BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

TOOLS FOR ENSURING INTEROPERABILITY BETWEEN ARDUINO/ESP AND ROS2

PROSTŘEDKY PRO ZAJIŠTĚNÍ INTEROPERABILITY ARDUINO/ESP A ROS2

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **MATÚŠ FABO**
AUTOR PRÁCE

SUPERVISOR **doc. Ing. VLADIMÍR JANOUŠEK, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2024

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

Zadání bakalářské práce

155181

Ústav:	Ústav inteligentních systémů (UITS)
Student:	**Fabo Matúš**
Program:	Informační technologie
Název:	**Prostředky pro zajištění interoperability Arduino/ESP a ROS2**
Kategorie:	Softwarové inženýrství
Akademický rok:	2023/24

Zadání:

1. Prostudujte problematiku robotických řídicích systémů, zaměřte se na platformu ROS2. Seznamte s prostředky pro tvorbu software pro mikrokontrolery firmy Espresif a s existujícími prostředky pro propojení těchto mikrokontrolerů middlewarem ROS2.
2. Vyberte nebo navrhněte vhodný způsob propojení a protokol pro komunikaci mezi mikrokontrolerem a řídicím počítačem, na kterém poběží ROS2 (například Raspberry Pi 4). Jako motivační a následně demonstrační aplikaci uvažujte například ESP32 s kamerou, ovládání servomotorů apod.
3. Prostředky pro komunikaci specifikované v bodu 2 navrhněte a realizujte. Navrhněte a realizujte i demonstrační aplikaci specifikovanou v bodu 2.
4. Realizovaný software otestujte a vyhodnoťte dosažené výsledky.

Literatura:
Dle pokynů vedoucího.

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce:	**Janoušek Vladimír, doc. Ing., Ph.D.**
Vedoucí ústavu:	Hanáček Petr, doc. Dr. Ing.
Datum zadání:	1.11.2023
Termín pro odevzdání:	9.5.2024
Datum schválení:	6.11.2023

Abstract

This thesis explores the integration of the low-cost ESP32 line of microcontrollers with the Robot Operating System 2 (ROS2), a widely-used framework in the field of robotics. The primary goal is to develop methods and tools that facilitate seamless integration of ESP32 microcontrollers into the ROS2 development ecosystem, leveraging existing middleware solutions provided by third parties. By focusing on the incorporation of ESP-IDF projects into ROS2 tooling, the research aims to streamline the deployment and management of ESP32-based systems within robotics applications. This approach enhances the accessibility and utility of ROS2, enabling more efficient and effective use of ESP32 microcontrollers in various robotic contexts.

Abstrakt

Táto práca skúma integráciu nízkonákladovej rady mikrokontrolérov ESP32 s Robot Operating System 2 (ROS2), populárnym frameworkom v oblasti robotiky. Primárnym cieľom je vyvinúť metódy a nástroje, ktoré uľahčia bezproblémovú integráciu mikrokontrolérov ESP32 do vývojového ekosystému ROS2 s využitím existujúcich middleware riešení poskytovaných tretími stranami. Zameraním sa na začlenenie projektov ESP-IDF do nástrojov ROS2 sa výskum zameriava na zefektívnenie nasadenia a riadenia systémov založených na ESP32 v robotických aplikáciách. Tento prístup zvyšuje dostupnosť a užitočnosť ROS2, čo umožňuje efektívnejšie využitie mikrokontrolérov ESP32 v rôznych robotických kontextoch.

Keywords

ESP-IDF, ESP32, ROS2, CMake, Interoperability, Microcontrollers, Integration, Tooling, Middleware, IoT, Robotics

Klíčová slova

ESP-IDF, ESP32, ROS2, CMake, Interoperabilita, Mikrokontroléry, Integrácia, Nástroje, Middleware, IoT, Robotika

Reference

FABO, Matúš. *TOOLS FOR ENSURING INTEROPERABILITY BETWEEN ARDUINO/ESP AND ROS2.* Brno, 2024. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor doc. Ing. Vladimír Janoušek, Ph.D.

TOOLS FOR ENSURING INTEROPERABILITY BETWEEN ARDUINO/ESP AND ROS2

Declaration

I declare that I have prepared this bachelor thesis independently under the supervision of doc. Ing. Vladimír Janoušek PhD. I have cited all literary sources, publications and other sources from which I have drawn.

. . . .
Matúš Fabo
May 8, 2024

Contents

List of Figures

Chapter 1

Introduction

The Robot Operating System 2 (ROS2) is an advanced open-source framework that provides tools, libraries, and conventions to build complex robotic applications. However, its utility can be limited by the hardware compatibility and accessibility issues, particularly with low-cost microcontrollers that are frequently used in educational settings, hobbyist projects, and prototype development.

Among these low-cost options, the ESP32 microcontroller stands out due to its processing power, wireless capabilities such as WiFi and Bluetooth. This makes it a very popular choice for IoT and simple robotic applications. But these microcontrollers are not limited only to hobbyists projects - due to its capabilities the ESP32 has the potential to become the industry standard. Despite this the integration between the two platforms is challenging. This is caused mainly by differences in their respective development environments.

Traditionally, ESP32 projects are developed using the ESP-IDF (IoT Development Framework), which is designed to utilize the full capabilities of these microcontrollers but does not natively support the structures and protocols used in ROS2. Because of this, leveraging the full capabilities of ESP32 within the ROS2 framework requires bridging communication protocols and development workflows.

This thesis aims to address these differences by developing streamlined methods and tools that allow the seamless integration of ESP32 microcontrollers into the ROS2 framework. By improving interoperability, the goal is to improve the developement of ROS2 applications. This work improves the practical deployment of mixed-technology solutions in real world applications.

Chapter 2

Robot Operating System (ROS2)

2.1 ROS

Robot Operating System (ROS) is an open-source collection of tools, libraries and conventions used for designing and building complex robotic software systems. Its main purpose is to ensure stable and reliable communication between different processesses, languages or a completly different devices. This framework also provides tools for easy interaction, debugging and testing individual components or the system as a whole.

ROS1

https://roboticsbackend.com/ros1-vs-ros2-practical-overview/

ROS1 was built from scratch by enthusiasts. This version has implemented most of the core ROS concepts. But as with most of projects written from scratch, this version became unstable as the framework evolved over the years. This paired with lacking features such as security or real-time processing led developers of ROS to rebuild it.

ROS2

ROS2 was rebuilt from scratch but it has built upon the ideas of ROS1 with added features stemming from years of experience. This version has improved on previous shortcomings such as stability and security, added features such as Quality of Service. But most importantly for us, this version has added support for embedded real-time systems.

2.2 Architecture

2.2.1 Workspace

A ROS2 workspace is a directory where ROS2 project is developed. This workspace includes various packages and metapackages configured for building and distribution. Workspaces allow for isolation of projects or integration of multiple related projects, allowing complex robotic software development.

Content

The workspace directory contains 4 subdirectories:

- **src/** holds all relevant ROS2 packages
- **build/** holds latest build artifacts
- **install/** holds latest successfully built packages
- **log/** stores all build and install output

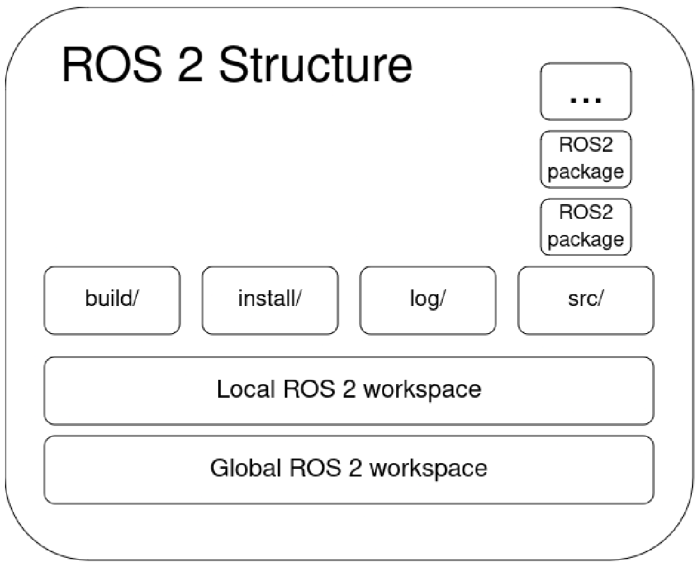


Figure 2.1: Visual representation of ROS2 environment

2.2.2 Packages

Each package is defined by its `package.xml` manifest file[3] . This file contains package description and its dependencies. ROS2 packages are the primary organizational unit of software code, and they should contain everything from executable nodes and libraries to configurations, data files, or anything else that can be logically grouped together. Each package should to provide a specific feature or functionality within the ROS2 ecosystem for ease of testing and maintenance.

2.2.3 Nodes

https://docs.ros.org/en/foxy/Tutorials/Beginner-CLI-Tools/Understanding-ROS2-Nodes/Understanding ROS2-Nodes.html

A node is a fundamental executable entity that performs some sort of computation[3]. Each node in ROS2 is designed to execute a specific task within a larger application, such as controlling a robotic arm, processing sensor data, or managing communication with other nodes. Nodes communicate with one another using one of three interfaces, depending on the requirements of the communication. This approach allows for independent opreation with collaborative functionality of the robotic system. This design allows easier debugging, testing, and maintenance, as changes to one node typically do not affect the operation of others. There can be multiple nodes per device or one node per device - the middleware layer ensures communication between nodes in both scenarios.

2.2.4 Interfaces

Interfaces are standardized templates for communication that nodes use to communicate effectively within the ecosystem. Interfaces are defined inside a package in their respective directories. As mentioned in section 2.2.3 there are three types of interfaces:

- **Topics** Subscribe-publish style of communication
- **Services** Request-response style of communication
- **Actions** Action based style of communication

These interfaces are declared using Interface Definition Language (IDL) files, which specify the data types and structures that ensure consistent communication across different programming languages and hardware platforms. By standardizing how data is exchanged, ROS2 interfaces ensure a scalable and reliable communication within the entire system.

Topic

Topics provide a way for simple communication using **messages**. A topic is designed for publishing and subscribing mechanisms where information flows between nodes continuously. A single topic can have multiple subscribers as well as multiple publishers. This is useful for aggregted messages such as logging.

A message consists of fields of primitive data types such as an integer or a string. Another messages can be used instead of primitive data types, allowing for more readable complex messages.

