UNIVERSITY OF BOLOGNA

MASTER SCIENCE THESIS

Efficient Support for Deep Sleeping Modes in Embedded Systems: the Case of Zenoh-Pico

Author:
Andrea ZANNI

Supervisor:
Paolo BELLAVISTA

A thesis submitted in fulfillment of the requirements
for the degree of Master Science in Computer Science Engineering

in the

School of Engineering and Architecture

March 19, 2024

"The following anecdote is told of William James. [...] After a lecture on cosmology and the structure of the solar system, James was accosted by a little old lady.

"Your theory that the sun is the centre of the solar system, and the earth is a ball which rotates around it has a very convincing ring to it, Mr. James, but it's wrong. I've got a better theory," said the little old lady.

"And what is that, madam?" inquired James politely.

"That we live on a crust of earth which is on the back of a giant turtle."

Not wishing to demolish this absurd little theory by bringing to bear the masses of scientific evidence he had at his command, James decided to gently dissuade his opponent by making her see some of the inadequacies of her position.

"If your theory is correct, madam," he asked, "what does this turtle stand on?"

"You're a very clever man, Mr. James, and that's a very good question," replied the little old lady, "but I have an answer to it. And it's this: The first turtle stands on the back of a second, far larger, turtle, who stands directly under him."

"But what does this second turtle stand on?" persisted James patiently.

To this, the little old lady crowed triumphantly,

"It's no use, Mr. James—it's turtles all the way down.""

UNIVERSITY OF BOLOGNA

Abstract

School of Engineering and Architecture

Master Science in Computer Science Engineering

Efficient Support for Deep Sleeping Modes in Embedded Systems: the Case of Zenoh-Pico

by Andrea ZANNI

The Internet of Things (IoT) is an ever evolving and ever growing field where plenty of devices communicate with each other and transfer data to/from the Cloud via an infrastructure, usually a pub/sub. In this extensive field comes to rescue Zenoh, a pub/sub/query middleware for Cloud and IoT applications. It supports many interesting functionalities but in this thesis we focus on its lightweight counterpart for embedded devices - Zenoh-Pico. Zenoh-Pico is entirely written using the C programming language thus making it extremely lightweight and efficient, perfect for an embedded device. In this work I focus on the support to deep-sleeping mode for Zenoh-Pico on UDP unicast communications. The deep-sleeping modes are typical of embedded devices such as ESP32 or Zephyr boards and they are used to put the board in a kind of energy saving state.

In order to work with Zenoh-Pico I used PlatformIO, ESP-IDF, VSCode, and the board az-delivery-devkit-v4 ESP32. PlatformIO and ESP-IDF are used to manage the toolchain for the creation of a firmware to be put on the ESP32 starting from the code on VSCode. VSCode is used to program the code that will run on the board.

I carried out many experiments from my support to deep-sleeping mode on the ESP32. The main and most interesting results being the time and space efficiency of my support to deep-sleeping mode, and the fact that with just 3 seconds of deep-sleeping on a low duty cycle programme (about 6 seconds of activity - 3 seconds of sleeping) with a 250mAh battery, the battery gains about 2 weeks of autonomy with 3 seconds of deep-sleeping.

In conclusion, the deep-sleeping mode is of paramount importance for the embedded system world when those embedded devices are powered from an external battery limited in its capacity thus the importance of a middleware like Zenoh-Pico to support the deep-sleeping mode.

Acknowledgements

I would like to start the acknowledgement section by saying thanks to whom gave me the opportunity to do an Erasmus in Paris and then a proposal for an internship at ZettaScale Tech., my current supervisor **Paolo Bellavista**.

I want to thank the CTO of ZettaScale **Angelo Corsaro**, my informal co-supervisor which has always been fully available to me before, during, and after my internship at ZettaScale Technology. Then, I want to say thank you to **Carlos Guimãraes** for their support during my stay at ZettaScale and having helped me to set up the job about the deep-sleeping mode by telling me what was needed to be done for such feature and guided me in the internals of Zenoh-Pico. I want to thank **Pierre Avital** as his support has been crucial for the development of my feature at the second step. Also, gave me the idea of how to deal with user-defined struct. Thanks to **Julien Enoch** with whom I shared many rides by car from home to the workplace and we listened to very cool rock music at the radio. I say thank you to **Luca Cominardi** who, maybe unconsciously, unblocked me when I was stuck searching and looking for what to do on my internship for my master's thesis. I say thank you to all the other colleagues at ZettaScale Tech. for always being kind and professional to me, giving small advice or help when I was in need of.

Voglio ringraziare la mia famiglia, in particolare mia madre **Marta Tagliabracci**, mio padre **Marco Zanni** e mio fratello **Mattia Zanni**. Alla mia famiglia io dedico questo lavoro perché è grazie alla mia famiglia se ho intrapreso e mantenuto gli studi universitari, se sono riuscito ad andare a Parigi e se sono riuscito a terminare gli studi universitari nonostante i mille problemi che hanno rischiato di mandare a monte la mia intera carriera accademica.

Infine, ringrazio tutti i buoni amici che mi sono rimasti accanto, nonostante tutto, durante questi anni e continuano a volermi bene. Grazie.

Contents

List of Figures

List of Tables

Dedicated to my brother Mattia, my mother Marta, my father Marco, and the forever living in our hearts - my brother Nicola…

Introduction

In this work, first I expose the **motivations** and the **objectives** of this master thesis and I try to give a possible driving trend led by **pub/sub communications**, **location-transparent communications**, and **Named Data communications**. Then, I guide the reader in the **Internet of Things** world by giving a possible definition of IoT, some important concepts in the IoT such as Gateways, Edge Computing, constrained networks and constrained devices. At last, I present some request/response protocols and pub/sub protocols widely used in the IoT.

What I do next is to expose the peculiar characteristics of an **embedded system** and where I can find an embedded system in the society of today. I introduce the **cyber-physical loop** typical of an embedded system along with its typical **Harvard architecture**. I present the **ESP32 board** since it is an embedded system and I explain its logical architecture, then I delve deeper in the technical details of the board, starting with the name of the 2 CPUs, the clock sources, the memories, the supported hardware protocols like UART or I2C. After that, I give a brief description of the environments needed to operate on the ESP32 or generally speaking on a wide range of IoT boards. I explain the common **active-sleep pattern** found on Micro Controller Units (MCUs) then I give some theoretical formulas used to understand the necessity of minimizing the T_{CPU} (composed by the execution time and the idle time) in order to extend the battery lifetime. I conclude by presenting to the reader the company where I did my internship which is called ZettaScale Technology and is located at Saint-Aubin, France, that is a little city near Paris.

I introduce **Zenoh** that is a **framework**, a **middleware**, a **protocol** that unifies data in **motion**, data at **rest**, and **distributed computations** by blending pub/sub communications with geo-distributed storages, queries, and computations. Then I introduce **Zenoh-Pico**, the Zenoh counterpart for **embedded systems**. With Zenoh-Pico, Zenoh opens the doors to ubiquitous computing. I introduce the core concepts of the Zenoh middleware like the keys, data message format, router/client/peer, scouting, supported communication models, types of reliability, control flow, how mobility is tackled, the usage of an Hybrid Logical Clock. I give a rough idea of the **Zenoh architecture** before moving my focus on the **Zenoh-Pico's architecture**. There I explain how some properties like the platform-agnostic and the file inclusion are achieved along with its architecture as macro blocks by explaining what a folder contains and what is its job. Along with the description of the folder I explain the anatomy of a Zenoh packet and how the protocol behaves when it is opened a Zenoh session. Moreover, I explain the composition of an important abstraction buffer called IOSli buffer.

Ultimately, I discuss my personal project developed at ZettaScale Technology to efficiently support the deep sleeping modes of the board **az-delivery-devkit-v4 ESP32** on the Zenoh-Pico middleware. I support only **UDP unicast communications** on Zenoh-Pico. The supported modes on the ESP32 are **deep sleeping** and **hibernation** as long as for the latter is not powered off the **RTC Slow Memory**. The project consists in the following phases being saving the session in the RTC Slow Memory, going to deep sleep/hibernation, restore the session from the RTC Slow Memory

and bind the UDP unicast socket to the previously used port. I show the components of the session that must be **serialized** onto the RTC Slow Memory, I discuss the whys and drawbacks of not using a third-party library to serialize/deserialize the components of the session. I discuss why I chose the RTC Slow Memory to save the session on. I explain how I found the point where I had to edit the code in order to use the same UDP port among deep sleeps/hibernations. I explain how I restore the session. I explain the concepts behind the serde functions used to serialize/deserialize user-defined structs in the Zenoh session. In the **implementation** section I deeply explain how I implemented the two functions added to the Zenoh-Pico's API: the ***zp_prepare_to_sleep()*** to save the session in the RTC Slow Memory, and the ***zp_wake_up()*** to restore the session from the RTC Slow Memory and restore the UDP unicast transport. After the implementation details, I showcase the **most difficult errors** encountered while developing the support to the deep sleeping modes then I explain how I glued the serialization/deserialization (serde) functions with the rest of Zenoh-Pico. At last, I discuss the **experimental evaluations** conducted on my implementation starting with the baseline measurements then varying the load of the session and make comparisons w.r.t. baseline measurements. Also, I make further measurements about the size and the average power consumed by the board az-delivery-devkit-v4 ESP32 with a low duty cycle active-sleeping pattern.

Chapter 1

Thesis Motivations, Objectives, and Internet of Things

The **UDP communication** at the transport layer is the favorite one for what concerns the **Internet of Things (IoT)**. This is due to the **lossy** nature of the majority of the IoT communications since the communications are mostly done over **long range network** and the Access Point may be far away from the devices. Moreover UDP, being simpler than the TCP, is easier to implement and **occupies less memory surface** than TCP, a feature constrained devices benefit a lot from. However, it is not still clear which application protocol will succeed in the application layer. Will be MQTT+ (enhanced version of MQTT), HTTP (or its lightweight version CoAP), or **Zenoh**? Let us tell that Zenoh, apart from being an application protocol, can also dive deep in the ISO/OSI stack straight to the data link layer since it has the capabilities to use UDP or TCP, Multicast in the former case, IPv4, IPv6, or 6LoWPAN, and many more capabilities. It must also be said that Zenoh has the support for RESTful interfaces via HTTP to query and edit the Zenoh router and it provides a plugin to make Zenoh capable of interacting with MQTT. The minimum common denominator of MQTT+ and Zenoh is that both use a **publish/subscribe communication model**. The pub/sub communication model consists of a broker that manages subscriptions from devices to a certain topic and, when one or more device publish something on a topic, all the subscribers to that topic will receive the information associated with the topic. A particular scenario would be the usage of the pub/-sub communication model in the edge of the network. By edge of the network I intend that specific region of network behind a router whose role is the gateway i.e. a virtualized application runs on the gateway and performs data aggregation, data pre-processing, security (by implementing the TLS encryption between the gateway and the cloud), scalability, service discovery, geo-localization, billing, etc.

The **Named Data Networking (NDN)** is a proposed Internet architecture, it is

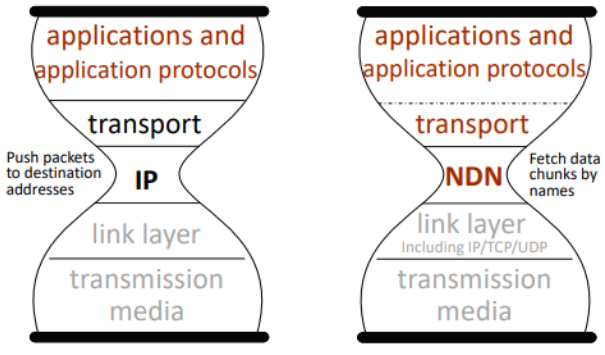


FIGURE 1.1: Hourglass model Named Data Networking (NDN)

designed to network the world of computing devices by **naming data** instead of naming data containers as IP does today (see figure 1.1). With this change, NDN brings a number of benefits to network communication, including built-in multicast, in-network caching, multipath forwarding, and securing data directly. NDN also enables **resilient communication** in intermittently connected and **mobile ad hoc environments**, which is difficult to achieve by today's TCP/IP architecture. In a nutshell, all a network does is ship data bits. It is the job of the network architecture to define how this data shipping is realized. A network architecture design makes two fundamental decisions: (1) what namespaces are used for data delivery, and (2) what are the specific mechanisms for the delivery. There are two main options for the first design decision: (1) name the locations to ship the bits to; (2) name the bits themselves. The second design decision depends on the first: if named by location, data bits to be sent from host A to host B may either be delivered along an established path between the two communicating endpoints (virtual circuit), or travel through the network as independent pieces to reach B (datagrams). If one names the data bits directly instead of by location, then a network delivers named bits (named data packets) to their requesters. Today's TCP/IP protocol architecture picked the first option of network namespace design, naming locations, the same communication model used by circuit-switched telephone networks. A telephone network assigns a phone number to each telephone set wired at a specific location and sets up a circuit between two calling parties. From telephony to IP networking, phone numbers are replaced by IP addresses, circuit switching by packet switching with datagram delivery, but the same location-based, point-to-point communication model remains. Named Data Networking (NDN), takes the second option of network namespace design, naming bits. As a proposed Internet architecture, NDN is designed to network the world of computing devices, ranging from IoT sensors to cloud servers, by naming data bits. Named data chunks make the centerpiece in the NDN network architecture, and the NDN network layer uses application data names to communicate. This design empowers the network to retrieve named data by any means necessary, treating networking, storage, and computing resources in the same manner and enables one to secure data directly [Afanasyev, 2018].

