditions).

5.6 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, in past years we implemented a simple node and subscriber directly against the rcl in the C programming language and counted the allocations and frees. 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

More information regarding the dynamic memory consumption in micro-ROS can be found in a detailed memory profiling carried out during 2020 by EPROS, and published on the micro-ROS website at this page.

5.6.1 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

5.7 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. It is even possible to use a subset only, which does not require libstdc++ at all.

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.

5.7.1 Libcxx

During early stages of NuttX support, eProsima did a small POC using libcxx support provided by NuttX. LLVM libc++ for NuttX. 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

5.7.2 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/cdb0cca50d49c5b5576bf88d2bb7a1d57ae1e00b

5.8 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, it was explicitly decided against a separate client library in the style of rclc.

During 2020, however, the rclc has gained weigth and several high-level concepts have been migrated to this layer, and more are to come (such as graphs). At the same time, the idea of employing rclcpp functionalities in micro-ROS has been progressively abandoned.

5.8.1 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)

- supports all handle types as in rclcpp executor (subscriptions, timers, services, clients, guard conditions)
- deterministic execution concepts
- dynamic memory allocation only at startup

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

6 Annex 3: micro_ros_setup

Content of https://github.com/micro-ROS/micro_ros_setup/blob/galactic/README.md from 26th November 2021.

This ROS 2 package is the entry point for building micro-ROS apps for different embedded platforms.

6.1 Supported platforms

This package is the **official build system for micro-ROS**. It provides tools and utils to cross-compile micro-ROS with just the common ROS 2 tools for these platforms:

RTOS	Platform	Version	Example
Azure RTOS	Renesas RA6M5	Renesas e2 studio	`renesas_ra ra6m5`
FreeRTOS	Renesas RA6M5	Renesas e2 studio	`renesas_ra ra6m5`
Bare metal	Renesas RA6M5	Renesas e2 studio	`renesas_ra ra6m5`
FreeRTOS	Olimex STM32-E407	STM32CubeMX latest	`freertos olimex-stm32-e407`
FreeRTOS	ST Nucleo F446RE 1	STM32CubeMX latest	`freertos nucleo_f446re`
FreeRTOS	ST Nucleo F446ZE 1	STM32CubeMX latest	`freertos nucleo_f446ze`
FreeRTOS	ST Nucleo F746ZG 1	STM32CubeMX latest	`freertos nucleo_f746zg`
FreeRTOS	ST Nucleo F767ZI 1	STM32CubeMX latest	`freertos nucleo_f767zi`
FreeRTOS	Espressif ESP32	v8.2.0	`freertos esp32`
FreeRTOS	Crazyflie 2.1	v10.2.1 - CF 2020.06	`freertos crazyflie21`
Zephyr	Olimex STM32-E407	v2.6.0	`zephyr olimex-stm32-e407`
Zephyr	ST B-L475E-IOT01A	v2.6.0	`zephyr discovery_l475_iot1`
Zephyr	ST Nucleo H743ZI 1	v2.6.0	`zephyr nucleo_h743zi`
Zephyr	Zephyr emulator	v2.6.0	`zephyr host`
Mbed	ST B-L475E-IOT01A	v6.6	`mbed disco_l475vg_iot01a`
-	Static library 3	-	`generate_lib`
Linux	*Host 2*	Ubuntu 18.04/20.04	`host`
Android	AOSP 4	Latest	`android generic`

1 Community supported, may have lack of official support

2 Support for compiling apps in a native Linux host for testing and debugging

3 a valid CMake toolchain with custom crosscompilation definition is required

4 Community supported, may have lack of official support

6.1.1 Standalone build system tools

micro-ROS also offers some other ways to crosscompile it for different platforms. These other options are secondary tools and may not have full support for all features. Currently micro-ROS is also available as:

- a standalone **micro-ROS component for Renesas e2 studio and RA6M5**: this package enables the integration of micro-ROS in Renesas e2 studio and RA6M5 MCU family.
- a standalone **micro-ROS component for ESP-IDF**: this package enables the integration of micro-ROS in any Espressif ESP32 IDF project.
- a standalone **micro-ROS module for Zephyr RTOS**: this package enables the integration of micro-ROS in any Zephyr RTOS workspace.
- a standalone **micro-ROS module for Mbed RTOS**: this package enables the integration of micro-ROS in any Mbed RTOS workspace.

- a standalone **micro-ROS module for NuttX RTOS**: this package enables the integration of micro-ROS in any NuttX RTOS workspace.
- a standalone **micro-ROS module for Microsoft Azure RTOS**: this package enables the integration of micro-ROS in a Microsoft Azure RTOS workspace.
- a set of **micro-ROS utils for STM32CubeMX and STM32CubeIDE**: this package enables the integration of micro-ROS in STM32CubeMX and STM32CubeIDE.
- a precompiled set of **Arduino IDE libraries**: this package enables the integration of micro-ROS in the Arduino IDE for some hardware platforms.
- a precompiled set of **Raspberry Pi Pico SDK libraries**: this package enables the integration of micro-ROS in the Raspberry Pi Pico SDK.