Messages are defined in `msg/` directory as `.msg` files.
Template for `.msg` files is:

```
# Message
<data-type> <name>
...
```

Service

Services provide a request-response interaction model, enabling one node to send a request to another and receive a response. Services are designed as server like request-response mechanism where the information flows on demand or the response depends on the request parameters.

A service consists of a request and a response separated by `--`. Request and response, like messages, consist of primitive data types or another messages.

Services are defined in `srv/` directory as `.srv` files.
Template for `.srv` files is:

```
# Request
<data-type> <name>
...

---

# Response
<data-type> <name>
...
```

Action

Actions extend the service model to handle long-running asynchronous processes, with the capability for feedback during task execution and the option for cancellation if necessary.

An action consists of a `goal` , `feedback` and `result` separated by `--`. Again, like messages, these consist of primitive data types or another messages.

A goal sent by a client to a server, then feedback that is periodically sent from the server to the client during the execution of a goal, and a result is sent back to the client when the goal is completed.

Actions are defined in `action/` directory as `.action` files.
Template for `.action` files is:

```
# Goal
<data-type> <name>
...

---

# Feedback
<data-type> <name>
...

---

# Result
<data-type> <name>
...
```

2.3 Middleware

At the heart of ROS2's communication system is the middleware layer, which provides a standardized interface for inter-process communication. This layer abstracts away the complexities of the underlying network protocols, which allows developers to focus solely on the application logic without worrying about communication. The middleware in ROS2 is modular, which means it can support different middleware implementations. However, the default and most commonly used is based on the Data Distribution Service (DDS).

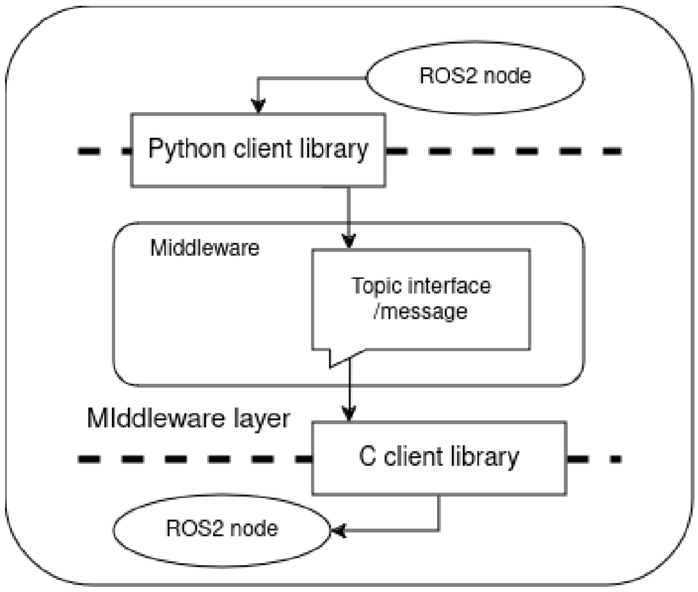


Figure 2.2: Visual representation of the ROS2 middleware

Quality of Service

Middleware in ROS2 implements Quality of Service (QoS), which crucial for ensuring stable and reliable communication. QoS settings include parameters such as **reliability** (whether messages should be delivered reliably or best-effort), **durability** (whether messages should be stored until they are taken by some subscriber) and **deadline** (maximum expected time between messages). These settings can be customized per topic, providing granular control over communication behavior based on specific requirements of each part of the system.

Node lifecycle management

The middleware manages the lifecycle of nodes within the ROS2 environment, including the creation, configuration, execution, and shutdown of nodes. This lifecycle management is essential for ensuring that resources are allocated and freed appropriately, and for maintaining the health and stability of the overall system.

Security

ROS2 middleware supports various security features to ensure secure communication across nodes. These features include encryption, authentication, and authorization mechanisms, which are implemented at the middleware level.

2.4 Build System

The ROS2 build system is designed to handle the compilation and distribution of software packages efficiently across various systems. It supports the complex dependencies of ROS2 applications and ensures that they are correctly built and integrated into the ROS ecosystem. The ROS2 build system consists of three tools: Ament, Colcon, and CMake.

CMake

CMake is used within ROS2 for its powerful cross-platform build capabilities. It generates native makefiles or project files that can be used in the compiler environment of your choice. For ROS2, CMake lists within packages define build targets, library and executable dependencies, and installation rules.

Ament

Ament is the build system and dependency management tool specifically designed for ROS2. It enhances CMake with additional features to streamline the building process. It manages package dependencies, builds processes, and environment setup through integrated tools and scripts. Ament ensures that packages are built in the correct order and that all necessary components are included at compile time. It also handles the environment configuration, setting up paths and variables that are required for packages to run correctly. Ament packages typically contain a manifest file that specifies package metadata, dependencies, and other configurations necessary for the build process.

Colcon

Colcon is a command line tool to improve the workflow of building, testing, and using multiple software packages. It extends the capabilities of both Ament and CMake, streamlining the process of building, testing, and packaging software that is developed as a part of ROS2 projects. Colcon supports parallel builds and integrates with various build systems to handle a wide array of programming languages and build environments. It also keeps track of dependencies between packages within the workspace, ensuring that lower-level libraries are compiled before the packages that depend on them.

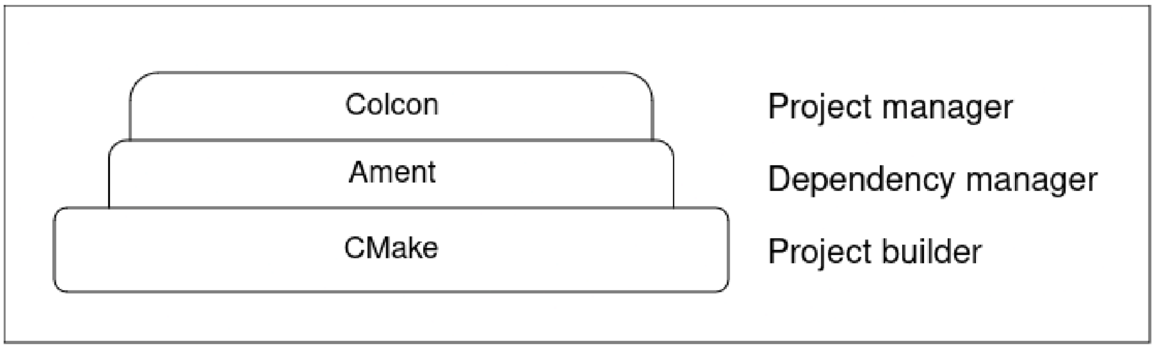


Figure 2.3: Visual representation of the CMake-Ament-Colcon build stack

Chapter 3

ESP-IDF

ESP-IDF (Espressif IoT Development Framework) is the official developement framework for their ESP SoCs. This framework provides a set of tools and libraries that simplify the development of high-performance embedded applications.

3.1 Structure

ESP-IDF is structured in a modular manner, where each feature is contained within a component. The root of the framework contains 4 directories `components/`, `examples/`, `tools/` and `docs/`, as well as global configurations and setup scripts.

- `components/` contains global set of libraries and drivers
- `examples/` contains a set of example applications that showcase most of ESP features
- `tools/` contains every tool needed for compilation, flashing or debugging an application
- `components/` contains its documentation. This documentation can be found at the [espressif's official documentation website[1]

ESP-IDF also contains a `.espressif` directory located at the home directory of your system. This direcotry contains most of the ESP-IDF toolchain such as the ESP-idf Python environment or compilers for the ESP chipsets.

3.2 Components

Components are standalone modules that encapsulate specific features such as drivers for peripheral devices, networking stacks, or utility libraries. These components are the fundamental building blocks of every application. ESP-IDF comes with a set of standard components provided by Espressif, but developers can also create custom components that suit their specific requirements.
Every component directory contains (but is not limited to):

- `CMakeLists.txt` which contains instructions for building the component
- `main/` directory (can be renamed, but requires effort) that contains its main logic
- `components/` directory (optional) that contains additional component dependencies that are not found in the global component directory
- `Kconfig` (optional) describes what configuration the components needs

Project structure

An ESP-IDF project is a direcoty that contains the main structure and source code of an executable application[1]. The project follows similar rules as other components. The root directory has to contain `CMakeLists.txt` that provides instructions for proper compilation. Additionaly, because this is the application's main structure, the directory has some additions:

- `build/` directory that holds build artifacts and the compiled application
- `sdkconfig` which holds configuration for the application as well as all its components dependencies.

The `sdkconfg` file is generated at build time with default configuration. Changes to the application configuration can be done with the `idf.py menuconfig` command. This menuconfig is based on linux' menuconfig - a configuration tool which utilizes the [Kconfig language](https://docs.kernel.org/kbuild/kconfig-language.html)

3.3 Build System

The ESP-IDF build system is based primarily on CMake, which can handle complex build behaviour and dependencies. THe build system is responsible for automating the configuration, compilation, and linking of the project files into a binary firmware that can be loaded onto an ESP32 device.

CMake

CMake is used within ESP-IDF as the primary build system tool that is repsonsible for the configuration and generation of build files. Cmake abstracts and automates the compilation process, managing dependencies between components, configuring the necessary compiler flags, and defining build targets. ESP-IDF enhances CMake with its own set of macros and functions that simplify common tasks such as registering new components, integrating third-party libraries, and customizing the build process for different environments.