The advent of Zenoh brings the pub/sub/query model and the NDN to work together over the Internet Protocol (IPv4 and IPv6), 6LoWPAN, and many others. That means an user can retrieve data simply by naming the data she wants (i.e. "demo/room2/temperature") or she can activate an actuator by simply naming it (i.e. "demo/room3/radiator/on"). In the case of zenoh, the IPs are specified in special **configuration files** that interconnect the routers which serve the role of broker in the pub/sub infrastructure, they also serve the role to route data in a **location-transparent** way. This is important in the IoT field since data can be accessed or edited just by naming it without the knowledge of how to reach it and in an ever evolving and ever growing field like the IoT where a lot of number of devices are connected to each other this location-transparent approach is beneficial as multiple devices can retrieve information from other multiple devices simply with a name (i.e. "demo/room3/**") and multiple devices can subscribe to multiple topics with the usage of a single name per topic.

1.1 Motivations

While at **ZettaScale Tech.**, the company where I worked off for over 5 months on **Zenoh-Pico** for my master's thesis, I was lost at first thinking the answer to what

should have been done on Zenoh would have been found on some paper taken from Google Scholar i.e. a search engine for academic papers relevant for many fields especially Computer Science and Engineering. I was saying that I was searching for papers on Google Scholar about 5G and Multi-Access Edge Computing (MEC) in order to insert Zenoh on a 5G-MEC environment and make some tests. But I was missing something important. Earlier in time, on October 2021, I went to visit for the first time the offices of AdLink Tech. at Saint-Aubin, France 91190 (at that time ZettaScale Tech. was AdLink, ZettaScale was born on February 2022 and is a spin-off of AdLink Tech.). There I met for the very first time the AdLink Team. and I told them I was quite interested in **embedded devices**. On March 2022 I was still looking for an answer on what to do on Google Scholar and my scope was 5G-MEC. One day of that month of that year I had an interesting conversation with my colleague **Luca Cominardi** who noticed I was **stuck** somewhere and just told me it was time to start writing some code. Well. A few months before I told the whole team I was interested on **embedded systems** and all of a sudden I changed my scope to 5G-MEC, that was hindering my master's thesis, so I decided to take a step back, think about what was told to my colleagues i.e. I like embedded devices, and start writing some code. After all, my bachelor's thesis was about HTTP/3 (HTTP over QUIC over UDP) and its applications in a possible **Internet of Things scenario**, thus why not embedded devices? I started out by looking at the documentation for the ESP32, a board supported by Zenoh-Pico, the counterpart of Zenoh for constrained devices, and my interests lashed out against the **deep sleeping modes** supported by the board and how each sleep mode **decreased the usage of the energy**. There are many boards supporting deep sleeping which are also supported by Zenoh-Pico, think about the Zephyr board, but I chose the ESP32 as I was somewhat bounded to its predecessor, the ESP8266 because in the course Software Systems Engineering (which I left for a couple of reasons) there was the opportunity to play around with either a Raspberry Pi or an ESP8266 board (way cheaper than a Raspberry even though it has more constrained resources).

In **2024** and beyond it is becoming of paramount importance the support to deep sleeping for devices ran with an **external battery** with limited capacity that is why I strongly believe (and demonstrate with my experiment) that the support for the deep-sleeping mode on embedded systems for Zenoh-Pico is jovial in terms of battery's life.

1.2 Objectives

The **objectives** of this master's thesis are:

- to better understand the world of the **Internet of Things**.
- to better understand the **Zenoh middleware** and what a middleware is.
- to better understand what improvements a feature to support the **deep-sleeping mode** in Zenoh-Pico can bring in the field of the Internet of Things.

1.3 What is the Internet of Things

The Internet of Things is subject to many definitions but I will anyway try to give one definition: it is a **"thing"** equipped with sensors and actuators, uniquely identifiable in the network, **connected to the Internet** and capable of taking decisions

without the human intervention. The above definition is pretty general so let me delve deeper into the IoT world. Possibly every single device and object, "a thing", can be connected to the internet (around 50 – 100 billions in 2020). Their main task is to gather information from things for monitoring and control, send information back and forth, store and aggregate information, analyze information, take decisions in a human-assisted or autonomous manner. In the past the idea of remote monitoring and control was already existent, think about Supervisory Control And Data Acquisition (SCADA) systems. There was, and still there is, a central control system with sensors and actuators, controllers, communication equipment and software. Periodic reading of values and status of sensors to collect data. Typical deployments are gas and oil distribution, electrical power, water distribution, bus traffic system, airport, ... These are widely tested solutions, real-time, reliable but they are expensive, **extensible only within the same scope** e.g. additional sensor or an additional machine, lacking of standards, and **no interoperability with other SCADA systems** being horizontal or vertical plus no connection to the Internet. Let's see what is the IoT accordingly to the IEEE: "An IoT is a network that connects uniquely identifiable "Things" to the Internet. The Things have sensing/actuation and potential **programmability capabilities**. Through the exploitation of unique identification and sensing: information about the Thing can be collected and the state of the Thing can be changed from anywhere, anytime, by anything.". The **difference** between an **IoT** and a **SCADA** system, beyond the cons of the latter, is the **connection with the Internet**. Moreover, IoT interconnects Things to the Internet through the use of standard communication protocols, are uniquely identifiable, and the Thing offers services with or without human intervention through the exploitation of unique identification, data capture, communication, and actuation capability. For this to be true there are two important **technologies** considered **IoT-enabler** that are: **reduced hardware cost** and **hardware size**, **pervasive and cheap wireless communications** from cables to large bandwidth or wide-coverage wireless communication. Nowadays the IoT is constituted by one horizontal layer managing heterogeneous information in an efficient manner (like AWS IoT, Microsoft Azure) and several vertical applications to provide specific information in a market-tailored manner.The several heterogeneous things make the convergence of sensed data to the Internet via multiple gateways capable of communicating through both classic IP protocols (IP, TCP/UDP, HTTP) and the IoT-specific protocols (ZigBee, Bluetooth, Serial, LoRaWAN). The gateways are geographically close to sensors or actuators and directly interact with things or dispatch data to and from the Internet. The server-side remote applications are stored in the Cloud and manage data. The logical architecture in the figure 1.2 can be teared-down to:

- Things: any **physical** or **digital object** that should be monitored or controlled: physical objects must be digitalized, virtual objects must be standardized.
- Gateway: close to one or multiple things to interact with them and send data and command back and forth the Internet. The gateway can be seen as a **point of convergence** to access the internet.
- Communication protocol: wired/wireless technology to actually send bytes.
- IoT Platform: data storing and management, application of (simple) aggregation / processing functions on data. Data exchange protocol to standardize how information are transported (and eventually also represented, typically JSON).

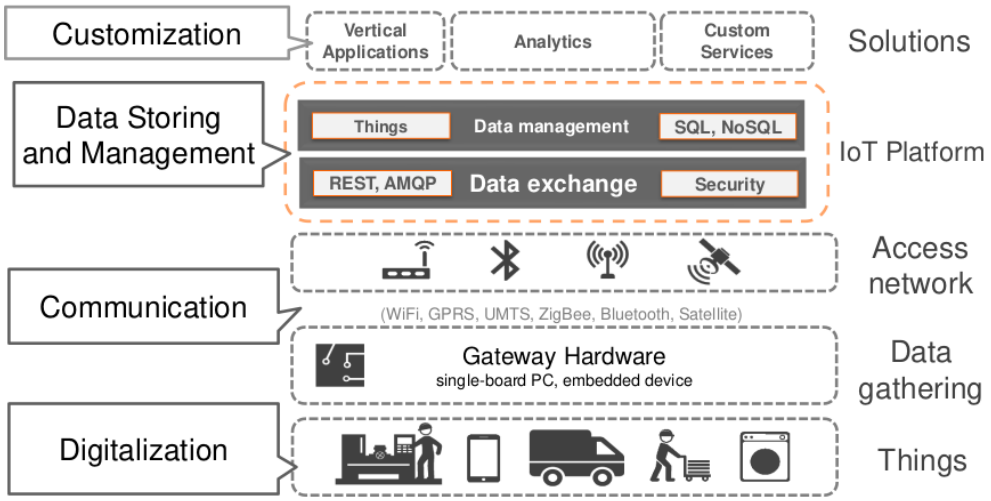


FIGURE 1.2: Logical Architecture for IoT

- Analytics: complex analysis on data to infer new knowledge.

1.3.1 The Gateway

It is important to better define the tasks of the gateway. The gateway provides **protocol translation** between peripheral trunks of the IoT, eventually provided with lower parts of the communication stacks. They may offer **pre-processing**, **security**, **scalability**, **service discovery**, **geo-localization**, **billing**, etc. With pre-processing we can intend data buffering i.e. temporarily store data to wait for connectivity or to increase efficiency, data efficiency i.e. temperature read every 1s but only per-minute average sent, data aggregation i.e. water level from different silos but only the sum is sent, data filtering i.e. send temperature values only if greater than 25°C. If there is no gateway, things have to send data on their own. In case of **constrained devices** there might be a **reduced set of capabilities** e.g. no security since cryptography is CPU-intensive, no data buffering, filtering aggregation, no programmability, etc.

1.3.2 IoT Fog/Edge Computing

With the gateways we have seen the first evolution of the **IoT Cloud Computing** architecture where most of the computation is on the Cloud, only gateways are deployed close to things, gateways perform few and simple tasks. The second evolution wave, **IoT Fog/Edge Computing** architecture, adds relatively powerful devices, it is closer to things but between gateways and the Cloud, complex analytical tasks are performed on the client side before sending data to the cloud. From now on I will refer to Fog/Edge computing as Edge computing. In an IoT Edge Computing architecture the use of Docker, VMWare, generally speaking the use of **virtualization** can be used on On-Premise servers that means Edge computing. The on-premise server is located between the gateway and the Cloud. We can think of Edge Computing as it extends the cloud to be **closer to** the things that produce and act on IoT data (Cisco). Edge computing allows to minimize latency, conserve network bandwidth, address security concerns in transit and at rest, move data to the best place for processing. The Edge computing is to be considered when data is collected at the extreme edge (vehicles, ships, factory floors, roadways, railways, etc.), thousands or millions of

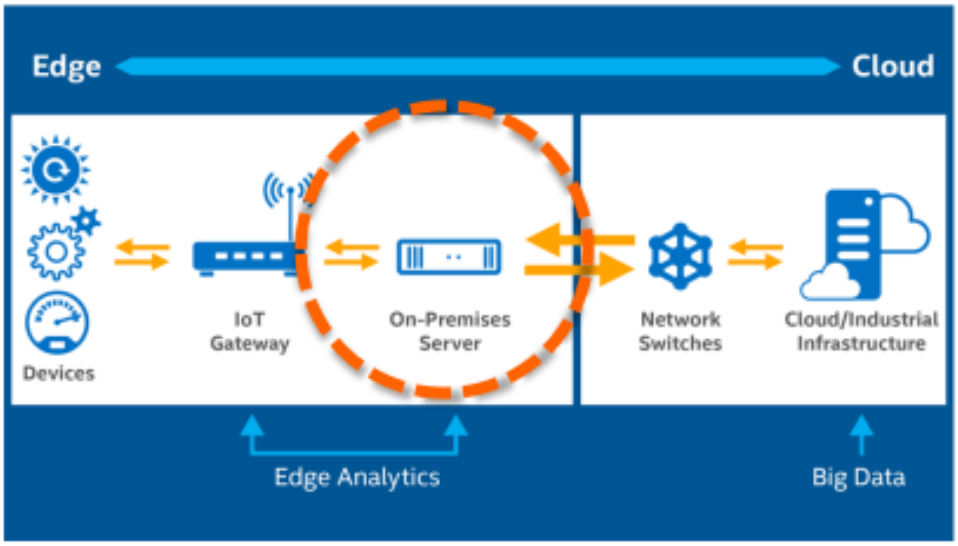


FIGURE 1.3: Fog/Edge Computing

things across a large geographic area are generating data, it is necessary to analyze and act on data promptly, in less then a second (see figure 1.3).

1.3.3 Constrained networks and devices

Constrained devices are "A node where some of the characteristics that are otherwise pretty much taken for granted for Internet nodes at the time of writing are not attainable, [...] due to **cost**, **size**, and **energy constraints**". Significant constrains on maximum code complexity (due to ROM/Flash memory), size of state and buffers (RAM), available computational power, connectivity. Later in this chapter I will discuss the constrained device **ESP32**, the one used for this thesis.