6.2 Dependencies

This package targets the **ROS 2** installation. ROS 2 supported distributions are:

ROS 2 Distro	State	Branch
Crystal	EOL	`crystal`
Dashing	EOL	`dashing`
Foxy	Supported	`foxy`
Galactic	Supported	`galactic`
Rolling	Supported	`main`

Some other prerequisites needed for building a firmware using this package are:

```
sudo apt install python3-rosdep
```

Building for Android needs Latest Android NDK to be installed and the following environment variables to be set: - `ANDROID_ABI`: CPU variant, refer here for details. - `ANDROID_NATIVE_API_LEVEL`: Android platform version, refer here for details. - `ANDROID_NDK`: root path of the installed NDK.

6.3 Quick start

Download here the micro-ROS docker image that contains a pre-installed client and agent as well as some compiled examples.

6.4 Building

Create a ROS 2 workspace and build this package for a given ROS 2 distro (see table above):

```
source /opt/ros/$ROS_DISTRO/setup.bash

mkdir uros_ws && cd uros_ws

git clone -b main https://github.com/micro-ROS/micro_ros_setup.git
   src/micro_ros_setup
```

```
rosdep update && rosdep install --from-path src --ignore-src -y

colcon build

source install/local_setup.bash
```

Once the package is built, the firmware scripts are ready to run.

You can find tutorials for moving your first steps with micro-ROS on an RTOS in the micro-ROS webpage.

6.4.1 Creating micro-ROS firmware

Using the `create_firmware_ws.sh [RTOS] [Platform]` command, a firmware folder will be created with the required code for building a micro-ROS app. For example, for our reference platform, the invocation is:

```
# Creating a FreeRTOS + micro-ROS firmware workspace
ros2 run micro_ros_setup create_firmware_ws.sh freertos olimex-stm32-e407

# Creating a Zephyr + micro-ROS firmware workspace
ros2 run micro_ros_setup create_firmware_ws.sh zephyr olimex-stm32-e407
```

6.4.2 Configuring micro-ROS firmware

By running `configure_firmware.sh` command the installed firmware is configured and modified in a pre-build step. This command will show its usage if parameters are not provided:

```
ros2 run micro_ros_setup configure_firmware.sh [configuration] [options]
```

By running this command without any argument the available demo applications and configurations will be shown.

Common options available at this configuration step are: - `--transport` or `-t`: `udp`, `serial` or any hardware specific transport label - `--dev` or `-d`: agent string descriptor in a serial-like transport (optional) - `--ip` or `-i`: agent IP in a network-like transport (optional) - `--port` or `-p`: agent port in a network-like transport (optional)

Please note that each RTOS has its configuration approach that you might use for further customization of these base configurations. Visit the micro-ROS webpage for detailed information about RTOS configuration.

In summary, the supported configurations for transports are:

	FreeRTOS	Zephyr	Mbed
Olimex STM32-E407	UART, Network	USB, UART	-
ST B-L475E-IOT01A	-	USB, UART, Network	UART
Crazyflie 2.1	Custom Radio Link	-	-

	FreeRTOS	Zephyr	Mbed
Espressif ESP32	UART, WiFI UDP	-	-
ST Nucleo F446RE 1	UART	-	-
ST Nucleo F446ZE 1	UART	-	-
ST Nucleo H743ZI 1	-	UART	-
ST Nucleo F746ZG 1	UART	UART	-
ST Nucleo F767ZI 1	UART	-	-

1 Community supported, may have lack of official support

6.4.3 Building micro-ROS firmware

By running `build_firmware.sh` the firmware is built:

```
ros2 run micro_ros_setup build_firmware.sh
```

6.4.4 Flashing micro-ROS firmware

In order to flash the target platform run `flash_firmware.sh` command. This step may need some platform-specific procedure to boot the platform in flashing mode:

```
ros2 run micro_ros_setup flash_firmware.sh
```

6.5 Building micro-ROS-Agent

Using this package is possible to install a ready to use **micro-ROS-Agent**:

```
ros2 run micro_ros_setup create_agent_ws.sh
ros2 run micro_ros_setup build_agent.sh
source install/local_setup.sh
ros2 run micro_ros_agent micro_ros_agent [parameters]
```

6.6 Contributing

As it is explained along this document, the firmware building system takes **four steps**: creating, configuring, building and flashing.