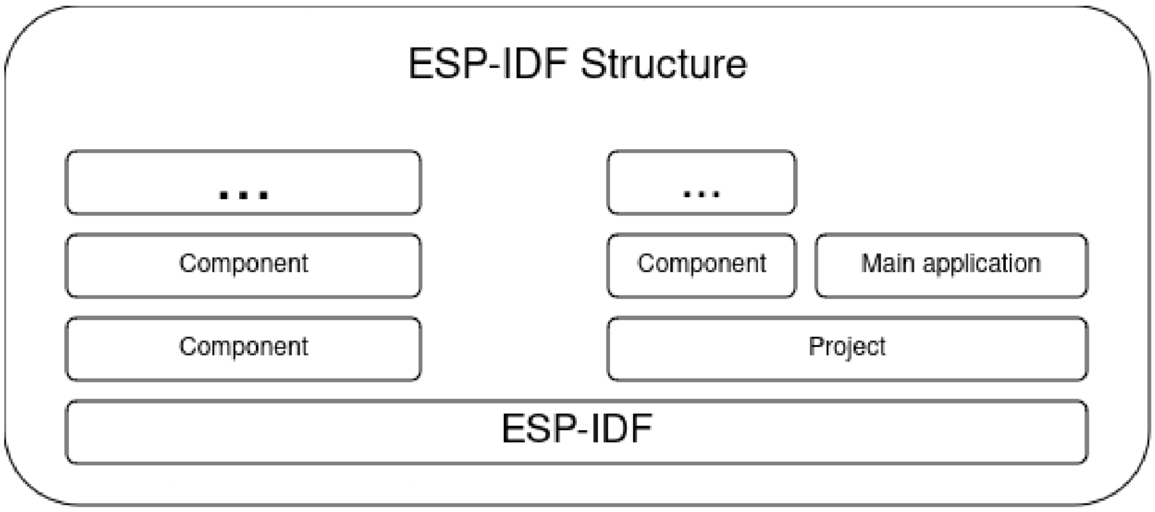


Figure 3.1: Visual representation of ESP-IDF structure

Compilers

ESP-IDF comes with GCC-based compilers for C and C++, specifically designed for the Xtensa chipset architecture. These compilers are custom made, since the Xtensa architecture is specifically designed for modular embedded systems. The toolchain also includes other utilities like GDB for debugging and ESPTool for interacting with the devices like flashing the compiled application.

Chapter 4

ROS2 - ESP solutions

4.1 Micro-ROS

https://micro.ros.org/ Micro-ROS is an adaptation of ROS 2 designed specifically for microcontrollers. This framework extends the ROS ecosystem into the realm of embedded systems by adapting ROS 2's features, tools, and conventions to resource-constrained devices. The goal of micro-ROS is to provide seamless integration with ROS 2, so microcontrollers can form complex behaviors and coordination between other microcontrollers or more powerful computing devices.

4.1.1 Integration with ROS

Micro-ROS is essentially a client of ROS 2, built to integrate seamlessly with the ROS 2 ecosystem. It adapts ROS 2 features to be suitable for the constraints of microcontrollers, such as reduced memory usage, lower processing power, and limited storage capacity. This integration ensures that systems developed with Micro-ROS can communicate with ROS 2 nodes. To achieve this, the client is split into a **client** and an **agent**.

Client

The micro-ROS client runs on the microcontroller, handling the execution of ROS 2 nodes, publishers, subscribers, services, and actions. It is designed to be lightweight and resource-efficient. The client provides an API that mirrors the API that is present in ROS 2 but is adapted to the constraints of embedded systems. The client has to connect to a micro-ROS agent in order to communicate with the broader ROS 2 ecosystem.

Agent

The micro-ROS agent acts as a bridge between the micro-ROS client and the ROS 2 data space. The agent runs on a more capable machine which allows it to translate messages to and from the micro-ROS clients, reducing the load of the clients, so the microcontrollers don't waste their limited resources. This allows microcontrollers to participate in a larger ROS 2 network as if they were regular ROS 2 nodes, sharing data and cooperating with other nodes across the network.

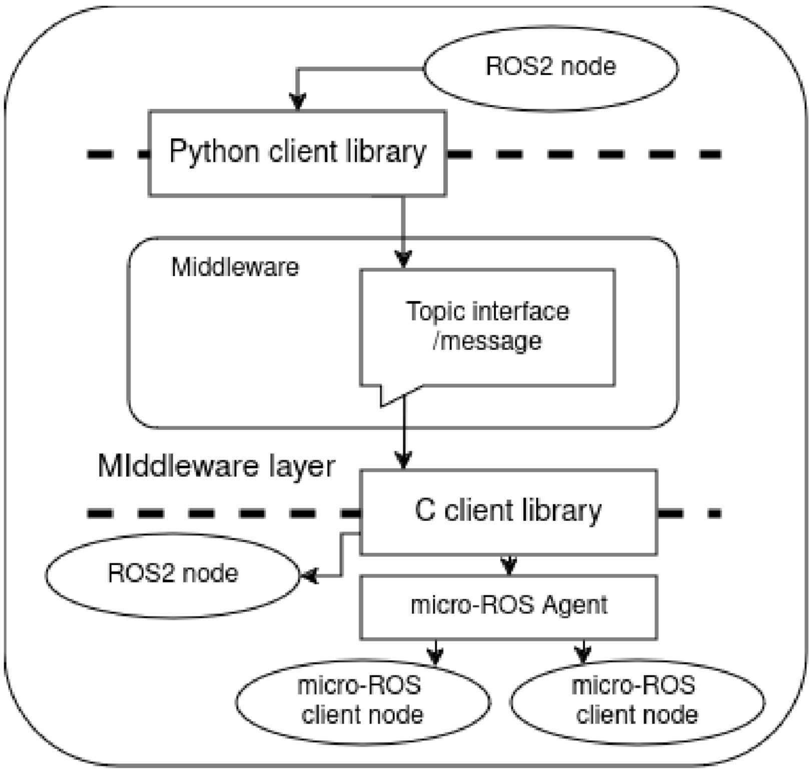


Figure 4.1: Visual representation of micro-ROS's agent-client structure in relation to the middleware

4.1.2 ROS 2 comparison

`https://micro.ros.org/docs/overview/ROS_2_feature_comparison/`

Becasue of the agent-client architecture, micro-ROS provides essentially the same features as ROS 2[2]. The client library `rclc` is a wrapper of ROS 2's base client library `rcl` with added convenience functions, which provides essentially the same API as with other ROS 2 client libraries.

4.2 Micro-ROS ESP-IDF component

The micro-ROS ESP-IDF component bridges the micro-ROS framework and the ESP-IDF, allowing the deployment of micro-ROS on ESP32 and other ESP-based microcontrollers.

4.2.1 IDF component

This ESP-IDF component integrates micro-ROS into the ESP-IDF ecosystem. This component produces a static library that can be linked to the ESP-IDF application. The integration is achieved within the ESP-IDF component's `CMakeLists.txt` file by wrapping the micro-ROS makefile in such a way that micro-ROS is compiled using the compilers provided by ESP-IDF.

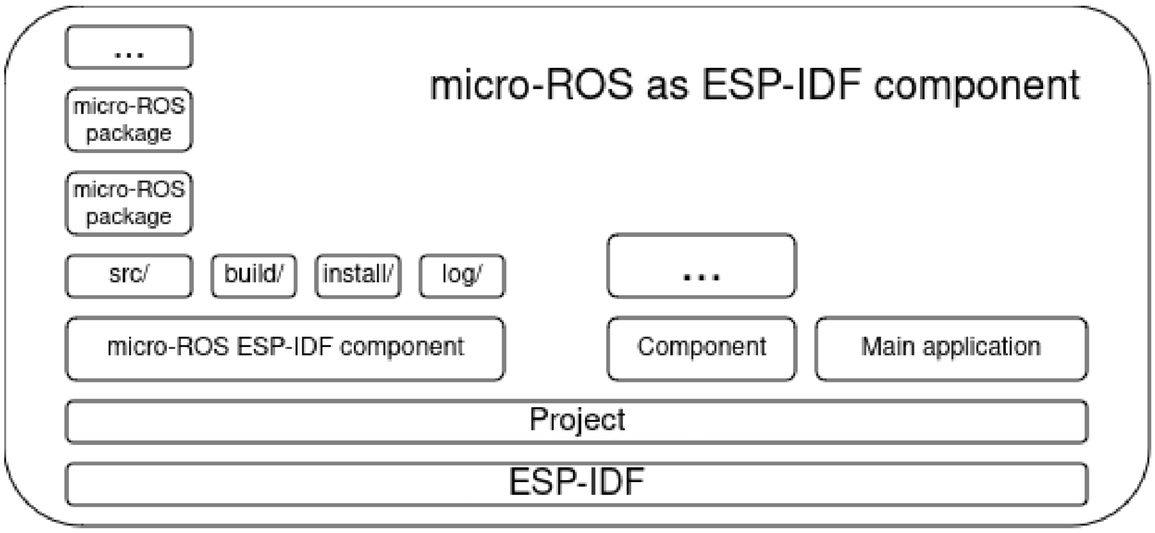


Figure 4.2: Visual representation of micro-ROS ESP-IDF project structure

CMake

The `CMakeLists.txt` file initially registers micro-ROS as an ESP-IDF component and setting up the necessary environment for its compilation. This CMake has several steps:

- **Environment Setup:** This configures essential variables and settings to ensure that micro-ROS compiles correctly under the ESP-IDF's compilation environment.
- **Compilation:** This invokes the micro-ROS build process to compile the `libmicroros` using ESP-IDF's toolchain, which can be utilized in the ESP environment.
- **Linking:** After micro-ROS is successfully compiled, it is linked as a static library. This step involves specifying dependencies and linking the library to ensure that it integrates well with other components of the ESP-IDF project.
- **Cleanup:** A custom cleanup command is registered to allow easy rebuilding of the micro-ROS component.

Chapter 5

Integration proposal

5.1 Current situation summary

Although micro-ROS addresses the integration of ROS into embedded systems and the micro-ROS ESP-IDF component addresses the integration of micro-ROS into the ESP-IDF plaform and hardware, integration of ESP-IDF applications, which utilized micro-ROS, into ROS2 ecosystem remained unaddressed. This led to fragmented developement and manual configuration of ROS2 projects which utilized ESP microcontrollers.

5.2 The goal of this thesis

The primary aim of this thesis is to streamline the integration of ESP-IDF projects into the ROS2 ecosystem. This involves developing methodologies and tools that will integrate ESP-IDF applications as ROS2 packages. This will reduce the manual labor which was previously needed to integrate ESP-IDF applicaitons into the ROS 2 ecosystem.

5.2.1 ESP-IDF projects as ROS2 packages

Primary objective is to provide a set of tools that would streamline the integration of ESP-IDF projects into the ROS 2 ecosystem. This is to simplify and streamline work with ESP-IDF projects within the ROS 2 ecosystem.

5.2.2 Micro-ROS IDF component as ROS2 package

Another objective is to integrate the Micro-ROS IDF component into the ROS2 ecosystem as a ROS package. This is to decopule the micro-ROS component library from a single ESP-IDF project. The build artifacts are quite large and takes relatively long time to compile. Leaving the micro-ROS component as an extra component inside each and every ESP-IDF project would consume unnecessary disk space and compilation time. This is also to simplify the addition of extra ROS packages like interface definitions, which will be immediately available to all ESP-IDF applications that would rely on this package.