Constrained networks are "A network where some of the characteristics pretty much taken for granted with link layers in common use in the Internet at the time of writing are not attainable". There are significant constraints on **low achievable throughput**, **high packet loss**, **highly asymmetric links**, **severe penalties for using larger packets**, **limits on reachability over time**.

1.3.4 Wireless communication protocols for the IoT

Wired and wireless protocols are used to transport bits and messaging protocols are used to transfer data. **Data is information** and **commands described following a given syntax and semantic** whereas messaging protocols to exchange data encapsulate data and transmit it via wired or wireless protocols. There are several wireless protocols for IoT, see figure 1.4 for an overview of the wireless protocols for IoT.

Data exchange protocols

There are two method to establish who should initiate the communication from gateways to servers:

- **push**: gateways autonomously decide to send messages to servers e.g. when new sensed values are available.

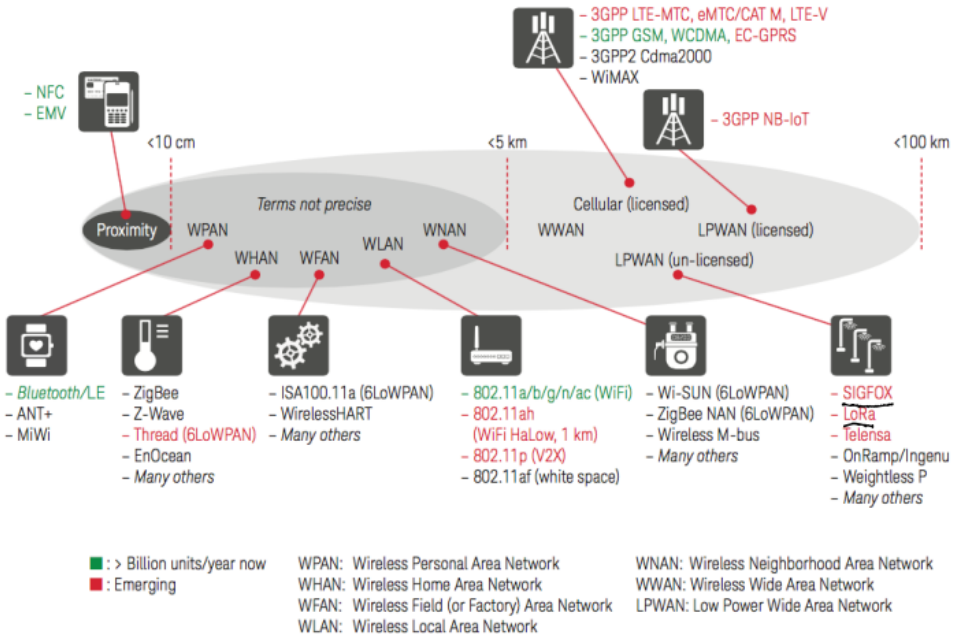


FIGURE 1.4: Wireless Protocols for IoT

- **pull**: servers ask to gateways to send messages e.g. only when servers actually require sensed values.

The two primary models to accomplish data exchange protocols are the **request/response** which can be push or pull and the **publish/subscribe** that usually is only push.

There are several protocols such as REST/HTTP, CoAP,... as request/response and MQTT, AMQP, DDS, ... as publish/subscribe.

Request/Response model

A client application **requests services** (e.g., send data, require data, perform an addition) and a **server application responds** to the service request (e.g., by providing the data or the addition results), direct communication between client and server, data exchange only if the client starts the communication and the server is not able to contact the client autonomously. Typical interactions in the IoT scenario are push: sensors send data to servers, pull: actuators request to servers new configurations.

REST (REpresentational State Transfer)

REST is not an actual protocol but substantially a solution architectural style. It promotes client/server and **stateless** interaction, oriented to the usage of **caching opportunities** also with possibility of code-on-demand to clients. It is **usually based on HTTP**, the protocol used to surf the web. REST is very simple but each time the client has to start the communication from the beginning.

URI (Uniform Resource Identifier) is used to identify the remote resource: any resource has a persistent identified, **do not transfer resources but their representations** via HTTP, example of an endpoint for a service managing user information: www.examples.com/resources/users. HTTP methods are used to interact with remote resources in a standard manner. GET to retrieve a specific resource (by ID) or

a collection of resources, PUT to create a new resource, POST to update a specific resource by ID, DELETE to remove a specific resource by ID. Usually it is used JSON for formatting and serializing data.

Constrained Application Protocol (CoAP)

Designed for **M2M and IoT applications** such as smart-metering, e-health, building and home automation with constrained devices e.g. 8-bit microcontrollers with 16KB of RAM and battery operated, constrained networks e.g. Wireless sensor networks. CoAP is a low-overhead **request/response protocol** that also supports discovery of services and resources. It is based on UDP communications with multicast. Inspired by and compatible with the HTTP protocol and REST architectures: CoAP is a specialized web transfer protocol.

CoAP features a **web RESTful protocol** fulfilling M2M requirements in constrained environments, simple request/response HTTP mapping to access CoAP resources via HTTP, URI and content-type support (a sensor is defined by an URI), low header overhead and parsing complexity, security binding to Datagram Transport Layer Security (DTLS), UDP binding with optional reliability supporting unicast and multicast, asynchronous message exchanges, services and resources discovery, publish/subscribe, simple caching.

Publish/Subscribe model

The **pub/sub model pattern** is an **alternative to** the traditional **client-server model**, where a client communicates directly with an endpoint. The roles are well defined and are 3: **publisher** that is a client sending a message, **subscriber** that is one or more receivers waiting for the message, **broker** that is the central component receiving and distributing messages to interested receivers.

The **broker** routes messages based on:

- message **topic**: a subject, part of each message. Receiving clients subscribe on the topics they are interested in with the broker and from there on they get all message based on the subscribed topics.
- message **type**: depending on the type of the message.
- message **header**: depending on a set of fields of the message.
- message **content**: possibly depending on the whole message content (expressive but expensive).

With the pub/sub model we experience 3 decouplings which are **space decoupling** i.e. publisher and subscriber do not need to know each other, **time decoupling** i.e. publisher and subscriber do not need to run at the same time, **synchronization decoupling** i.e. operations on both components are not halted during publish or receiving.

Event system as logically centralized system provide anonymous communication, possibility to use filters (on headers or entire messages), basic primitives such as subscribe, unsubscribe and publish.

Message Queue Telemetry Transport (MQTT)

MQTT is the evolution of the WebSphere MQ developed by IBM. MQTT is simple, lightweight, broker-based, pub/sub, open messaging protocol. Ideal for use in

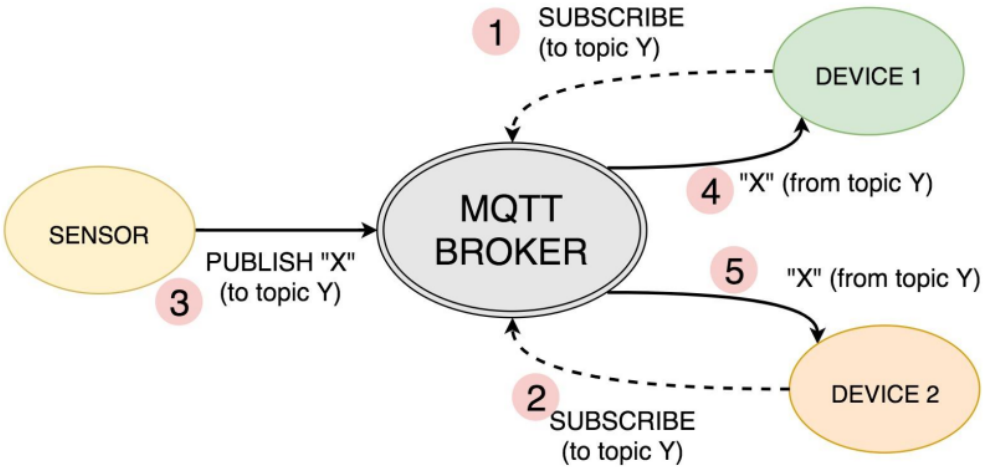


FIGURE 1.5: MQTT Architecture

constrained nodes and networks like **embedded devices** with limited processor or memory resources, where the network is expensive has low bandwidth or is unreliable. Then we have MQTT-SN for wireless sensor networks, aimed at embedded devices non non-TCP/IP networks (such as Zigbee).

The features of MQTT are **pub/sub pattern** to provide one-to-many message distribution, decoupling applications (see figure 1.5). TCP/IP is used to provide basic network connectivity. Transport overhead of only 2 bytes and protocol exchanges minimized to reduce the network traffic. It is easy to use with few commands and messages are retained that means and MQTT broker can retain a message that can be sent to newly subscribing clients. The Quality of Service has three delivery semantics i.e. **at least once**, **at most once**, **exactly once**. Client subscriptions remain active even in case of disconnection. Subsequent messages with high QoS are stored for delivery after connection establishment. A client can setup a will that is a message to be published in case of unexpected disconnection e.g. an alarm.

Advanced Message Queuing Protocol (AMQP)

AMQP has a **richer semantic than MQTT** e.g. supports topics and queues but it is also heavier than MQTT e.g. **the broker is more complex**. Among the AMQP implementations we have Apache Qpid that focuses on providing several features like queuing, message distribution, security, management, clustering, federation, heterogeneous multi-platform support (most of them possibly not essential in the IoT scenario).

Data Distribution Service (DDS)

DDS is **pub/sub** but is **broker-less** based on **multicast**. It is scalable, real-time, dependable, high performance and interoperable. Proposed in several mission-critical environments where performance and reliability are essential (see figure 1.6).

Industrial data exchange frameworks

In the following are showcased the main frameworks used in the industry.

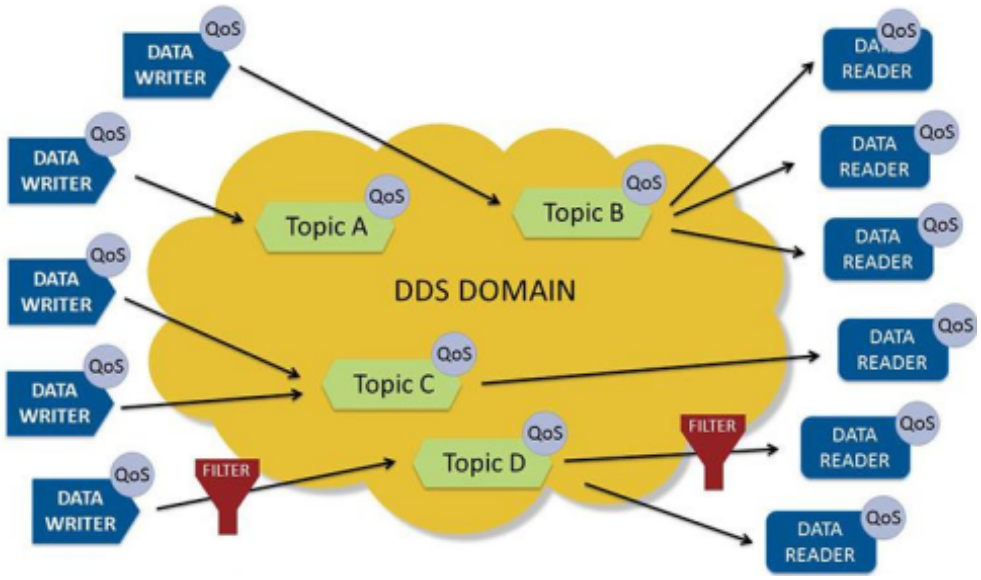


FIGURE 1.6: DDS

RabbitMQ

RabbitMQ actually is not an IoT platform but an interesting **messaging infrastructure**. It has robust messaging for applications, reliable message delivery, distributed deployment as clusters for high availability and throughput, federate across multiple availability zones and regions, multi-protocol support like AMQP, MQTT, HTTP, etc., managing and monitoring via HTTP-API, command line tool, and UI. RabbitMQ runs on all major operating systems, supports a huge number of developer platforms, open source and commercially supported. Exchanges (topics) and queues can be durable i.e. capable of surviving a broker restart (as opposed to transient). Messages can be defined as persistent i.e. only persistent messages survive exchange/queue/brokers/channel failures. The synchronization has an automatic acknowledgement model i.e. verify that the message is actually delivered to the application waiting for it, it also has an explicit acknowledgement model i.e. wait for explicit acknowledgement sent back by the application immediately at message reception, after processing.

MTConnect

Mostly used in the USA, it is a protocol/platform specialized for data exchange between shop floor equipment and software applications, over networks using the Internet Protocol (IP). MTConnect is **lightweight**, **open**, **extensible**, and **read-only** as it was introduced only for the monitoring of numerically controlled machine tools. MTConnect exploits several internet open standards, XML format for data, RESTful interface via HTTP for communication, Ligtweight Directory Access Protocol (LDAP) for discovery services.