New combinations of platforms and RTOS are intended to be included in `config` folder. For example, the scripts for building a **micro-ROS** app for **Crazyflie 2.1** using **FreeRTOS** is located in `config/freertos/crazyflie21`.

This folder contains up to four scripts: - `create.sh`: gets a variable named `$FW_TARGETDIR` and installs in this path all the dependencies and code required for the firmware. - `configure.sh`: modifies and configure parameters of the installed dependencies. This step is **optional**. - `build.sh`:

builds the firmware and create a platform-specific linked binary. - `flash.sh`: flashes the binary in the target platform.

Some other required files inside the folder can be accessed from these scripts using the following paths:

```
# Files inside platform folder
$PREFIX/config/$RTOS/$PLATFORM/

# Files inside config folder
$PREFIX/config
```

6.7 Purpose of the Project

This 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.

6.8 License

This repository is open-sourced under the Apache-2.0 license. See the LICENSE file for details.

For a list of other open-source components included in ROS 2 system_modes, see the file 3rd-party-licenses.txt.

6.9 Known Issues/Limitations

There are no known limitations.

If you find issues, please report them.

7 Annex 4: Docker

Content of https://github.com/micro-ROS/docker/blob/galactic/README.md from 26th November 2021.

This repository contains Docker-related material aimed at setting up, configuring and developing a micro-ROS-based application.

This set of Dockerfiles provides ready-to-use environments to easily execute micro-ROS examples in your host machine, as well as to use the standalone micro-ROS build system. In addition, two images are provided that allow using micro-ROS as an external library, both in ESP-IDF.

The Docker images can be found at dockerhub.

7.1 Pre-requisites

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

7.2 Usage

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

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

You can select the preferred tag by appending `:tag` to the image name

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

Once you have the image locally, type `docker run` to start it. It is not mandatory, although usually useful, to launch your containers using the `--rm` and `--net=host` flags:

- e.g. `docker run -it --rm --net=host microros/micro-ros-agent:galactic`

`--rm` makes sure that the docker image will be removed after exiting. `--net=host` provides the container with the same network access as the host. `-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.

7.2.0.1 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.

7.2.0.2 micro-ros-agent

This image is meant to be used as a stand-alone application. It includes the installation of the ROS 2 version selected by the tag selected, together 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 the micro-ROS Agent command line input. Running:

- e.g. `docker run -it --net=host microros/micro-ros-agent:galactic udp4 -p 9999`

will start a micro-ROS Agent listening to UDP messages on port 9999.

7.2.0.3 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 machines). This image entry point has a ROS 2 environment set up with micro-ROS examples. You can run regular ROS 2 tools to launch the examples.

- eg: `docker run -it --net=host microros/micro-ros-demos bash`

The currently available examples are listed here.

7.2.0.4 micro_ros_static_library_builder

The `micro_ros_static_library_builder` docker image provides you with a set of include files and pre-compiled micro-ROS libraries to develop your micro-ROS application within the Arduino IDE environment. To be able to use it, use the following command to instantiate a container of this image:

- e.g. `docker run -it -v $(pwd):/arduino_project --net=host microros/micro_ros_static_lib`

Note that folders added to `extras/library_generation/extra_packages` and entries added to `extras/library_generation/extra_packages/extra_packages.repos` will be taken into account by this build system.

7.2.0.5 esp-idf-microros

This Docker image allows you to use micro-ROS as a component of the ESP-IDF build system.

To use it:

- e.g. `docker run -it --user espidf --volume="/etc/timezone:/etc/timezone:ro" -v $(pwd):/micro_ros_espidf_component -v /dev:/dev --privileged --workdir /micro_ros_espidf_component microros/esp-idf-microros:latest /bin/bash"`

Then, you can navigate to your example aplications and build it using the ESP IDF buildtool system script, `idf.py menuconfig/build/flash/monitor`.

7.3 Automated builds

These Dockerfiles are used for automatically creating images on Docker Hub. These builds are tagged with the ROS 2 version they will be compatible with: e.g. foxy, galactic, rolling... The latest tag will always correspond to the latest release of ROS 2.

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

Image	Triggers
base	https://github.com/micro-ROS/micro-ROS-build
micro-ros-agent	https://github.com/micro-ROS/micro-ROS-Agent https://github.com/eProsima/Micro-XRCE-DDS-Agent
micro-ros-demos	https://github.com/micro-ROS/micro-ROS-demos
esp-idf-microros	https://github.com/micro-ROS/micro_ros_espidf_component

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

7.4 Purpose of the Project

This 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

This repository is open-sourced under the Apache-2.0 license. See the LICENSE file for details.

For a list of other open-source components included in this repository, see the file 3rd-party-licenses.txt.

7.6 Known Issues/Limitations

There are no known limitations.

If you find issues, please report them.