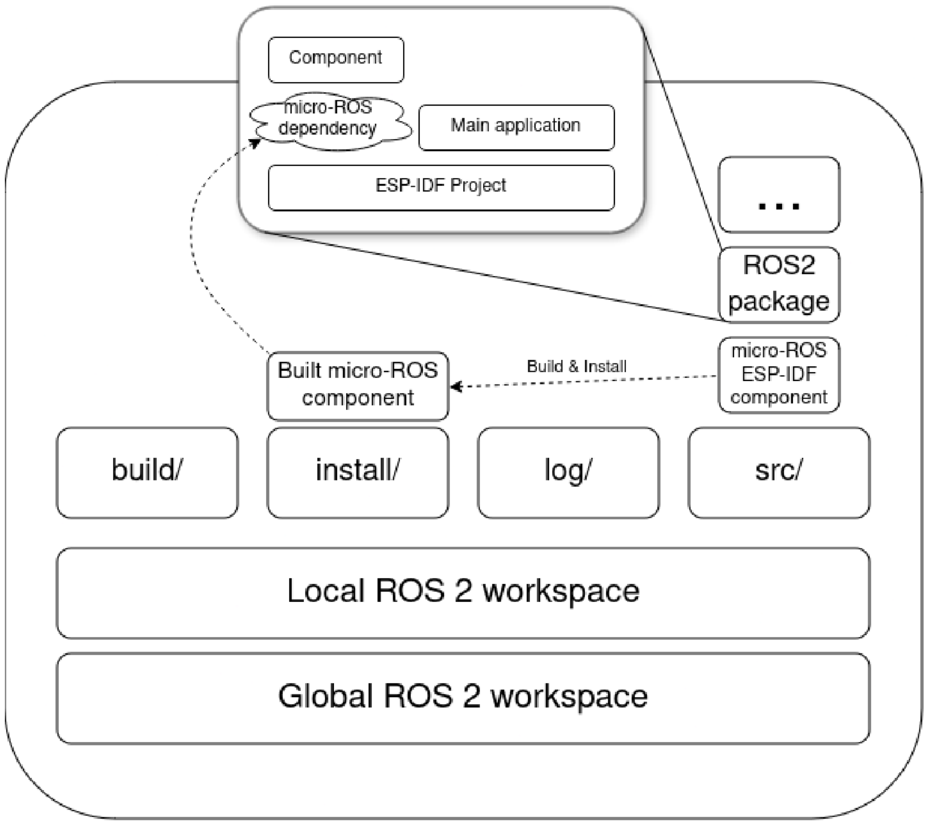


Figure 5.1: ESP-IDF project integration into ROS2

5.2.3 Example

The final objective of this thesis is to demonstrate the integration process through a practical example. This example involves creating a ROS 2 project that consists of four separate nodes and a custom interface for communication. These nodes collectively form a system that processes visual data and responds to environmental cues, showcasing the interaction between different components in a ROS 2 ecosystem.

- **Camera node**: This node is responsible for capturing live video feed from a camera. It is configurable, allowing adjustments to camera settings such as resolution or exposure through ROS 2 services
- **Camera user interface node**: This node provides a user interface for viewing the camera feed and interacting with the Camera Node. It allows the user to adjust camera settings of the Camera Node.
- **QR code detector node**: This node processes the video feed to detect QR codes. It identifies and decodes any QR code present in the camera feed and publishes the decoded information as ROS 2 messages.
- **LED strip node**: This node controls an LED strip, changing its color based on the information decoded from QR codes. The color change serves as a visual indicator of the QR code content, demonstrating a physical response to digital input.

Chapter 6

Implementation

https://github.com/ros2/ros2cli/issues/427

In the ROS 2 environment exists a command `ros2 pkg create <package-name> -build-type cmake, ament_cmake, ament_python`. The first attempt at creating a tool for integrating ESP-IDF projects was to expand upon this command. However this is not natively possible, so another method was needed.

The implementation comes in a single package `esp_env` which contains tools for setting up the micro-ROS component or arbitrary ESP-IDF projects and interacting with ESP-IDF projects inside the ROS 2 ecosystem.

6.1 `esp_env` package

The `esp_env` is an `ament_cmake` package which consists of three bash scripts `build_micro_ros`, `new_project`, `idf` and a `templates/` direcory that contains template files needed for new packages. The scripts automate the manual process of creating a ROS2 compliant package and interacting with the underlying ESP-IDF project if need be.

Package contents

- `CMakeLists.txt` — Basic CMake script that installs this package so the scripts will be available in the ROS 2 environment.
- `build_micro_ros` — A bash script that automates building the micro-ROS ESP-IDF package.
- `idf` — A bash wrapper around `idf.py` for other ESP-IDF projects within the ROS 2 environment. While every ESP-IDF ROS package comes with their own `idf.py` wrapper, sometimes the build requires configuration before compilation. This is achieved using the `idf.py menuconfig` command, but since the package was not built yet, the `idf` package wrapper is not available yet.
- `new_project` — A bash script that creates a new ESP-IDF package and integrates it into a ROS 2 package.
- `templates/` — A directory with template files and scripts used in creation of ESP-IDF projects as ROS 2 packages.

6.1.1 `idf` script

The `idf` bash script takes a package name as its only argument. If there are any other arguments provided to this script, all of them will be passed over unchanged to the subsequent `idf.py` command. Since the user is most likely to invoke this script from the root direcoty of the ROS 2 local workspace via the `ros2 run` command, the package name is required to correctly set up paths to the ROS 2 package that contains the ESP-IDF project.

```
PACKAGE_NAME="$1"
IDF_ARGS="${@:2}"
PACKAGE_DIR="$COLCON_PREFIX_PATH/../src/$PACKAGE_NAME"
PACKAGE_BUILD_DIR="$COLCON_PREFIX_PATH/../build/$PACKAGE_NAME/build"
UROS_COMPONENT_DIR="$(ros2 pkg prefix uros_esp32)/component"

if [ ! -d "$PACKAGE_DIR" ]; then
    echo "Package '$PACKAGE_NAME' not in current workspace!" > /dev/stderr
    exit 1
fi
```

- `PACKAGE_NAME` is the package name provided as an argument
- `IDF_ARGS` is the rest of provided arguments
- `PACKAGE_DIR` is where the ROS2 package which contains the ESP-IDF project is located
- `PACKAGE_BUILD_DIR` is where the build artifacts will be stored as per convention set by `colcon`
- `UROS_COMPONENT_DIR` is the location of the compiled micro-ROS component

In this section we set up all required variables to correctly invoke the `idf.py` command and check for any errors. If the micro-ROS component is not built (or it has been built but local install script, like `install/setup.bash`, has not been sourced yet), the bash command `$(ros2 pkg prefix uros_esp32)` will fail and the script will not continue. Likewise if `PACKAGE_DIR` is not a valid directory, that means that either colcon has not been set up correctly (local install script has not been sourced), or the specified package doesn't exist, the script fails and exits. This prevents incorrect setup for wrapping the `idf.py` command and invoking it for the correct ESP-IDF project. However, any additional commands are not sanitized, which can lead to intorrect usage of the `idf.py` command.

```
pushd "$PACKAGE_DIR/$PACKAGE_NAME" > /dev/null
    UROS_COMPONENT_DIR="$UROS_COMPONENT_DIR" \
            idf.py -B $PACKAGE_BUILD_DIR $IDF_ARGS
popd > /dev/null
```

In this section the environment is set up correctly and the `idf.py` command can be invoked to manage the specified ESP-IDF project. To invoke the `idf.py` command in the correct project we step into its direcory, which location is stored at previously set variable `PACKAGE_DIR`, using the `pushd` command. Since the micro-ROS component is decopuled, we need to somehow link the component back into the ESP-IDF build system. The linking itself is done by the ESP-IDF project itself 6.1.3, but we need to provide the correct path to the component. This is done by setting an environment variable `UROS_COMPONENT_DIR`. Then we invoke the `idf.py` command itself, but to comply with the colcon conventions, we specify the directory where the build artifacts should be stored. After the `idf.py` command is done, we return back to our current direcory by using the `popd` command.

6.1.2 `build_micro_ros` script

The `build_micro_ros` script is designed to automate the building of the Micro-ROS ESP-IDF component as a ROS2 package.

This script provides additional option, so there is a usage information that can be printed out by specifying the `-h` or `-help` argument.

```
COLCON_VERBOSE=""
if [ "$1" = "-v" ] || [ "$1" = "--colcon-verbose" ]; then
    COLCON_VERBOSE="--event-handle console_direct+"
fi
```

The option is to pipe build output from colcon directly into terminal. This option is enabled by specifying `-v` or `-colcon-verbose` argument, which upon detecting the script will set `COLCON_VERBOSE` variable that contains proper arguemt for colcon to pipe its build output directly into terminal.

```
UROS_PACKAGE_NAME="esp_uros"
PREFIX="$(ros2 pkg prefix esp_env)"
UROS_PREFIX="$(ros2 pkg prefix $UROS_PACKAGE_NAME)"
UROS_IDF_NAME="micro_ros_espidf_component"
UROS_SRC="$UROS_PREFIX/../../src/$UROS_PACKAGE_NAME"
```

Here, the script sets up several important variables:

- `UROS_PACKAGE_NAME` is the name of ROS 2 package for micro-ROS component ESP-IDF to reside in
- `PREFIX` gets the `esp_env` package prefix (which is the install directory)
- `UROS_PREFIX` gets the `UROS_PACKAGE_NAME` package prefix. This is to ensure we don't unnecessarily overwrite existing micro-ROS package
- `UROS_IDF_NAME` is the micro-ROS ESP-IDF component name and where the repository will be located
- `UROS_SRC` is the where the ROS 2 package containing micro-ROS ESP-IDF component located

```
MOVED_DIRS=
if [ ! "$(basename $PWD)" = "src" ]; then
    pushd "src" > /dev/null
    MOVED_DIRS=true
fi
```