OPC Foundation

OPC = OLE (Object Linking and Embedding) for Process Control. OPC Foundation is a Microsoft software architecture for industrial control systems. It was

designed for **connecting Windows based PC with PLC and SCADA systems** in industrial automation. Based on COM and DCOM, so formerly tightly related to Microsoft technology. It presents a client/server architecture where client and server can be on the same or different machines. The interface between OPC Server and a specific PLC is always vendor-specific: typically each PLC producer sells also its specific OPC Server component. The **OPC Unified Architecture (OPC UA)** is a modern release of OPC, greatly differing from the previous one. It is platform independent: PC, ARM, Cloud, Windows, Linux, Android, it is secure as it allows for **authentication**, **encryption**, **auditing**, **fault tolerance**, **extensible** as it is multi-layered architecture of OPC UA that provides a future proof framework.

The whole content and images of these subsections are taken from the slides of the professor [Bellavista, 2023].

Chapter 2

Embedded Systems

Nowadays embedded systems are everywhere, from your smartphone to your smartwatch passing through consoles (PS4) or voice assistants such as Alexa. The Internet of Things refers to a world in which a large range of objects are **addressable via the network**. These objects can be the above already mentioned plus washing machines, fridges, bridges, railways, wearable devices, medical devices, and possibly everything in the world. An IoT application can be described as a cyber-physical loop (see figure 2.1) in which **information** is processed by the CPU, such information can be either displayed or to be used to control actuators that modify the surrounding environment. Moreover, there can be sensors that sense parameters like heat, humidity, CO_2 of the surrounding environment and feed these data to the CPU in order to be processed and either be displayed or used to control actuators which close the loop.

The architecture commonly used on embedded systems is the **Harvard Architecture** (see figure 2.2). This architecture allows for a **parallel fetch** and **store** that means separate instruction bus and data bus can make the programme do a read from the instruction memory in parallel with a write to the data memory. This was not possible in the **Von Neumann's architecture** as it uses the same memory unit for both data and instructions. The program is saved in the Read-Only Memory (ROM) so the board knows what to do when turned on or in case of a crash.

This section has references to the slides of the professor [Benini, 2023].

2.1 Espressif ESP32

The Espressif ESP32 is a **versatile** development board produced by Espressif Systems. The board is broadly used by a plethora of **IoT applications**. Here I describe the architecture, its functionalities, the technical details, and at last the two main environments to develop, cross-compile, and deploy software on an ESP32 which are PlatformIO and ESP-IDF.

The ESP32 development board presents the following logical architecture (figure 2.3):

It consists of 5 different macro-areas, to be specific:

- Core and Memory: contains the microprocessor Xtensa LX6 32-bit dual-core, the ROM and the SRAM.
- RTC and low-power management subsystem: contains the subsystem for the low power modes supported by the microprocessor.
- Peripheral interfaces: contains all the supported peripheral interfaces for I/O, the temperature sensor, the touch sensors, a Digital-to-Analog converter and an Analog-to-Digital converter.

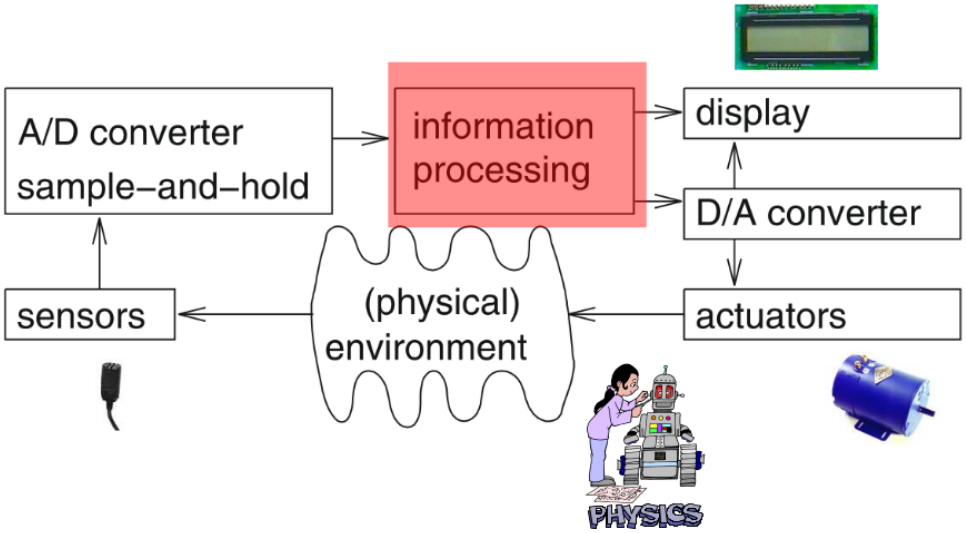


FIGURE 2.1: The cyber-physical loop

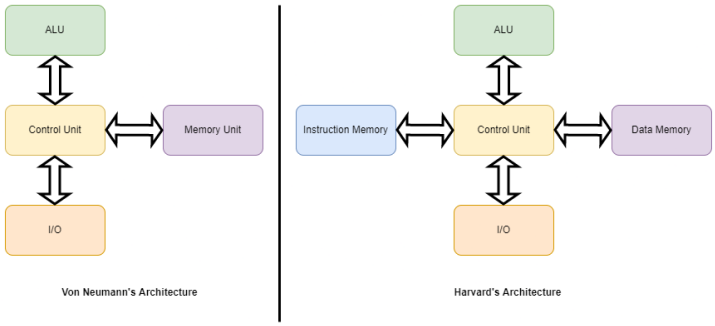


FIGURE 2.2: Von Neumann - Harvard

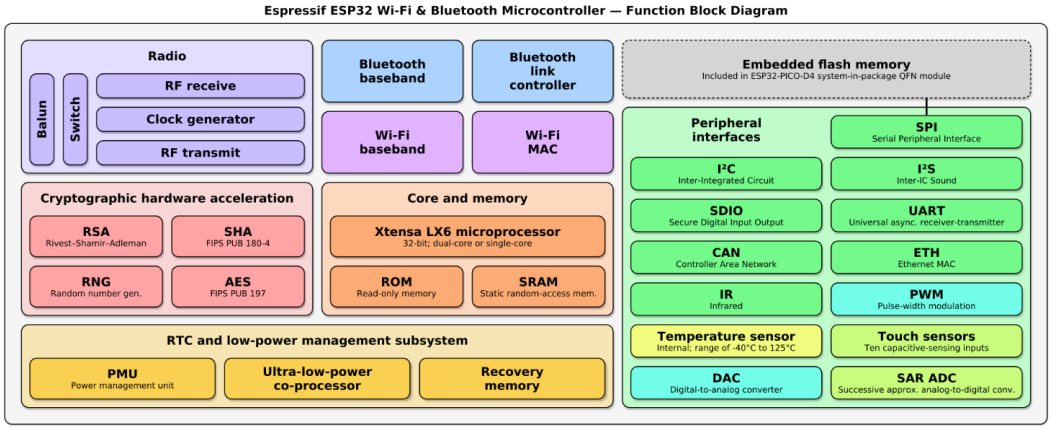


FIGURE 2.3: ESP32 - Logical Architecture

- Cryptographic hardware acceleration: contains the acceleration hardware for cryptographic purposes.
- Radio: contains the system for radio transmitting and receiving.

Among the just discussed macro-areas there is support for Bluetooth and WiFi.

2.1.1 Technical Details

The Espressif ESP32 development board has a **dual-core** Xtensa LX6 32-bit RISC microprocessor @ 160MHz up to 240 MHz, 512 KiB of SRAM, 4 MiB of flash, ROM 448 KiB, 8 KiB of RTC Slow Memory, and 8 KiB of RTC Fast Memory. One core is called **PRO_CPU** (core 0), the other one is called **APP_CPU** (core 1). As the core's name suggests, usually the core 0 is used by protocols activities tasks such as Bluetooth or WiFi whereas core 1 is used by the application logic but every core can be used interchangeably by assigning to a task a particular affinity via the function xTaskCreatePinnedToCore which takes as a parameter the core number where the task must run on. The affinity can be tskNO_AFFINITY so the task can run on both cores.

Clock Sources

The ESP32 integrates **multiple clock sources** for the CPU, peripherals and the RTC. These clocks can be configured to meet different requirements. Some are High Speed clocks, other one are low power clocks. For what concerns low power clocks, XTL32K_CLK is a clock with a frequency of 32 Khz, RC_FAST_CLK has a default frequency of 8 MHz, the RC_FAST_DIV_CLK has a frequency of RC_FAST_CLK divided by 256, RC_SLOW_CLK has a frequency of 150 Khz, and RC_SLOW_CLK is an internal low power clock with a default frequency of 150 Khz. The RTC_SLOW_CLK is used to clock the Power Management module, hence to clock the RTC Slow Memory and RTC Fast Memory, and is sourced by RC_SLOW_CLK, XTL32K_CLK or RC_FAST_DIV_CLK whereas RTC_FAST_CLK clocks the On-chip sensor module and is sourced by XTL_CLK or RC_FAST_CLK. By default, are used RC_SLOW_CLK and RC_FAST_CLK as clock sources for RTC_SLOW_CLK and RTC_FAST_CLK respectively (see figure 2.4).

RTC Fast Memory and RTC Slow Memory

RTC Fast Memory is **8 KiB** of SRAM and can be **read and written only** by **PRO_CPU** at an address range of 0x3FF8 0000 – 0x3FF8 1FFF on the data bus or at an address range of 0x400C 0000 – 0x400C 1FFF on the instruction bus. The two address ranges of PRO_CPU access the memory in the same order, for instance a load or store from the address 0x3FF8 0000 accesses the same word from a load or store from the address 0x400C 0000. It is about 10 times faster than the RTC Slow Memory. **RTC Slow Memory** is **8 KiB** of SRAM and can be accessed **either by PRO_CPU or APP_CPU** in read or write at an address range of 0x5000 0000 – 0x5000 1FFF from the instruction/data bus.

Universal Asynchronous Receiver Transmitter (UART)

A serial communication protocol is to be used when it is not possible to move data in parallel either because of physical or cost terms. Serial protocols are meant for short distances, they have low complexity, low cost, low speed.

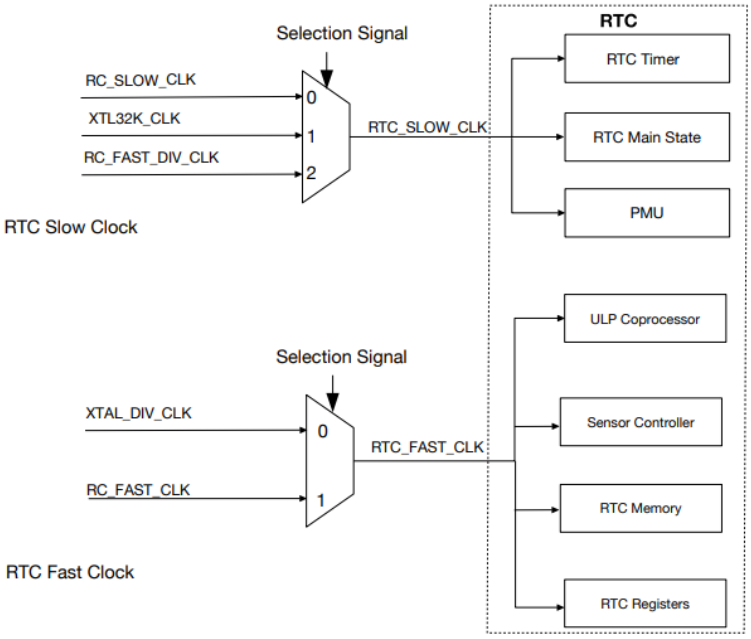


FIGURE 2.4: RTC Clocks

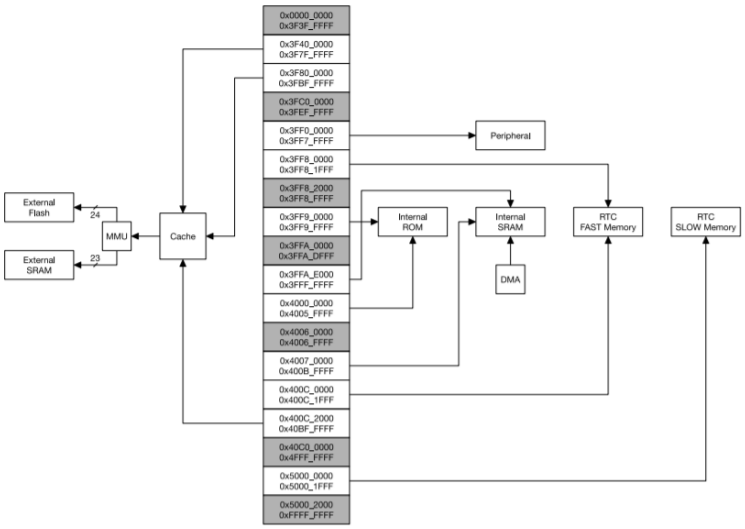


FIGURE 2.5: RTC addresses

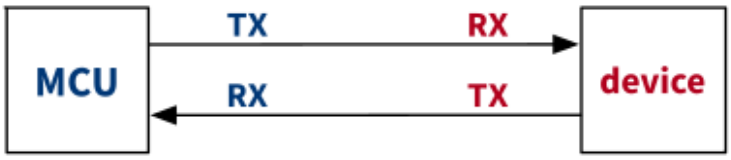


FIGURE 2.6: UART

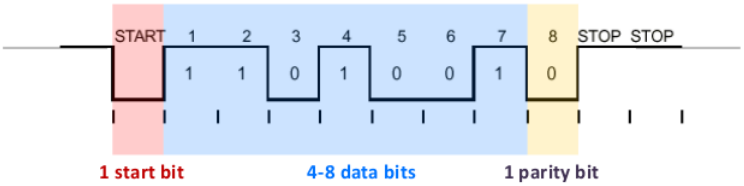


FIGURE 2.7: UART Protocol

Sometimes also found as Universal Synchronous Asynchronous Receiver Transmitter (USART), is used to **interface MCUs with other computing devices** such as other processors, a PC, interface microcontroller with **others transmission bus** as RS232, USB, CAN BUS, etc..., used to connect MCU with modems and transceivers as telephone modems.