Colcon commands have to be invoked from the local ROS 2 workspace directory, but commands like `ros2 pkg create` work with the directory from which the command is invoked. So to ensure the pakcage is created within the correct directory we check if we are in te `src/` directory. If we aren't, then we step into the `src/` directory. This script doesn't check if we are in the root of a workspace or even if the `src/` directory even exists. This is is intentional, since there is little to no reason to be anywhere else than the workspace root or `src/` directories.

```
if [ -z "$UROS_PREFIX" ]; then
    ros2 pkg create                     \
        --build-type ament_cmake        \
        --description "Micro ROS wrapper to esp32 microros component
                      for ease of use as a standalone ros2 package" \
        "$UROS_PACKAGE_NAME"
    pushd "$UROS_PACKAGE_NAME" > /dev/null
        UROS_SRC="$PWD"
        cp $PREFIX/templates/micro_ros.cmakelists CMakeLists.txt
        cp $PREFIX/templates/idf_build.sh .
        cp $PREFIX/templates/rebuild.sh rebuild
        cp $PREFIX/templates/add_packages.sh add_packages
        rm -r src include > /dev/null
    popd > /dev/null
    UROS_PREFIX="$(ros2 pkg prefix $UROS_PACKAGE_NAME)"
fi
```

In this section, the script checks if the Micro-ROS package exists in the workspace to prevent redownloading. If it doesn't exist, we create a new ROS2 package and copy over template files `templates/micro_ros.cmakelsits` as `CMakeLists.txt`, `templates/idf_build.sh`, `templates/rebuild.sh` as `rebuild` and `templates/add_packages.sh` as `add_packages` from the `esp_env` `templates/` directory that are required to build and manage the Micro-ROS component. It also removes unnecessary directories which were created by `ros2 pkg create` command.

```
if [ -f "$UROS_PREFIX/components/libmicroros.a" ]; then
    echo "Micro ROS is already built."
    exit 0
fi
```

This section simply checks if the Micro-ROS has already been built by looking for the 'libmicroros.a' file. If it exists, then the micro-ROS package is already build, so there is nothing more to do.

```
if [ "$MOVED_DIRS" ]; then
    popd > /dev/null
    colcon build --packages-select "$UROS_PACKAGE_NAME" $COLCON_VERBOSE
else
    pushd ".." > /dev/null
        colcon build --packages-select "$UROS_PACKAGE_NAME" $COLCON_VERBOSE
    popd > /dev/null
fi
```

Finally, if the micro-ROS component has not been built yet, this section builds the Micro-ROS package using ‘colcon‘. However as previously stated, the `colcon` command has to be invoked from the ROS 2 workspace root directory. So we need to change directories. Depending on where we invoked the `build_micro_ros` command we need to either go back to where we came from (if we stepped into the `src/` directory earlier), or step into the previous directory, which should be the ROS 2 workspace root.

6.1.3 `new_project` script

This script reduces the manual labor of creating an ESP-IDF project wrapped as a ROS 2 package.

This script takes at least one argument which is the name of the ROS 2 package. For easier configuration the containing ESP-IDF projects name is also the first argument of this script. Any additional arguments will be passed unchanged to the `ros2 pkg create` command.

```
MOVED_DIRS=""
if [ ! "$(basename $PWD)" = "src" ]; then
    pushd "src" > /dev/null
    MOVED_DIRS=true
fi
```

This is the same as in 6.1.2. If we aren’t in the `src/` directory, step in the `src/` directory.

```
ros2 pkg create $@ --build-type ament_cmake
pushd "$1" > /dev/null
    idf.py create-project "$1"
    pushd "$1" > /dev/null
        sed -i "/^project(.*)$/i \
                idf_build_component(\$ENV{UROS_COMPONENT_DIR})" \
                CMakeLists.txt
    popd >/dev/null
    cp $PREFIX/templates/new_project.cmakelists.in CMakeLists.txt
    sed -i "s/@PACKAGE_NAME@/$1/" CMakeLists.txt
    sed -i "/<license>/a \  <depend>esp_uros</depend>" package.xml
    cp $PREFIX/templates/idf.sh.in idf
    sed -i "s/@PACKAGE_NAME@/$1/" idf
    rm -r src/ include/
popd > /dev/null
```

Here we create the new ROS 2 package. As stated before, any additional arguments besides the package name will be passed over to the `ros2 pkg create` command. The build type is `ament_cmake` since ESP-IDF projects also rely on CMake. This will make it easier to integrate the ESP-IDF project into the ROS 2 environment

Next we step into the newly created ROS 2 package and create a new ESP-IDF project with the same name. If there is a need to configure the ESP-IDF project further using the `idf.py` command, we can use the `ros2 run esp_env idf <package_name> [<idf_argument> ...]` command, or compile the package once, so its own `idf` script will be installed and the `ros2 run <package_name> idf [<idf_argument> ...]` command will be available to reach the package with the `idf.py` tool.

After the ESP-IDF package is created, we step into the project directory and insert the `idf_build_component(`
`$ENVUROS_COMPONENT_DIR)`„ line to the project's `CMakeLists.txt`, so that the decopuled micro-ROS component can be used within the newly created ESP-IDF project.

`https://cmake.org/cmake/help/latest/command/configure_file.html#transformations`

As we step out of the ESP-IDF project back into the ROS 2 package directory, we copy the `templates/new_project.cmakelists.in` from the `esp_env` package and save it as `CMakeLists.txt`. This CMakeLists.txt file has **CMake tranformations** style variable, so this could have been imported by CMake. This is ultimately not included by CMake, but the template variable stayed in this style. We replace the `@PACKAGE_NAME@` inside with the package name provided as an argument. Then we insert an `esp_uros` dependency inside the `package.xml` manifest file.

We then copy the `templates/idf.sh.in` file as just an `idf` file. We make the same replacement here as before, replacing `@PACKAGE_NAME@` with the package name provided by the agrument.

Lastly we remove unnecessary directories that were generated by the `ros2 pkg create` command and go back to the direcotory the script was invoked in.

6.2 esp_uros package

The `esp_uros` package, designed as an `ament_cmake` package, allows the integration and management the Micro-ROS ESP-IDF component within the ROS2 environment. This package contains several scripts essential for building, adding packages, and maintaining the Micro-ROS environment.

Package Contents

The package includes:

- `CMakeLists.txt` which prepares and builds the micro-ROS ESP-IDF project and handles the installation of the compiled component as well as its management scripts
- `add_packages` automates adding additional ROS2 packages to the Micro-ROS build environment.
- `idf_build.sh` automates the building process of the Micro-ROS ESP-IDF component..
- `rebuild` automates cleanint and rebuilding the Micro-ROS component.

6.2.1 CMake

The `CMakeLists.txt` file initially utilized the `ExternalProject_Add` function for integrating the external Micro-ROS ESP-IDF component hosted on GitHub. However after adding additional package to the micro-ROS environment, the CMake re-downloaded the entire repository, which led to the removal of all extra packages. The build step was modified to manual download and build processes, even though there already exists a solution for adding external projects into CMake.

```
set(IDF_PROJECT_NAME "micro_ros_espidf_component")
set(UROS_SOURCE_DIR ${CMAKE_SOURCE_DIR}/${IDF_PROJECT_NAME})
set(DUMMY_PROJ_DIR ${UROS_SOURCE_DIR}/examples/int32_publisher)
```

This configuration sets essential options needed for compilation:

- `IDF_PROJECT_NAME` is the name of the micro-ROS ESP-IDF component
- `UROS_SOURCE_DIR` is the location of the micro-ROS ESP-IDF component
- `DUMMY_PROJ_DIR` is the location of an ESP-IDF example application utilizing micro-ROS

```
if(NOT EXISTS ${UROS_SOURCE_DIR})
    execute_process(
        COMMAND git clone
                -b $ENV{ROS_DISTRO}
                https://github.com/micro-ROS/${IDF_PROJECT_NAME}
        WORKING_DIRECTORY ${CMAKE_SOURCE_DIR})
endif()
```

This section ensures that the micro-ROS ESP-IDF component with the correct branch is fetched from its repository if it is not already present in the workspace.

```
add_custom_target(
    ${PROJECT_NAME}_build ALL
    env -i
        IDF_PATH=$ENV{IDF_PATH}
        PROJECT_BUILD_DIR=${CMAKE_BINARY_DIR}/build
        DISPLAY=$ENV{DISPLAY}
        ${CMAKE_SOURCE_DIR}/idf_build.sh
    WORKING_DIRECTORY ${DUMMY_PROJ_DIR}
    VERBATIM
)
```

This section builds the micro-ROS ESP-IDF component. The build process uses a clean environment to avoid conflicts between the ESP-IDF's usage of colcon and the ROS 2 environment. This is achieved by the `env -i` command, which creates a completly clean environment. However the environment cannot be completly clean, as we need the ESP-IDF toolchain to be able to build the ESP-IDF component. We provide this with the `IDF_PATH` variable, which points to the root directory of our currently active ESP-IDF toolchain. Along with this we provide the package's conventional ROS 2 build directory location. The `DISPLAY` variable passed as colcon uses it for sending system notifications about its build status.

```
add_custom_command(
    TARGET ${PROJECT_NAME}_build
    POST_BUILD
    COMMAND sed ARGS -n "/^idf_component_register/,/)/p"
                    CMakeLists.txt > /tmp/CMakeLists.txt
```

```
        COMMAND sed ARGS -n "/^add_prebuilt_library/,//p"
                         CMakeLists.txt >> /tmp/CMakeLists.txt
        WORKING_DIRECTORY ${UROS_SOURCE_DIR}
        VERBATIM
    )
```

This command generates a modified `CMakeLists.txt` from the micro-ROS ESP-IDF component without its build step. In the installation step we copy all necessary files, which the original `CMakeLists.txt` would cause problems with the ESP-IDF's build system. When other ESP-IDF projects would include the `esp_uros` installed component, the ESP-IDF's build system would attempt to rebuild the micro-ROS library. To prevent this we removed the build step and leave only the links to the built library.

```
install(
    DIRECTORY
        ${IDF_PROJECT_NAME}/include
        ${IDF_PROJECT_NAME}/network_interfaces
    DESTINATION component
    USE_SOURCE_PERMISSIONS
)
install(
    FILES
        /tmp/CMakeLists.txt
        ${IDF_PROJECT_NAME}/Kconfig.projbuild
        ${IDF_PROJECT_NAME}/libmicroros.a
    DESTINATION component
)

install(
    PROGRAMS
        add_packages
        rebuild
    DESTINATION lib/${PROJECT_NAME}
)
```

This installation script places the necessary header files located in the `include/` and `network_interfaces/` directories, the built micro-ROS library `libmicroros.a` along with its `Kconfig.projbuild` and `CMakeLists.txt` files, and `add_packages` and `rebuild` scripts into the appropriate locations within the ROS 2 package structure, ensuring they are accessible for projects that depend on this package.