It is essentially a **parallel2serial** in transmission and **serial2parallel** upon reception converter couple e.g. using a shift register for parallel2serial conversion. It is asynchronous as there is no common clock shared, each device has its **own local clock** typically running faster than the bit rate. The phase of the receiver clock is locked onto the edge of the transmitted data. UART is highly configurable for instance parity or no parity bit, data framing (e.g. number of stop bits, number of payload bits), communication simplex/full-duplex/half-duplex (see figure 2.6). The wording "8-N-1" printed when used UART as transmission and reception means 8 bits of payload, No parity bit, 1 stop bit.

Let us see how the UART protocol works. At first, (1) in **idle** the transmission line is driven to 1. The (2) transfer begins with a **start bit** where the transmission line is driven to 0 for one clock. Then, (3) a symbol of 5 to 9 bits is transmitted, most often 8 bits (i.e. an ASCII character), the symbol size is defined by the application and known a-priori w.r.t. the communication. One of the data bits can be used for (4) parity. Finally, (5) 1 or 2 stop bits where the transmission line is brought back to 1 and are used 1 or 2 stop bit depending on the application. Usually, 2 stop bits are used to realign the sampler (see figure 2.7).

The UART communication speed is defined by its **symbol rate** measured in **baud** where 1 baud is 1 symbol per second. In UART, a symbol has two values (0/1) hence 1 bit. This number includes both data payload and protocol bits (e.g. parity, framing) - this number is also called physical or gross bit rate. To be remembered that in some devices (e.g. modems) one symbol might correspond to more bits that means the baud rate is not the same as the gross bit rate. The baud rate of the ESP32 to communicate with the terminal of VSCode is to be set to **115200**. The ESP32 has integrated on its board the USB2UART and UART2USB converter.

The UART protocol can also include a **handshake**. First, a Request-to-Send (RTS) signal from the MCU to the device means MCU can accept new data. A Clear-to-Send (CTS) signal from the device to the MCU means that the device can send new data. The signals have a dual meaning if send from the other point of view. Exchange happens when CTS and RTS are both asserted.

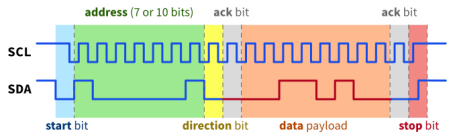


FIGURE 2.8: I2C

Inter-Integrated Circuit (I2C)

Usually pronounced "I-Squared-C", it was introduced by Philips in 1982. It is used for communication with **external peripherals**, for example: EEPROMs, thermal sensors, real-time clocks. Also used as a control interface for signal processing devices with separate data interfaces, for example: radio frequency tuners, video decoders and encoders, audio processors. I2C has three supported modes being **slow** (under 100 kbps), **fast** (400 kbps), **high speed** (3.4 Mbps) in I2C v2.0. The maximum inter-IC distance is of about **3 meters** for moderate speeds, the distance decrease for higher speeds. The communication protocol can support multi-master mode for complex applications where the communication is always started by a master, both in single-master and multi-master mode. The half-duplex synchronous communication scheme is where the master of the communication generates the clock (SCL) on which data (SDA) is synchronized.

I2C is based on two lines: serial clock (SCL) and serial data (SDA). Two pull-up resistors connected respectively to Vdd on the lines SCL and SDA. In idle, (0) **both SCL and SDA lines are pulled-up to 1**. (1) To start the communication, the master asserts the start bit that means the SDA bit is brought from 1 to 0 while SCL remains to 1. Then, the master starts generating the SCL clock. Except for the start and stop bits, **SDA has transitions only when SCL is 0**. (2) The master transmits the slave address that is broadcasted to all devices on the I2C bus, the slave address is used to select the target slave and the address ranges from 7 bits up to 10 bits (newer devices - 7 bits address space is small). (3) The master transmits a direction bit, if this bit is 0 the transfer is master to slave (a write - w.r.t. the master)), if this bit is 1 the transfer is slave to master (a read - w.r.t. the slave). In our case we assume a write transfer (i.e. bit set to 0 - a write). (4) The slave then acknowledges the reception by driving SDA to 0. If not acknowledged, the transaction must be repeated by the master. (5) The master transmits its data payload, each payload packet is 8 bits. There might be more than one packet, depending on the application. (6) The slave acknowledges the reception of each data packet (1 ack bit every 8 bit of payload). (7) At the end of the transfer, the master transmits a stop bit: first, it sets SDA to 0, then it releases SCL (it let the SCL signal go to 1), finally it releases SDA which goes to 1. It is done at the same clock cycle (see figure 2.8). The reads (w.r.t. the master) work similarly, but data transfer and ack roles are reversed: the slave drives the SDA when transmitting data byte, the master acknowledges the transfer. Note that the clock is always generated by the master.

A slave can ask for more time to process a bit by clock stretching where the slave drives the SCL to 0 if in need of more processing time.

Serial Peripheral Interface (SPI)

SPI was introduced by Motorola for the MC68HCxx line of microcontrollers. The use cases are generally similar to I2C but it is **generally faster than I2C** (up to serveral Mbit/s). Single-master, multiple slave architecture it is needed **one chip select per**

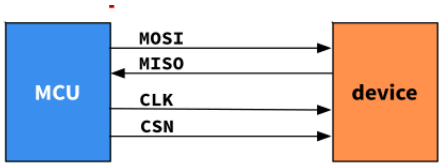


FIGURE 2.9: SPI

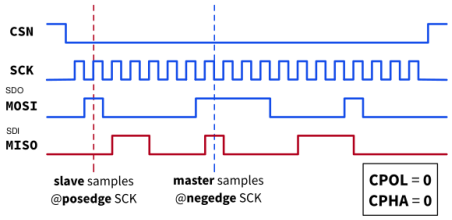


FIGURE 2.10: SPI CPHA = 0

slave device. The communication scheme is full-duplex synchronous - the master drives the clock (SCLK or CLK) with a clock polarity (i.e. write/read edges) and phase depending on the specific application.

SPI is based on two data and two control lines (see figure 2.9): MISO (master in, slave out data), MOSI (master out, slave in data), CLK (clock), CSN (chip select, one per slave, usually active low). The names are not standard, you might encounter SDI (SPI data in) instead of MISO, SDO (SPI data out) instead of MOSI, SCLK/SPC instead of CLK, CCS/SS/SSN instead of CSN.

The full duplex transfer means that data is streamed between master and slave shift registers/FIFO buffers: the master pushes the content of its buffer to the slave via MOSI, the slave pushes the content of its buffer to the master via MISO.

SPI has four operating modes varying by clock polarity (CPOL) and phase (CPHA): polarity sets the initial value of the SPI clock signal, phase defines the edge at which MOSI is switched and the one at which MISO is sampled (see figures 2.10, 2.11, 2.12).

The master is completely in charge of the transfer i.e. no ack no clock stretching contrarily to I2C. A more complex behavior than simple data streaming can be mapped on top of SPI protocol.

For point2point, SPI is **simple and efficient for the less overhead than I2C due to the lack of addressing**, plus SPI is full-duplex. For multiple slaves, each slave needs separate slave select signal then **SPI requires more effort and more hardware than I2C**.

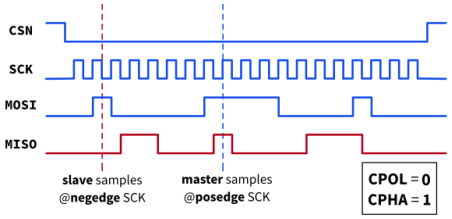


FIGURE 2.11: SPI CPHA = 1

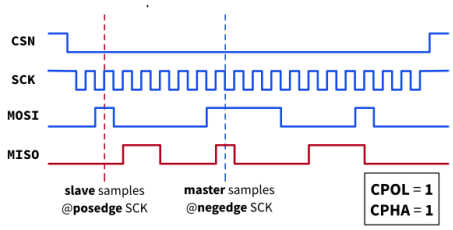


FIGURE 2.12: SPI CPOL = 1, CPHA = 1

Name	Bits	Direction	Meaning
PADDR	32	Initiator to Target	Address of peripheral
PWRITE	1	Initiator to Target	1 = store, 0 = load
PSEL	1	Initiator to Target	1 for selected slave during the full transaction
PENABLE	1	Initiator to Target	1 during *ACCESS* phase
PWDATA	32	Initiator to Target	Data to be stored
PRDATA	32	Target to Initiator	Data loaded
PREADY	1	Target to Initiator	1 during *ACCESS* phase terminates transaction

FIGURE 2.13: APB Names

Advanced Peripheral Bus (APB)

It is the peripheral bus from which the RTC Slow RAM is accessed. It is a **low-cost interface optimized for power consumption and reduced interface**. It is fully synchronous, every transfer takes at least two cycles, two phases: SETUP and ACCESS. It is a simple protocol designed for peripherals (see figure 2.13).

The subsection and images about UART, I2C, SPI, and APB are taken from the slides of the professor [Conti, 2023].

Sleep Modes

- **Light-Sleep mode**: digital peripherals, most of the RAM, and CPUs are clock-gated and the supply voltage is reduced. Upon exit from this mode, the digital peripherals, RAM, and CPUs resume operation and their internal states are preserved
- **Deep-Sleep mode**: the CPUs, most of the RAM, and all digital peripherals that are clocked from APB_CLK are powered off. The only parts of the chip that are powered on are: RTC controller, ULP coprocessor, RTC fast memory, RTC slow memory.
- There is also a third sleep mode called **Modem sleep** which, if needed, only affects WiFi in station mode but is treated here for completeness' sake: the station will switch between active and sleep state periodically after connecting to AP successfully. In sleep state, RF, PHY, and BB are turned off in order to reduce power consumption.

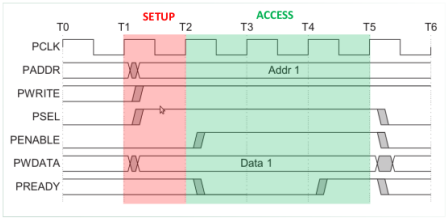


FIGURE 2.14: APB Write Transaction

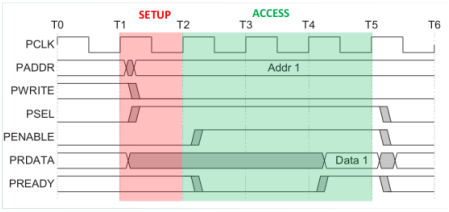


FIGURE 2.15: APB Read Transaction

Power mode	Description			Power consumption
Modem-sleep	The CPU is powered on.	240 MHz*	Dual-core chip(s)	30 mA ~ 68 mA
			Single-core chip(s)	N/A
		160 MHz*	Dual-core chip(s)	27 mA ~ 44 mA
			Single-core chip(s)	27 mA ~ 34 mA
		Normal speed: 80 MHz	Dual-core chip(s)	20 mA ~ 31 mA
			Single-core chip(s)	20 mA ~ 25 mA
Light-sleep	-			0.8 mA
Deep-sleep	The ULP coprocessor is powered on.			150 μA
	ULP sensor-monitored pattern			100 μA @1% duty
	RTC timer + RTC memory			10 μA
Hibernation	RTC timer only			5 μA
Power off	CHIP_PU is set to low level, the chip is powered off.			1 μA

FIGURE 2.16: ESP32 - Sleep Modes

A particular sleep mode is called **hibernation** which keeps powered on only the RTC Timer and consumes less power than Deep-Sleep mode. Such mode is achieved by manually, **programmatically speaking**, powering off all the sections except the RTC Timer.

Memory model

In this subsection I am going into the details of the **memory model** for the ESP32.

The diagram in figure 2.17 shows the ESP32 internal memory (SRAM) layout. The SRAM is divided into 3 memory blocks being **SRAM0**, **SRAM1** and **SRAM2** (and two small blocks of RTC fast and slow memory which we'll consider separately later). The SRAM is used in two ways — one for instruction memory — IRAM(used for code execution — text data) and data memory — DRAM (used for BSS, data, heap). SRAM0 and SRAM1 can be used as a contiguous IRAM whereas SRAM1 and SRAM2 can be used as a contiguous DRAM address space (see figure 2.18). While

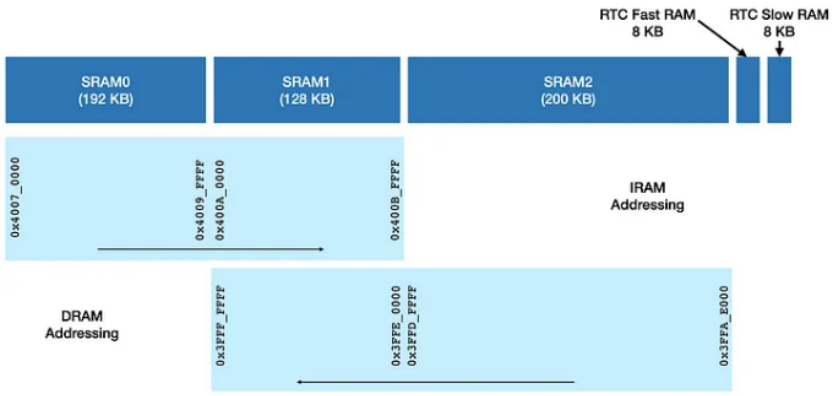


FIGURE 2.17: Internal RAM layout

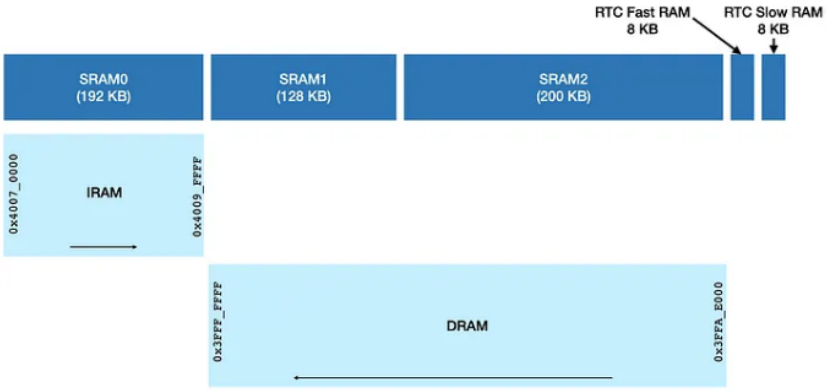


FIGURE 2.18: Programmer's memory map

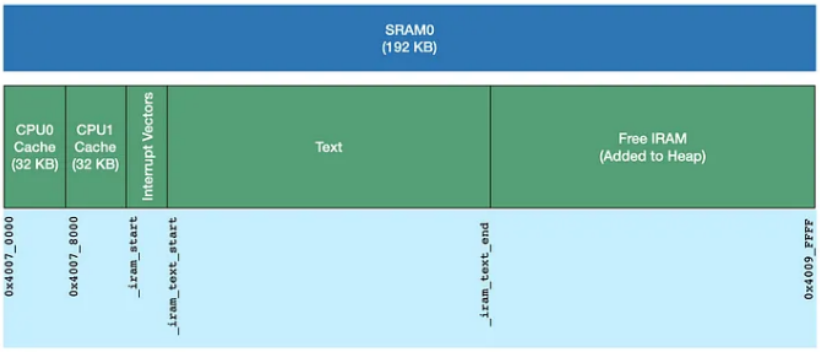


FIGURE 2.19: IRAM Layout

SRAM1 can be used as a IRAM and DRAM both, for practical purposes, ESP-IDF uses SRAM1 as DRAM, as it's generally the data memory that applications fall short of. The diagram in figure 2.17 shows the memory map for programmers to consider for their application development where they get 192KB **IRAM** and 328KB **DRAM**. While it does not matter much for the application as there is no overlap, please note that the direction of the address range is opposite for IRAM and DRAM address spaces.

While SRAM1 can be used as a IRAM and DRAM both, for practical purposes, ESP-IDF uses SRAM1 as DRAM, as it's generally the data memory that applications fall short of. The diagram 2.18 shows the **memory map for programmers** to consider for their application development where they get 192KB IRAM and 328KB DRAM. While it does not matter much for the application as there is no overlap, please note that the direction of the address range is opposite for IRAM and DRAM address spaces.

Let us now zoom into **the IRAM section** (see figure 2.19).

The 192 KB of available IRAM in ESP32 is used for code execution, as well as part of it is used as a cache memory for flash (and PSRAM) access.

- First 32KB IRAM is used as a CPU0 cache and next 32KB is used as CPU1 cache memory. This is statically configured in the hardware and can't be changed.
- After the first 64KB, the linker script starts placing the text region in IRAM. It first places all the interrupt vectors and then all the text in the compiled application that is marked to be placed in IRAM. While in common case, majority of

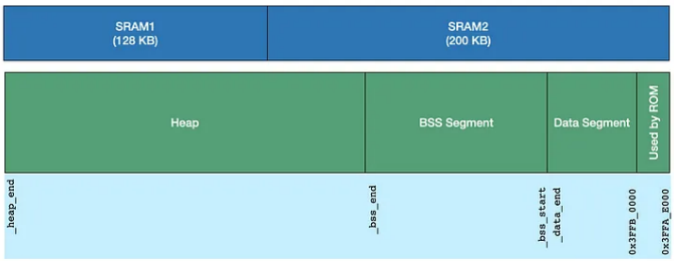


FIGURE 2.20: DRAM Layout

the application is executed out of the flash (XiP), there are some portions of the applications which are time critical, or that operate on flash itself. They need to be placed in IRAM and that is achieved using a special attribute to these functions or files and linker script doing a job of placing them in IRAM. The symbols _iram_text_start and _iram_text_end are placed by the linker script at the two boundaries of this text section.

- The IRAM after the text section remains unused and is added to the heap.

_iram_text_start and **_iram_text_end** symbols are **placed by the linker script at the two boundaries of this text section**. The IRAM after the text section remains unused and is added to the heap. Also, when the application is configured in a single-core mode, the CPU1 is not functional and CPU1 cache is unused. In that case, CPU1 cache memory (0x40078000–0x4007FFFF) is **added to the heap**. The unused IRAM, that is placed in the heap, can be accessed through dynamic allocations. It can be used to place any code in IRAM if the application has such a requirement. However this is quite uncommon. The IRAM can also be used for data, but with two important limitations.

1. The address used for access to the data in IRAM has to be 32-bit aligned.
2. The size of data accessed too has to be 32-bit aligned.

If the application has such data that can obey these two rules of accesses, it can make use of IRAM memory for that data.

Now let us see the **DRAM organization** (see figure 2.20).

The diagram 2.20 shows a typical (simplified) DRAM layout for an application. As the DRAM addresses start at the end of SRAM2, increasing in backward direction, the link time segments allocation happens starting at the end of SRAM2.

- The first 8KB (0x3FFAE000–0x3FFAFFFF) are used as a data memory for some of the ROM functions.
- The linker then places initialised data segment after this first 8KB memory.
- Zero initialised BSS segment comes next.
- The memory remaining after allocating data and BSS segments, is configured to be used as a heap. This is where typical dynamic memory allocations go.

Please note that the size of data and BSS segments depend on the application. So **each application**, based on the components that it uses and APIs it calls **has a different available heap size** to begin with. There are two regions within the heap

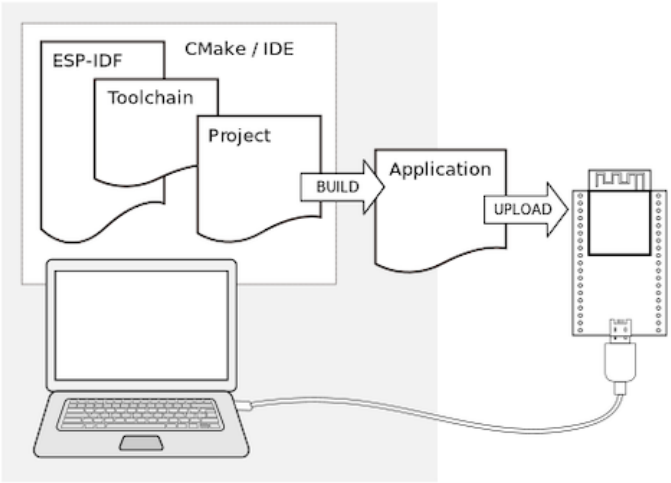


FIGURE 2.21: ESP-IDF Toolchain

(0x3FFE0000–0x3FFE0440 — 1088 bytes) and (0x3FFE3F20–0x3FFE4350–1072 bytes) that are used by ROM code for its data. These regions are marked reserved and the heap allocator does not allocate memory from these regions.

The above content along with images has been taken from the ESP32's blog post written by [Inamdar, 2020].

2.1.2 Brief description of the environments

Usually, working with **embedded devices** is defined by a development environment on your laptop and your microcontroller attached to the laptop with a USB cable. In order to write the **firmware** onto the microcontroller we need a **compiler toolchain** with target the microcontroller to generate the firmware and then a protocol that allows us to write (or flash) the firmware on the microcontroller. The process of compiling a programme/firmware for another architecture or board is called cross-compilation. I used two tools to help me with the cross-compilation and the protocol, being ESP-IDF and PlatformIO

ESP-IDF

To start using ESP-IDF on ESP32, install the following software:

- Toolchain to compile code for ESP32.
- Build tools - CMake and Ninja to build a full Application for ESP32.
- ESP-IDF that essentially contains API (software libraries and source code) for ESP32 and scripts to operate the Toolchain (see figure 2.21).

Once the ESP-IDF toolchain is installed, you are able to **cross-compile** and **flash** the **firmware** on your board via Python scripts. If you followed the installation instructions you just type the command *get_idf* on a terminal and you will have the environment prepared for that terminal session.

```
$ get_idf
Setting IDF_PATH to '/home/chloe/esp/esp-idf'
Detecting the Python interpreter
```

```
Checking "python3" ...
Python 3.10.12
"python3" has been detected
Checking Python compatibility
Checking other ESP-IDF version.
Adding ESP-IDF tools to PATH...
Not using an unsupported version of tool cmake found in PATH: 3.22.1.
Not using an unsupported version of tool ninja found in PATH: 1.10.1.
Checking if Python packages are up to date...
Constraint file: /home/chloe/.espressif/espidf.constraints.v5.0.txt
Requirement files:
 - /home/chloe/esp/esp-idf/tools/requirements/requirements.core.txt
Python being checked: /home/chloe/.espressif/python_env/idf5.0_py3.10_env/bin/python
Python requirements are satisfied.
Added the following directories to PATH:
  /home/chloe/esp/esp-idf/components/esptool_py/esptool
  /home/chloe/esp/esp-idf/components/espcoredump
  /home/chloe/esp/esp-idf/components/partition_table
  /home/chloe/esp/esp-idf/components/app_update
  /home/chloe/.espressif/tools/xtensa-esp-elf-gdb/11.2_20220823/
  xtensa-esp-elf-gdb/bin
  /home/chloe/.espressif/tools/riscv32-esp-elf-gdb/11.2_20220823/
  riscv32-esp-elf-gdb/bin
  /home/chloe/.espressif/tools/xtensa-esp32-elf/esp-2022r1-11.2.0/
  xtensa-esp32-elf/bin
  /home/chloe/.espressif/tools/xtensa-esp32s2-elf/esp-2022r1-11.2.0/
  xtensa-esp32s2-elf/bin
  /home/chloe/.espressif/tools/xtensa-esp32s3-elf/esp-2022r1-11.2.0/
  xtensa-esp32s3-elf/bin
  /home/chloe/.espressif/tools/riscv32-esp-elf/esp-2022r1-11.2.0/
  riscv32-esp-elf/bin
  /home/chloe/.espressif/tools/esp32ulp-elf/2.35_20220830/esp32ulp-elf/bin
  /home/chloe/.espressif/tools/cmake/3.24.0/bin
  /home/chloe/.espressif/tools/openocd-esp32/v0.11.0-esp32-20221026/
  openocd-esp32/bin
  /home/chloe/.espressif/tools/ninja/1.10.2/
  /home/chloe/.espressif/python_env/idf5.0_py3.10_env/bin
  /home/chloe/esp/esp-idf/tools

Detected installed tools that are not currently used by active ESP-IDF version.
For removing xtensa-esp32s3-elf, esp32ulp-elf, cmake, xtensa-esp32-elf,
riscv32-esp-elf-gdb, riscv32-esp-elf, ninja, xtensa-esp32s2-elf, xtensa-esp-elf-gdb use com
To free up even more space, remove installation packages of those tools.
Use option 'python3 /home/chloe/esp/esp-idf/tools/idf_tools.py uninstall
--remove-archives'.

Done! You can now compile ESP-IDF projects.
Go to the project directory and run:

  idf.py build
```

Another interesting feature ESP-IDF provides is the **addr2line** feature that allows you to convert an address into a function and position in the code w.r.t. a specific firmware image.

```
xtensa-esp32-elf-addr2line -pfiaC -e build/PROJECT.elf ADDRESS
```

PlatformIO

PlatformIO is a **user-friendly** and extensible **integrated development environment** with a set of professional development instruments, providing modern and powerful features to speed up yet simplify the creation and delivery of embedded products. It leverages sometimes the tools provided by the ESP-IDF to operate on the ESP32's board. It leverages a platformio.ini file for configuration purposes.

```
//example of a platformio.ini file

[env:az-delivery-devkit-v4]
platform = espressif32
board = az-delivery-devkit-v4
framework = espidf
build_flags =
    -DZENOH_ESPIDF
monitor_speed = 115200 // baud rate
```

While working on zenoh-pico I found the following bug on PlatformIO: it does not detect correctly the environment used thus it does not correctly set the environment variable, hence you have to specify on platformio.ini the build flags to build the zenoh-pico part for the zenoh-espidf.