`idf_build.sh` script

```
#!/usr/bin/bash

if [ -z "$IDF_PATH" ]; then
    echo "IDF_PATH is undefined!" > /dev/stderr
    exit 1
fi
```

```
export PATH=$(getconf PATH)
. "$IDF_PATH/export.sh" > /dev/null

if [ "$PROJECT_BUILD_DIR" ]; then
    PROJECT_BUILD_DIR="-B $PROJECT_BUILD_DIR"
fi

idf.py $PROJECT_BUILD_DIR build
```

Because when building the micro-ROS ESP-IDF project we work within a clean environment (see 6.2.1), which is why we need to set up the ESP-IDF toolchain. To achieve this we source the texttt$IDF_PATH/export.sh in our clean environment. After which we can invoke the `idf.py` command and build the micro-ROS ESP-IDF component.

6.2.2 add_packages script

The `add_packages` script simplifies the process of adding external ROS2 packages to the Micro-ROS environment by automating the copying of package directories.

The script takes one or more package names as arguments. Since this script behavior depends on its arguments and it might not be entirely clear on what those should be, there is an agument option (`-h` or `-help`) that will print the script's usage information.

```
    PACKAGE_DIR="$COLCON_PREFIX_PATH/../src"
    echo -n "$@" | xargs -d " " -I % cp -r $PACKAGE_DIR/% \
            "$PACKAGE_DIR/esp_uros/micro_ros_espidf_component/extra_packages/"
```

This script will copy any directories (which should be valid ROS 2 packages) that were provided by arguments to the micro-ROS ESP-IDF project's `extra_packages/` directory. This is achieved by the `xargs` command which can run duplicate commands based on its input. We utilize this to run `cp -r` command for each provided package.

6.2.3 rebuild script

The `rebuild` script simplifies the rebuilding of the `esp_uros` package. When trying to build the package again the micro-ROS would not get rebuilt. This is because of how the component gets built, which it builds an ESP-IDF example provided by the micro-ROS package, the ESP-IDF build system will not trigger a rebuild of the micro-ROS component when there are no direct changes to its dependencies.

The only way to rebuild the micro-ROS package was to either manually invoke `make -f libmicroros.mk clean` in the micro-ROS ESP-IDF component directory, or to remove the whole directory, upon which CMake would download the project repository again (see 6.2.1).

The `rebuild` script has the same options and help message as the `build_micro_ros` (see 6.1.2) from the `esp_env` package.

```
    pushd "src/esp_uros/micro_ros_espidf_component" > /dev/null
        make -f libmicroros.mk clean
    popd > /dev/null
    colcon build --packages-select esp_uros $VERBOSE
```

This script takes the former route of stepping into the micro-ROS ESP-IDF directory and invoking the `make -f libmicroros.mk clean` command, which removes all build artifacts. The script assumes that we are invoking it from the ROS 2 local workspace root directory, so that it can step into the micro-ROS ESP-IDF component directory directly.

Lastly the script just invokes `colcon build` and let CMake take care of the rebuild.

6.3 ESP-IDF project as a ROS2 package

https://docs.espressif.com/projects/esp-idf/en/v5.2.1/esp32/api-guides/build-system.html Despite the fact that both ROS 2's and ESP-IDF's build systems are based on CMake, the ESP-IDF project integration into the ROS 2 environment was quite challenging. Initial attempts to integrate directly using ESP-IDF's CMake API were unsuccessful, leading to the adoption of a simpler, more robust approach using CMake's `ExternalProject_Add` function.

This approach is particularly suitable given that the ESP-IDF projects, that would be integrated as ROS 2 packages, are typically applications rather than libraries, and do not generally require exporting build artifacts as reusable components. However, if exporting is necessary, the methodology used in the `esp_uros` (see 6.2.1) package for handling ESP-IDF components can be adapted.

Package Contents

- A directory named after the ROS 2 package containing the ESP-IDF project.
- `CMakeLists.txt` that prepares and builds the containing ESP-IDF project
- `idf` bash script that wraps the `idf.py` command

6.3.1 `CMakeLists.txt`

The `CMakeLists.txt` for integrating an ESP-IDF project is relatively straightforward.

```
    find_package(esp_uros REQUIRED)
    set(UROS_COMPONENT_DIR ${uros_esp32_DIR}/../../../component)
```

This script utilizes ROS 2's ament package management to locate the `esp_uros` package, setting up a link to the built micro-ROS component.

```
include(ExternalProject)
ExternalProject_Add(
    ${PROJECT_NAME}
    SOURCE_DIR ${CMAKE_CURRENT_SOURCE_DIR}/${PROJECT_NAME}/main
    BINARY_DIR ${CMAKE_CURRENT_SOURCE_DIR}/${PROJECT_NAME}
    CONFIGURE_COMMAND ""
    BUILD_COMMAND UROS_COMPONENT_DIR=${UROS_COMPONENT_DIR} \
            idf.py -B ${CMAKE_BINARY_DIR}/build build
    INSTALL_COMMAND ""
    BUILD_ALWAYS TRUE
)
```

Here, `ExternalProject_Add` configures the ESP-IDF project, similar to `esp_uros` package (see 6.2.1), as an external project within the ROS 2 build environment. However, unlike in the `esp_uros` CMake setup, there is no need for a clean environment because typical ESP-IDF projects do not conflict with ROS 2 settings. The project is set to rebuild always to ensure updates are processed during every build cycle.

The CMake build finishes by installing the `idf` script, so the `idf.py` command can be utilized later.

6.3.2 `idf` script

The `idf` script in this package mirrors the `idf` script in the `esp_env` package (see 6.1.1), providing a consistent interface for project configuration and building. This script simplifies the execution of the `idf.py` tool, streamlining the development process for the package's ESP-IDF project.

Chapter 7

Example Implementation

This example illustrates the integration of ESP32 microcontrollers into the ROS 2 ecosystem by implementing a multi-node application that interfaces with both hardware and software components. The implementation consists of four nodes: **Camera Node**, **Camera User Interface Node**, **QR Detector Node**, and **LED Strip Node**. Communication between the camera node and the camera user interface node is dependent on a **custom interface**.

7.1 Custom Interface

To efficiently configure the **camera node**, we have defined a custom message and service types.

7.1.1 CameraConfig message

The `CameraConfig.msg` defines the structure for camera configuration settings. It includes a set of parameters that can be adjusted to modify the behavior and output of the camera.

7.1.2 GetCameraConfig service

The `GetCameraConfig.srv` service is defined to allow the retrieval of the current camera configuration on demand. The service does not require any input parameters and responds with the current configuration settings.

```
---
custom_interface/CameraConfig config
```

7.1.3 SetCameraConfig Service

The `SetCameraConfig.srv` service allows for the configuration of the camera settings. The service accepts a `CameraConfig` message as a request and returns a status boolean indicating the success or failure of the operation, along with the new camera configuration:

```
custom_interface/CameraConfig config
---
bool status
custom_interface/CameraConfig config
```

7.2 Camera Node

The Camera Node is implemented on an ESP32 microcontroller with an attached camera module. Its primary function is to capture and periodically publish images from the camera feed through a ROS 2 topic. Additionally, this node offers services to retrieve the current camera configuration and to adjust the configurations.

Since this node deals with images, the ESP32 will most likely not have enough RAM to store the images above a certain size. Enabling **PSRAM** solves this issue.

7.2.1 Camera initialization

```
    camera_config_t camera_config = {
            ...
        .pixel_format = PIXFORMAT_JPEG,
        .frame_size = FRAMESIZE_UXGA,

        .jpeg_quality = 10,
        .fb_count = 2,
        .grab_mode = CAMERA_GRAB_WHEN_EMPTY
    };
    esp_err_t ret = esp_camera_init(&camera_config);
    if(ret != ESP_OK) return ret;
    sensor_t * s = esp_camera_sensor_get();
    ret = s->set_framesize(s, FRAMESIZE_QVGA);
    if(ret != ESP_OK) return ESP_FAIL;
    return ESP_OK;
```

To initialize the camera, the camera configuration structure is populated and the camera is started. This code configures the ESP32 camera to the highest camera resolution, which allocates the frame buffer to fit the maximum frame size possible during the initialization. This ensures that all subsequent camera image grabs will have enough space to store the frame buffer.

7.2.2 Node initialization

```
    rcl_allocator_t allocator = rcl_get_default_allocator();
    rclc_support_t support;

    rcl_init_options_t init_options = rcl_get_zero_initialized_init_options();
    RCCHECK(rcl_init_options_init(&init_options, allocator));

#ifdef CONFIG_MICRO_ROS_ESP_XRCE_DDS_MIDDLEWARE
    rmw_init_options_t* rmw_options =
                rcl_init_options_get_rmw_init_options(&init_options);
    RCCHECK(rmw_uros_options_set_udp_address(CONFIG_MICRO_ROS_AGENT_IP,
                                            CONFIG_MICRO_ROS_AGENT_PORT,
                                            rmw_options));
#endif
    RCCHECK(rclc_support_init_with_options(&support, 0, NULL,
                                              &init_options, &allocator));

    rcl_node_t node;
    RCCHECK(rclc_node_init_default(&node, CONF_ROS_NODE_NAME, "", &support));
```

This section sets up a default micro-ROS node, initializing the default ROS environment and connecting to a micro-ROS agent.

```
    RCCHECK(rclc_publisher_init_default(
        &publisher,
        &node,
        ROSIDL_GET_MSG_TYPE_SUPPORT(sensor_msgs, msg, CompressedImage),
        CONFIG_CAMERA_TOPIC_PATH)
    );
```

This section initializes the publisher responsible for publishing compressed images captured by the camera to the topic defined by the project configuration.

```
    rcl_service_t srv_cam_conf_set;
    custom_interface__srv__SetCameraConfig_Request msg_cam_conf_set_req;
    custom_interface__srv__SetCameraConfig_Response msg_cam_conf_set_res;
    RCCHECK(rclc_service_init_default(
        &srv_cam_conf_set,
        &node,
        ROSIDL_GET_SRV_TYPE_SUPPORT(custom_interface, srv, SetCameraConfig),
        CONF_CAMERA_SET_CONF_TOPIC)
    );
    rcl_service_t srv_cam_conf_get;
    custom_interface__srv__GetCameraConfig_Request msg_cam_conf_get_req;
    custom_interface__srv__GetCameraConfig_Response msg_cam_conf_get_res;
    RCCHECK(rclc_service_init_default(
        &srv_cam_conf_get,
        &node,
        ROSIDL_GET_SRV_TYPE_SUPPORT(custom_interface, srv, GetCameraConfig),
        CONF_CAMERA_GET_CONF_TOPIC)
    );
```