2.2 Deep-Sleeping in IoT

A common pattern found in software with Micro Controller Units (MCUs) is an **active-sleep pattern** where the software operates on data for a while then it makes the board fall in deep sleep for a certain amount of time or is waken up by an external interrupt or sensors. By giving the capability to zenoh-pico of supporting deep-sleep by restoring the session after a deep-sleep we give the final user of zenoh-pico the possibility to consume less average power by the MCU **thus a longer battery lifetime**. According to the following formula the average power consumed by a MCU is the sum of the Power consumed while active plus the power consumed while in sleep mode plus the division of the Energy consumed when active by the time needed to wake up

$$P_{avg} = P_{always\_on} + P_{sleep} + \frac{E_{active}}{T_{wkup}}$$

The figure below represents a common pattern found on MCUs connected to sensors. We can get a longer battery lifetime by minimizing the energy consumed by the general operations. The real time processing of sensor data equals to the minimization of

$$minE_S + E_{MCU} s.t. T_{CPU} < \frac{1}{F_S}$$

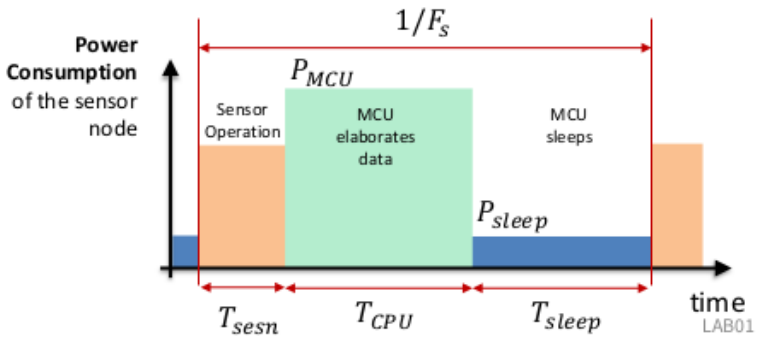


FIGURE 2.22: Lifecycle of a MCU

where E_S is the energy consumed by the sensors and E_{MCU} is the energy consumed by the microcontroller. Assuming a fixed sample rate F_S, a negligible sensor energy cost $E_S << E_{MCU}$ and a constant power envelope of the MCU for both active mode (P_{MCU}) and sleep mode (P_{sleep}) we get

$$minT_{CPU}P_{MCU} + T_{sleep}P_{sleep}whereT_{CPU} + T_{sleep} = \frac{1}{F_S}$$

thus minimizing the T_{CPU} will extend the battery lifetime. The processing time T_{CPU} is composed by the execution time and the idle time where the CPU waits for events or interrupt requests (IRQs). Given a CPU clock frequency, the performance of a task can be measured as

$$T_{CPU} = N_{CLK}T_{CLK} = CPI_{avg}N_{istr}T_{CLK}whereCPI_{avg} = N_{CLK}N_{istr}$$

2.3 The Company - ZettaScale Technology

ZettaScale Technology is the company which hosted me for my internship in the period February 2022 - July 2022. It is a spin-off of the Taiwan-based **ADLINK Technology**, a leader in the **embedded computing**, **edge computing**, and **intelligent computing** areas and it develops cutting-edge technologies. ADLINK Tech. has many offices located **around the world**, in particular it has its global headquarters in Taiwan and offices in China, Japan, South Korea, Singapore, United States of America, United Kingdom, Israel, Germany, Netherlands, and France. ADLINK Technology is composed of 6 Design Centers, 6 Operations and Logistic Centers, and 22 Support Offices. ADLINK Technology has been on the **Internet of Things** scene since 1995 when it was first founded.

The Support Office ZettaScale Technology at Saint-Aubin, France, 91190, is led by the CEO & CTO **Angelo Corsaro**. Him and his world class team maintain and develop zenoh, zenoh-pico, and zenoh-flow. Zenoh is a **framework**, a **protocol**, and a **middleware** capable to operate in the **cloud-to-thing continuum**. Zenoh-Pico is the subject of this master thesis and it provides the zenoh API on embedded devices like ESP32 boards or Zephyr boards. Zenoh-Flow is a data-flow programming framework mostly used in automotive and robotics.

I will introduce Zenoh and deeply explain Zenoh-Pico in the next chapter.

Chapter 3

Zenoh-Pico

Zenoh provides a stack that unifies data in **motion**, data at **rest**, and **distributed computations** by carefully blending traditional pub/sub with geo-distributed storages, queries, and computations while retaining a level of time and space efficiency that is well beyond any of the mainstream stacks. **Zenoh** and **Zenoh-Pico** provide a slim API to not overload tiny microcontrollers. Feature extensibility is provided by the **pattern option** struct in C. This allows Zenoh API to accomodate the addition of any new capability in the future whilst being backward compatible.

Zenoh-Pico is the Zenoh implementation that targets **constrained devices** by providing a lightweight implementation of the functionalities provided by Zenoh itself. Zenoh-Pico can run on Unix systems, Windows systems, Zephyr board, Arduino environments, ESP-IDF environments, MbedOS, OpenCR, and Emscripten. It is compatible with its main Zenoh implementation. What Zenoh-Pico does is managing data in the **cloud-to-thing continuum** that is data moving from the cloud to devices and vice-versa. With Zenoh-Pico, Zenoh opens the door to **ubiquitous computing** with something more with respect to vertical and horizontal scalability - it is support for constrained devices with **low duty cycle**. In other terms, devices that are disconnected or sleeping most of the time as well as ability to deal with both data in motion and data at rest. It allows configuration of features (including enable/disable) at compilation time in order to reduce its footprint, including support for single or multiple thread mode. Additionally, Zenoh-Pico is compliant to MISRA-C.

3.1 About Zenoh

3.1.1 Zenoh Keys

Zenoh being a **Named-Data oriented protocol**, its address space is the space of names given to data. In Zenoh to every **key** is associated a **value**, the key looks like a Unix file system path e.g. **/home/kitchen/temp**, and the value can be of different encodings like JSON, string, raw byte buffer *et cetera*. The key represents a resource which can be a read value from a sensor, for instance. Thus, to address data are used **key expressions**. A formal definition and formalization has been done for the following reasons:

1. to **avoid ambiguity** in key expression definition and matching.
2. to **improve the key expression matcher** for better performance.
3. to allow **future extensibility** to be introduced for more complex matching and behaviours.

We can define a key expression as a /-separated list of chunks, where each chunk is a non-empty (UTF-8) string. The single wild * expresses exactly one chunk of any value, and is equivalent to the $[\hat{/}]+$ regular expression. For example, a/*/b:

- includes a/c/b, a/hi/b
- intersects */a/b, */*/*
- does not intersect a/**/b/c

3.1.2 Zenoh data messages

Zenoh data message has a **minimal wire overhead** which sums up to at least **3 bytes** – 1 byte for the data header, 1+ bytes for the resource, 1+ bytes to encode the user data length then the user payload. The minimal wire overhead has been achieved using Variable Length Encoding (VLE) variables, such variables can be compressed with zero error and read back symbol by symbol without losing any data. Zenoh-Pico uses the following coding strategy. For the user data length, the field represents a 64-bit integer. The resource key is dynamically mapped in a more compact integer form called **resource ID**, thus having the possibility to encode the resource ID with a smaller number of bytes dependently on its size. Now, the representation of resource keys allows to represent prefixes. Previously, zenoh could represent on the wire either a numerical resource identifier or a full resource key and now it is able to represent a prefix and a suffix for instance /org/eclipse/zenoh/demo/hello becomes (42, "demo/hello"). Whereas zenoh frame messages use at least 2 bytes as overhead – 1 byte for the frame header and 1+ byte for the sequence number. The frame can have multiple user messages decided in a process called **automatic batching**. The sequence number can be dimensioned as the end-user's leisure. A higher resolution means the network can have more messages on the flight whereas a lower resolution means the network can have way more less messages on the flight but once encoded the sequence number will occupy a few bytes (just 1 byte if the resolution is set to 128).

3.1.3 Zenoh Router

Zenoh routers **route data** between clients and local subnetworks of peers. They can be deployed using any topology. They, by default, never try to interconnect themselves automatically. They must be configured with the endpoints of the other routers they are supposed to connect to via a configuration file.

3.1.4 Client mode and peer mode

In client mode the client **communicates with a Zenoh router**, whereas **in peer mode** it **routes information between themselves** and can also route on behalf of clients acting like routers. Peer-to-peer communication is supported for arbitrary connectivity graphs and supports cliques as special case. Peer mode and client mode can be chosen at runtime. Communicating peer to peer implies establishing **multiple sessions** with multiple peers and a state must be maintained for those sessions. Maintaining such states can be undesirable either for scalability reasons or because the application runs on a constrained device. In the latter case, the Zenoh application can be configured to operate in client mode. In such mode, the application will maintain, at any given time, a single session with another process, typically a Zenoh router, that will grant connectivity with the rest of the system.

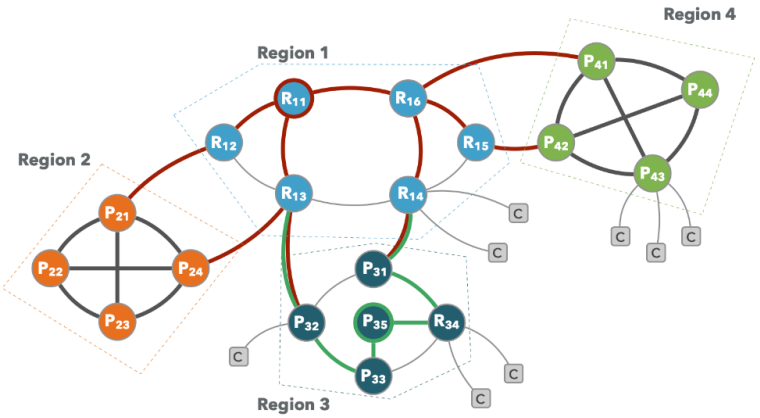


FIGURE 3.1: A Zenoh topology

3.1.5 Scouting

By default all **Zenoh applications run in peer to peer mode**. Such applications, in order to discover other Zenoh applications or Zenoh routers, run both

- **multicast scouting**: Zenoh applications in peer mode join the multicast group 224.0.0.224:7446 over UDP. Scout messages are sent to this address to discover local applications and routers. The scouting address and the behavior can be configured via configuration files.
- **gossip scouting**: Zenoh applications in peer mode forward all local applications and router they have already discovered to the newly scouted applications. Applications need to connect to an **entry point** to discover the rest of the system. This entry point can be one or several Zenoh routers or one or several Zenoh peers.

Closure-based discovery, supported by the scouting protocol, is deployed in order to ease the deployment of systems that want to leverage a clique connectivity for cases in which multicast is not available. In few terms starting from a single peer we can discover its closure that are peers reachable directly or indirectly from the starting point. **Region-based routing** means routing information required to build and maintain the routing tables scales with the size of the region, and each region can decide whether to route over an arbitrary connectivity graph or assume a clique.

3.1.6 Communication models

The most common **communication patterns** found in IoT applications:

- Request and Response.
- Publisher and Subscriber.
- Push and Pull.

Request and Response

The Request and Response communication model consists of:

- A client: it initiates the exchange of messages with the server. The first message can be called "request".

- A server: it receives messages from the clients and produces responses.

Nowadays, it is the current communication model for the HTTP protocol.

Publisher and Subscriber

The Publisher and Subscriber communication model consists of:

- A publisher: publishes data named for instance demo/example/zenoh-pico-pub/temp.
- A subscriber: receives data of which it is interested for example data named demo/example/zenoh-pico-pub/temp.
- A broker: manages the published data by the publisher by routing it to the correct subscribers.

In our case, the zenoh routers (or the zenoh peers) can assume the role of a broker whereas clients can be the publishers and the subscribers. Examples can be found in z_pub and z_sub.

Push and Pull communication model

The Pull communication is a type of network communication where the client application initiates the communication by requesting updates. It pollingly asks for updates to the server. The Push communication model allows the server to send updates to the client whenever new data becomes available without the need for the client to explicitly request it. Example can be found in z_pull. The Push counterpart can be seen as in the publisher and subscriber model when the subscriber subscribes to a named data and wait for updates to that data.