This segment details the initialization of two ROS services within the camera node. The first service allows users to set the camera configuration, and the second service enables retrieving the current camera settings. The paths of these services are defined in the projects configuration.

```
    rcl_timer_t timer;
    RCCHECK(rclc_timer_init_default(&timer,
                                    &support,
                                    RCL_MS_TO_NS(10),
                                    publish_timer_callback));
```

In this section a timer is created to trigger the `publish_timer_callback` function with a timer execution timeout of 10 milliseconds.

```
    img_msg.header.frame_id = micro_ros_string_utilities_init("ESP32 camera");
    img_msg.header.stamp.sec = 0;
    img_msg.header.stamp.nanosec = 0;
    img_msg.format = micro_ros_string_utilities_init("jpeg");
```

Immediately following node initialization, we prepare the image message that will be published. It sets the frame identifier, timestamp, and image format, establishing the necessary metadata for image messages sent by this node.

```
    rclc_executor_t executor;
    RCCHECK(rclc_executor_init(&executor, &support.context, 3, &allocator));
    RCCHECK(rclc_executor_add_service(&executor,
                                      &srv_cam_conf_get,
                                      &msg_cam_conf_get_req,
                                      &msg_cam_conf_get_res,
                                      srv_cam_conf_get_callback));
    RCCHECK(rclc_executor_add_service(&executor,
                                      &srv_cam_conf_set,
                                      &msg_cam_conf_set_req,
                                      &msg_cam_conf_set_res,
                                      srv_cam_conf_set_callback));
    RCCHECK(rclc_executor_add_timer(&executor, &timer));
```

The executor is a critical component in ROS 2, responsible for managing the callbacks associated with timers and services. This initialization code sets up the executor to handle service requests for camera configuration and to invoke the publishing timer, allowing the asynchronous functionality of the node. In `rclc` we have to declare the maximum number of registrations for the executor, since the ESP32 is a device with limited resources. This allows for better memory management, since the executor size is not dynamic.

```
    while(1){
        rclc_executor_spin_some(&executor, RCL_MS_TO_NS(10));
        vTaskDelay(1 / portTICK_PERIOD_MS);
    }
```

This loop is the core of the node's operation. It repeatedly checks for events such as timer expirations or service requests and processes them as needed. The use of `rclc_executor_spin_some` allows the node to remain responsive, processing incoming requests and maintaining regular image publication.

7.2.3 Event handling

Camera feed publishing

```
void publish_timer_callback(rcl_timer_t *timer, int64_t last_call_time) {
    static int64_t since_last_publish = 0;
    if (!timer) return;

    since_last_publish += last_call_time;
    if (since_last_publish < RCL_MS_TO_NS(33)) return;
    since_last_publish = 0;

    camera_fb_t *fb = NULL;
    fb = esp_camera_fb_get();
    if(!fb) {
        ESP_LOGW(TAG, "Camera capture failed.");
        return;
    }

    img_msg.data.data = fb->buf;
    img_msg.data.size = fb->len;
    img_msg.data.capacity = fb->len;

    ESP_LOGD(TAG, "Publishing Image: %dx%d - %d B",
                  fb->width, fb->height, fb->len);
    RCSOFTCHECK(rcl_publish(&publisher, &img_msg, NULL));

    esp_camera_fb_return(fb);
    img_msg.data.data = NULL;
    img_msg.data.size = 0;
    img_msg.data.capacity = 0;
}
```

In this function, the timer callback manages the periodic publishing of the camera feed. It ensures a frame rate limit of approximately 30 frames per second by publishing the image roughly every 1/30th of a second. The function retrieves a frame buffer from the ESP camera and publishes it. After publication, the buffer is returned to free up memory to prevent memory leaks.

Service handling

```
void srv_cam_conf_get_callback(const void *req, void *res) {
    RCLC_UNUSED(req);
    custom_interface__srv__GetCameraConfig_Response *response =
            (custom_interface__srv__GetCameraConfig_Response *) res;

    const sensor_t *camera = esp_camera_sensor_get();
    copy_camera_status(&(response->config), camera->status);
    response->config.pixformat = camera->pixformat;
}
```

This callback function handles requests to get the current camera configuration. It extracts the camera settings from the hardware and populates the response object, which is then sent back to the requester. This service is crucial for clients needing to adjust settings based on current configurations.

```
void srv_cam_conf_set_callback(const void *req, void *res) {
    custom_interface__srv__SetCameraConfig_Request *request =
        (custom_interface__srv__SetCameraConfig_Request *) req;
    custom_interface__srv__SetCameraConfig_Response *response =
        (custom_interface__srv__SetCameraConfig_Response *) res;
    sensor_t *camera = esp_camera_sensor_get();

    custom_interface__msg__CameraConfig tmp;
    copy_camera_status(&tmp, camera->status);

    response->status = 0;
    SET_IF_DIFFERENT(framesize);
    ...
    SET_IF_DIFFERENT(colorbar);
    SET_IF_DIFFERENT_EXPLICIT(awb, whitebal);
    SET_IF_DIFFERENT_EXPLICIT(aec, exposure_ctrl);
    SET_IF_DIFFERENT_EXPLICIT(agc, gain_ctrl);
    copy_camera_status(&(response->config), camera->status);
    copy_camera_status(&tmp, camera->status);
}
```

This callback function handles service requests to set new camera configurations. It compares requested settings to current settings and only applies changes if they differ, optimizing performance by avoiding unnecessary hardware interactions. After adjustments, it updates the response with the new settings to confirm the changes to the client.

7.3 Camera User Interface Node

Camera User Interface is a Python node which also runs a **Flask** server to interact with users. This node subscribes to the image data published by the Camera Node, forwarding the live camera feed to the user interface. Users can modify camera settings through the web interface. Upon receiving user inputs, the node requests an adjustment to the new settings to the Camera Node using the **custom interface**.

7.3.1 Node initialization

```
def __init__(self):
    super().__init__('ESP32_Camera_Web_Server')
    self.subscription = self.create_subscription(CompressedImage,
                                                 '/esp32/camera/image_jpeg',
                                                 self.image_callback, 1)
    self.sub_color = self.create_subscription(String,
                                              '/qr_message',
                                              self.qr_color_callback, 1)

    self.get_conf_client = self.create_client(GetCameraConfig,
                                              '/esp32/camera/config/get')
    self.set_conf_client = self.create_client(SetCameraConfig,
                                              '/esp32/camera/config/set')

    self.get_conf_timer = None
    self.set_conf_timer = None
    self._get_promise = None
    self._set_promise = None
    self.current_cam_conf = CameraConfig()

    self._wait_for_services()
    self.request_config()
```

This function initializes the node and its subscriptions to both image and QR color data streams. It then initiates service availability checks and requests the current camera configuration to synchronize its state with the camera's settings.

```
def _wait_for_services(self):
    print("Waiting for Getter", end="")
    while not self.get_conf_client.wait_for_service(timeout_sec=1.0):
        self.get_logger().info('Get service not available, waiting again...')
        print(".", end="")
    print()
    self.srv_get_req = GetCameraConfig.Request()

    print("Waiting for Setter", end="")
    while not self.set_conf_client.wait_for_service(timeout_sec=1.0):
        self.get_logger().info('Set service not available, waiting again...')
        print(".", end="")
    print()
    self.srv_set_req = SetCameraConfig.Request()
```

This segment initializes the `GetCameraConfig` and `SetCameraConfig` service clients. The node will check every second to see if the services are available before proceeding.

7.3.2 Node communication

```
def request_config(self):
    def _request_get():
        if self._get_promise and self._get_promise.done():
            self.current_cam_conf = self._get_promise.result().config
            self._get_promise = None
            self.destroy_timer(self.get_conf_timer)
            self.get_conf_timer = None

    if not self.get_conf_timer:
        self.get_conf_timer = self.create_timer(0.1, _request_get)
        self._get_promise = self.get_conf_client.call_async(self.srv_get_req)
```

This method manages asynchronous requests to retrieve the current camera configuration. It sets a timer to check the promise status and updates the configuration state upon completion.

```
def adjust_config(self, new_config):
    def _request_set():
        if self._set_promise and self._set_promise.done():
            self.current_cam_conf = self._set_promise.result().config
            self._set_promise = None
            self.destroy_timer(self.set_conf_timer)
            self.set_conf_timer = None

    if not self.set_conf_timer:
        self.set_conf_timer = self.create_timer(0.1, _request_set)
        self.srv_set_req.config = new_config
        self._set_promise = self.set_conf_client.call_async(self.srv_set_req)
```

This function handles the adjustment of camera settings by sending asynchronous service requests to set the new configuration parameters.

7.3.3 Flask server

`https://htmx.org/`

Flask is a simple developement http server. For client-side reactivity we boosted the html with HTMx.

```
    app = Flask(__name__, root_path=f"{env['COLCON_PREFIX_PATH']}/
                                      {__name__.split('.')[0]}/flask")
```

The Flask server is initialized with a root path of the installed HTML templates.

Server paths

```
@app.route("/", methods=["GET", "POST"])
def index():
    global server

    if request.method == "POST":
        converted = {}
        for key, val in request.form.items():
            converted[key] = int(val.replace("on", "1"))
        new_config = CameraConfig(**converted)
        node.set_config(new_config)
        sleep(1)
    server.update_config(node.get_config())

    return render_template("index.html", menus=server.menus,
                                         sliders=server.sliders,
                                         toggles=server.toggles,
                                         qr_color=qr_color)
```

The main server route handles both GET and POST requests. POST requests process form submissions to adjust camera configurations, while GET requests serve the main page template populated with the current settings.

```
@app.route('/stream')
def image_stream():
    global server
    return Response(server.generate_image_stream(),
                    mimetype='multipart/x-mixed-replace; boundary=frame')
```

This route serves a continuous stream of images from the camera, formatted as **multipart/x-mixed-replace** for continuous update without requiring page refreshes.

```
@app.route("/qr_color")
def qr_color_square():
    return f'<div class="color_box"
                 hx-get="/qr_color"
                 hx-swap="outerHTML"
                 hx-trigger="every 1s"
                 style="background-color: {qr_color};"></div>'
```

This route dynamically updates the background color of a displayed element based on the most recently detected QR code, utilizing HTMx to periodically fetch updates without refreshing the page.

7.4 QR Detector Node

QR Detector is a Python node that processes the incoming video feed from the Camera Node to identify and decode QR codes. Each detected QR code is decoded, and the content is published onto a ROS 2 topic, making the information available to other nodes in the network. This node leverages image processing techniques of the `pyzbar` python module to efficiently scan and interpret QR codes from the video stream.

7.4.1 Node initialization

```
def __init__(self):
    super().__init__('QR_Scanner')
    self.publisher_ = self.create_publisher(String, '/qr_message', 1)
    self.subscription = self.create_subscription(CompressedImage,
                                                 '/esp32/camera/image_jpeg',
                                                 self.image_callback, 1)
```

This function initializes the QR Detector Node, setting up a publisher for the detected QR code messages on the `/qr_message` topic. It also subscribes to the `/esp32/camera/image_jpeg` topic to receive compressed images from the Camera Node, which are then processed to detect QR codes.

7.4.2 QR detection

```
def image_callback(self, msg):
    img = PIL_Image.open(io.BytesIO(bytes(msg.data)))
    decoded_objects = decode(img)
    self._old_qr
```

```
for obj in decoded_objects:
    qr_message = obj.data.decode("utf-8")
    if  self._old_qr != qr_message:
        self._old_qr = qr_message
        print(f"Decoded new qr_message {qr_message}")
        tmp = String()
        tmp.data = qr_message
        self.publisher_.publish(tmp)
```

Each incoming image frame is converted from its compressed format into a `PIL` image object. The `pyzbar` library's `decode` function scans the image for QR codes. If a QR code is detected and its content has changed from the previous scan, the new content is published to the `/qr_message` topic. This method ensures that only new or changed QR code information is sent, reducing redundant data transmission and enhancing the efficiency of the system.

7.5 LED Strip node

The LED Strip node is implemented on an ESP32 microcontroller, which controls an RGB LED strip. This node listens to the messages published by the QR Detector Node. When it receives a message containing a hex color code (formatted as either #RGB or #RRGGBB), it parses the data and changes the color of the LED strip. This setup demonstrates the reactive capability of the system to image feedback received by a camera sensor.

7.5.1 Node initialization

LED strip node initialization is largely same as Camera node (see 7.2.2).

```
RCCHECK(rclc_subscription_init_default(
    &subscriber,
    &node,
    ROSIDL_GET_MSG_TYPE_SUPPORT(std_msgs, msg, String),
    CONFIG_ROS_COLOR_TOPIC));
```

This section initializes a subscription to a topic specified by `CONFIG_ROS_COLOR_TOPIC`. This topic carries messages that potentially contain hex color codes. If the message is indeed a hex color code, the node will change the color of the LED strip to match the received color.

```
while(1){
    rclc_executor_spin_some(&executor, RCL_MS_TO_NS(100));
    vTaskDelay(10 / portTICK_PERIOD_MS);
}
```

In this loop, the node periodically checks for new messages with a reduced processing demand, since it has a non-critical role in the system. This frequency reduction helps to conserve computational resources and bandwidth, prioritizing more critical tasks within the network.

7.5.2 Subscriber handling

```
void subscribe_ledcolor_callback(const void *msg) {
    std_msgs__msg__String *parsed_msg = (std_msgs__msg__String *)msg;
    if(parsed_msg->data.data[0] != '#'){
        return;
    }
    if(!led_strip) {
        ESP_LOGW(TAG, "LED strip not initialized!");
        return;
    }

    uint8_t r, g, b;
    char tmp[3] = {0};
    switch(parsed_msg->data.size) {
        case 4: {
            tmp[0] = parsed_msg->data.data[1];
            r = (strtol(tmp, NULL, 16)&0xf) << 4;
            tmp[0] = parsed_msg->data.data[2];
            g = (strtol(tmp, NULL, 16)&0xf) << 4;
            tmp[0] = parsed_msg->data.data[3];
            b = (strtol(tmp, NULL, 16)&0xf) << 4;
        } break;
        case 7: {
            tmp[0] = parsed_msg->data.data[1];
            tmp[1] = parsed_msg->data.data[2];
            r = strtol(tmp, NULL, 16)&0xff;
            tmp[0] = parsed_msg->data.data[3];
            tmp[1] = parsed_msg->data.data[4];
            g = strtol(tmp, NULL, 16)&0xff;
            tmp[0] = parsed_msg->data.data[5];
            tmp[1] = parsed_msg->data.data[6];
            b = strtol(tmp, NULL, 16)&0xff;
        } break;
        default: {
            return;
        } break;
    }
    ESP_LOGI(TAG, "Changing colors to {%d, %d, %d}", r, g, b);
    for (int i = 0; i < CONFIG_LED_COUNT; i++) {
        ESP_ERROR_CHECK(led_strip_set_pixel(led_strip, i, r, g, b));
    }
    ESP_ERROR_CHECK(led_strip_refresh(led_strip));
}
```

This callback function checks if the received string starts with a hash (#) symbol, which indicates the start of a hex color code. It then parses the string to extract RGB values, handling both three-character and six-character formats. Color validation is done only with a leading hash symbol check and a string length check. If the received string passes those checks, but does not represent a color, then the color will be set depending on the ASCII representation of received characters. The LED strip then changes to the requested color.

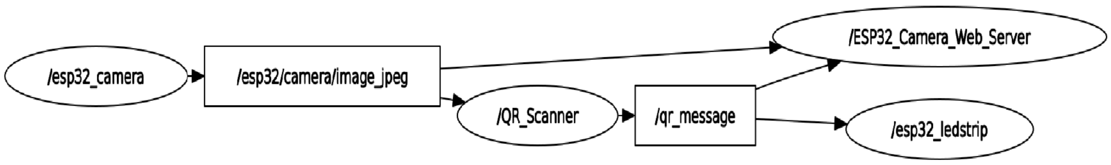
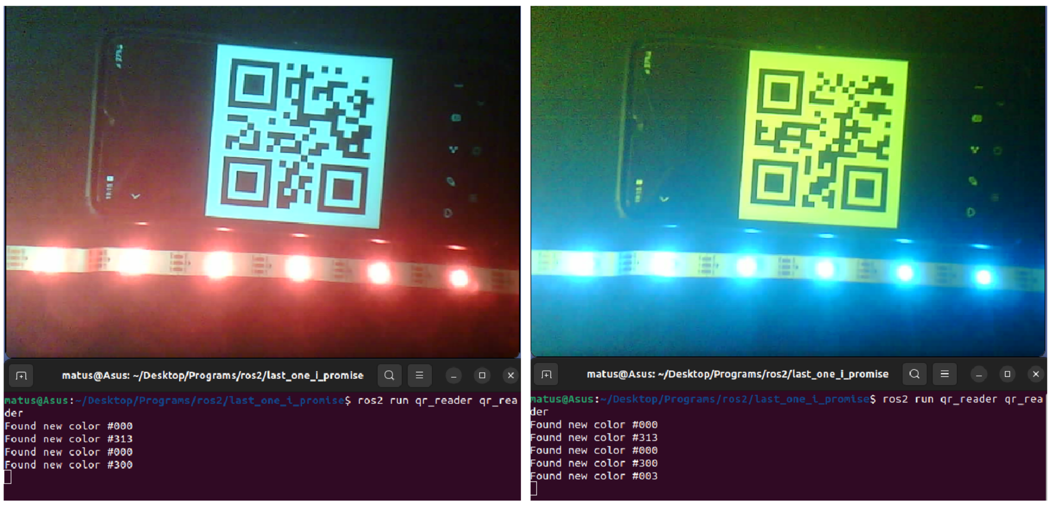


Figure 7.1: ROS graph of the generated by `rqt_graph`


matus@Asus: ~/Desktop/Programs/ros2/last_one_i_promise
matus@Asus: ~/Desktop/Programs/ros2/last_one_i_promise

```
matus@Asus:~/Desktop/Programs/ros2/last_one_i_promise$ ros2 run qr_reader qr_rea
der
Found new color #000
Found new color #313
Found new color #000
Found new color #300
```

```
matus@Asus:~/Desktop/Programs/ros2/last_one_i_promise$ ros2 run qr_reader qr_rea
der
Found new color #000
Found new color #313
Found new color #000
Found new color #300
Found new color #003
```

Figure 7.2: Screenshots of the working example

Chapter 8

Conculsion

In conclusion, the main objective of this thesis was the successful integration of ESP-IDF projects into the ROS 2 framework, enabling these projects to function seamlessly as standard ROS2 packages. This integration was designed to address the fragmentation encountered in managing projects that utilize ESP microcontrollers within a ROS 2 environment. This objective was accomplished through the development and deployment of the `esp_env` and `esp_uros` packages, which simplify the integration process and expand the potential applications of ESP32 microcontrollers in robotics. By accomplishing this, the thesis adheres to and fulfills its initial intention and formal assignment.

The integration of ESP-IDF projects into the ROS 2 framework posed several challenges. Merging two distinct build systems directly, even though both are based on the same technology, proved to be difficult enough. The implemented solution involved treating ESP-IDF projects as standalone projects within the ROS 2 packages, which utilized the modular nature of both systems to maintain a clear boundary while ensuring functional integration. This thesis has significantly deepened my understanding of the CMake build system and expanded my knowledge on robotics and distributed systems.

There are several future advancements to this thesis could focus on further refining. For exmaple the integration of the micro-ROS project within the ROS 2 architecture. A possible direction is the potential reconfiguration of the `esp_uros` package with the ESP-IDF toolchain to fully utilize colcon, the ROS 2 build system, which would create a more unified development environment. This would involve adjusting the colcon configuration to completely eliminate the need for the `idf.py` command, which would streamline the build process.

Bibliography

[1] *ESP-IDF Programming Guide v5.2.1 documentation* online. Available at: https://docs.espressif.com/projects/esp-idf/en/v5.2.1/esp32/api-guides/build-system.html. [cit. 2024-05-06].

[2] *Features and Architecture | micro-ROS* online. Available at: https://micro.ros.org/docs/overview/ROS_2_feature_comparison/. [cit. 2024-05-06].

[3] Quigley, M.; Conley, K.; Gerkey, B.; Faust, J.; Foote, T. et al. ROS: an open-source Robot Operating System. In: Kobe, Japan. *ICRA workshop on open source software*. 2009, vol. 3, 3.2, p. 5.