3.1.7 Reliability

Hop to Hop reliability

The Zenoh protocol is composed of **two layers**: the **session protocol** and the **routing protocol**. The session protocol establishes a bidirectional 1-1 session between two Zenoh runtimes which can be either client, peer, or router. Each session comes by default with a best-effort channel and a reliable channel. The session protocol, among other things, takes care of performing automatic batching in order to maximise network usage and fragmentation to give the illusion of an unlimited MTU. Whereas, the routing protocol leverages the session protocol to propagate interests and route data from many producers to many consumers. Consequently, the reliability state maintained by each application is independent of the number of data producers and data consumers hosted by the application. As Zenoh offers routed communication, a single data producer connected to a Zenoh router can reliably send data to as many data consumers as needed while maintaining a single reliability state (see figure 3.2). This strategy is **highly scalable** and offers a **good level of reliability**. No data samples are lost while the infrastructure is stable. If a Zenoh router fails, the Zenoh infrastructure will automatically adapt to the new topology. During the **failover**, data samples may be **lost**. As soon as the infrastructure is re-stabilized, data is reliably distributed again.

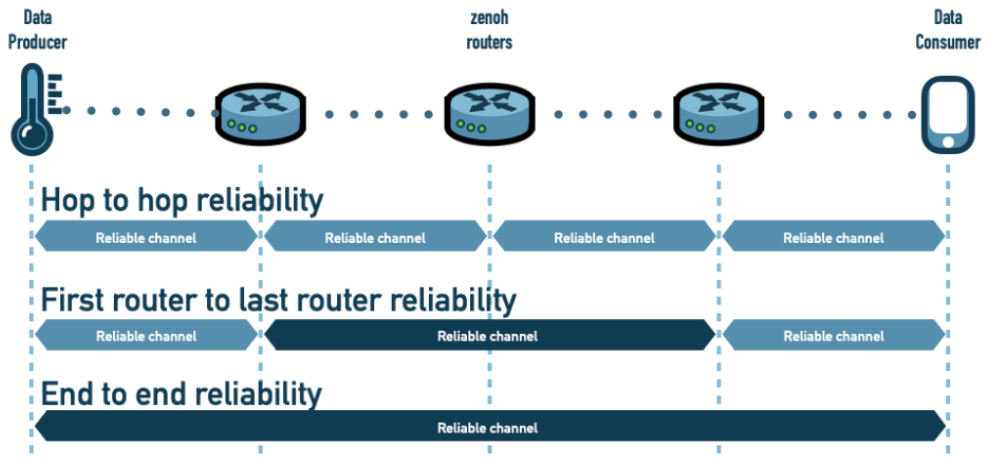


FIGURE 3.2: Zenoh Reliability

End to End reliability

A reliability channel is established between each data producer and data consumer pair. This avoids sample loss even during topology changes but is **less scalable** and **induces higher resource consumption** on producers and consumers.

First router to last router reliability

A reliability channel is established between the first Zenoh router and the last Zenoh router of each data route. This allows to **relax pressure on producers and consumers** by deporting the pressure to the nearest infrastructure components.

3.1.8 Reliability & Control Flow

- In Zenoh the receivers control reliability by selecting a **resending strategy**. It must be declared.
- The senders and intermediate infrastructure components (I.E. the Zenoh routers) individually decide how much **memory** they are willing to **dedicate to reliability**. This allows constrained devices to dedicate few resources while resourceful intermediate routers can dedicate more memory lowering the probability of congestion situations.
- Senders control congestion by selecting a **message dropping** strategy. For each sample they decide what should be done in case of congestion I.E. drop the sample or block the publication. The congestion control strategy is propagated from the sender to all involved infrastructure components and applied along the entire routing path.

3.1.9 Mobility

When mobility gets in, things get more complicated. As a device is moving and the **handoff** occurs in the physical layer, if the device remains connected to the same Zenoh router I.E. the anchor point, the data path becomes less and less optimal as the distance within them grows. The Zenoh wire protocol is very lightweight and its discovery very efficient along with its infrastructure and its several routing algorithms which support high dynamicity. This allows Zenoh devices to **migrate** their

sessions from one Zenoh router to another one in a smooth and seamless manner. Devices can then connect to the nearest router **after the mobility occurs** and maintaining optimal latency by migrating from Zenoh router to another one by moving. To perform such Zenoh session migration it is needed that the device:

1. **Detects** that it moved, more precisely that the handover occurred. This detection is highly dependant on the kind of wireless communication involved (5G, WiFi, etc.) and it could be detected by the Zenoh scouting protocol.

2. **Finds** the new closest Zenoh router. This is dependent on the underlying infrastructure and would leverage an implementation of Zenoh's scouting that is specific to the network.

3.1.10 Zenoh over Serial

By supporting serial communications, Zenoh is unleashing its potential to a new set of devices **lacking any kind of networking interface**. This is a common scenario, especially in robotics, vehicles, bus, maritime, agricultural, and industrial devices where conventional computer network technologies are rarely used.

3.1.11 Replicated storages

Zenoh ensures **eventual consistency** for storages that subscribe for the same key expression, even in the presence of network partitions and system faults. The storage alignment protocol is called "anti-entropy protocol"; as the name suggests it keeps the entropy in the system low.

3.1.12 Payload to the query

Zenoh has the possibility to **attach some user payload when issuing a query**. This simplifies and makes it more efficient for an application to pass information, such as arguments or body, in the query that can be received and interpreted by the matching queryables. For example, you can attach a picture to a query that is analyzed by a queryable running an object detection algorithm. The above contents have been taken from the [ZenohTeam-Blog, 2024].

3.1.13 Hybrid Logical Clock (HLC)

Logical clock (LC). LC was proposed in 1978 by **Lamport** as a way of timestamping and **ordering events** in a **distributed system**. LC is divorced from physical time (e.g., NTP clocks): the nodes do not have access to clocks, there is no bound on message delay and on the speed/rate of processing of nodes. The causality relationship captured, called **happened-before (hb)**, is defined based on passing of information, rather than passing of time.1 While being beneficial for the theory of distributed systems, LC is impractical for today's distributed systems: 1) Using LC, it is not possible to query events in relation to physical time. 2) For capturing hb, LC assumes that all communication occurs in the present system and there are no backchannels. This is obsolete for today's integrated, loosely-coupled system of systems. In 1988, the vector clock (VC) was proposed to maintain a vectorized version of LC. VC maintains a vector at each node which tracks the knowledge this node has about the logical clocks of other nodes. While LC finds one consistent snapshot (that with same LC

values at all nodes involved), VC finds all possible consistent snapshots, which is useful for debugging applications.

Physical Time (PT). PT leverages on physical clocks at nodes that are **synchronized** using the Network Time Protocol (NTP) [20]. Since perfect clock synchronization is infeasible for a distributed system, there are uncertainty intervals associated with PT. While PT avoids the disadvantages of LC by using physical time for timestamping, it introduces new disadvantages: 1) When the uncertainty intervals are overlapping, PT cannot order events. NTP can usually maintain time to within tens of milliseconds over the public Internet, and can achieve one millisecond accuracy in local area networks under ideal conditions, however, asymmetric routes and network congestion can occasionally cause errors of 100 ms or more. 2) PT has several kinks such as leap seconds and non-monotonic updates to POSIX time which may cause the timestamps to go backwards.

TrueTime (TT). TrueTime is proposed recently by Google for developing Spanner, a multiversion distributed database. TT relies on a well engineered tight clock **synchronization** available at **all nodes** thanks to GPS clocks and atomic clocks made available at each cluster. While TT avoids some of the disadvantages of LC/VC/PT, it introduces new disadvantages: 1) TT requires special hardware and a custom-build tight clock synchronization protocol, which is infeasible for many systems (e.g., using leased nodes from public cloud providers). 2) If TT is used for ordering events that respect causality then it is essential that if e hb f then tt.e < tt.f. Since TT is purely based on clock synchronization of physical clocks, to satisfy this constraint, Spanner delays event f when necessary. Such delays and reduced concurrency are prohibitive especially under looser clock synchronization.

HybridTime (HT). HT, which combines VC and PT clocks, was proposed for solving the stabilizing causal deterministic merge problem. HT maintains a VC at each node which includes knowledge this node has about the PT clocks of other nodes. HT exploits the clock synchronization assumption of PT clocks to trim entries from VC and reduces the overhead of causality tracking. In practice the size of HT at a node would only depend on the number of nodes that communicated with that node within the last ε time, where ε denotes the clock synchronization uncertainty.

The Hybrid Logical Clock (HLC) is a logical clock version of HT. We could examine how the algorithm works under the hood but for the purpose of this master's thesis knowing that HLC are a logical clock version of HT is more than sufficient [Kulkarni, 2014].

3.2 Architecture

The whole Zenoh framework is written using the **Rust programming language** thus it operates through an Object-Oriented Programming paradigm and it supports the features of a functional language. Zenoh has a complex architecture (see figure 3.3) which allows it to perform from the **pub/sub/query** to **store** in different type of databases the results of the publications/subscriptions/queries. It manages REST, the different type of backends, the storage manager through a **plugin architecture** which contains a plugin manager that manages all the plugins by loading the needed ones at runtime, this is the behavior of the zenohd router. Indeed, we have already seen that zenoh has 2 mode of communication: client and peer mode. The former connects to a zenohd router from which its messages are routed in the zenoh's infrastructure. The latter forms a network with the different peers or zenohd routers and their messages are sent in the just created network composed of peers and zenohd

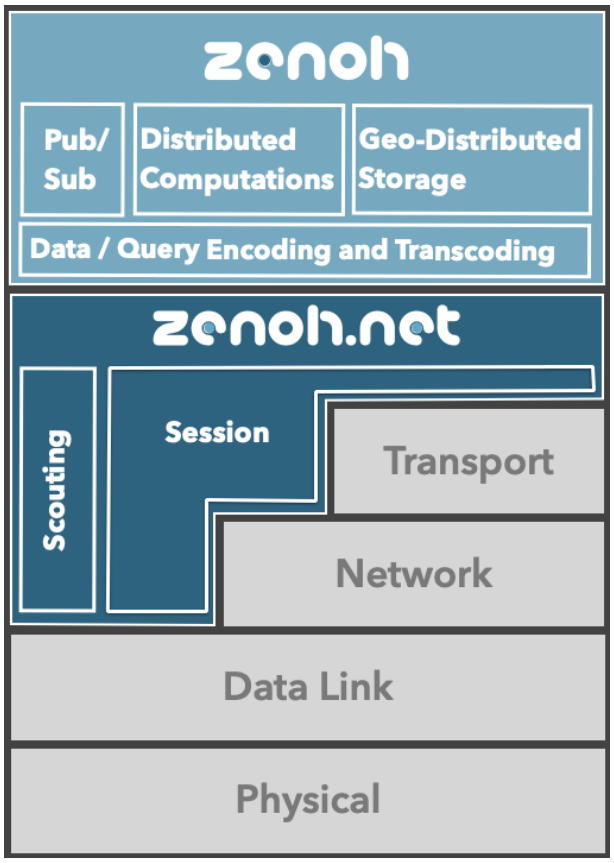


FIGURE 3.3: Zenoh Layers

routers. It is possible to configure the majority of the zenoh's features through a **configuration file**.

3.2.1 Zenoh-Pico's architecture

I have written so far of the main characteristics of the zenoh's architecture. Now let me write and delve deeper on the zenoh-pico's architecture, the counterpart of zenoh for embedded devices which is written using the **C programming language**, the constrained devices' language par excellence.

Platform-Agnostic

The zenoh-pico's architecture aims, among the many other things, to a **platform-agnostic** architecture. This is due to the fact that the embedded systems world is constituted by many different microcontrollers' environments, the most famous ones (and supported by zenoh-pico) are ESP-IDF, zephyr, Arduino, mbedOS, OpenCR, Emscripten (for web applications), FreeRTOS-Plus-TCP and at last Unix and Windows. What changes between these environments are:

- the **system calls** used for instance to open a network socket.
- their respective **architecture**.
- the **operating systems** in use e.g. ESP-IDF uses the freeRTOS operating system.

In order to accomplish the platform-agnostic architecture the professional embedded systems development environment **PlatformIO** comes to rescue us. Indeed,

zenoh-pico uses a small python script called **extra_script.py**. PlatformIO sets at runtime the framework in use (in our case it is ESP-IDF which corresponds to the string 'espidf'). The python script tests at runtime which one is the framework in use and accordingly to that information loads the correspondent **system/<platform> folder** in our case the folder is system/espidf. Along with it, it allows to set to "1" macros like ZENOH_ESPIDF which are used later in the code. Alas, PlatformIO at the current date has a bug which does not correctly set the macro at runtime resulting in the file platformio.ini to contain

```
build_flags = -DZENOH_ESPIDF
```

otherwise the compiler would have returned **"Unknown platform"**. This is because the macro ZENOH_ESPIDF is used in the file platform.h to choose what to include among the different environments supported by zenoh-pico. Here a snippet on what the code looks like just to give a rough idea:

```
#elif defined(ZENOH_ESPIDF)
#include "zenoh-pico/system/platform/espidf.h"
```

The file espidf.h contains nothing but the definition of the zenoh type _z_sys_net_socket_t and the zenoh type _z_sys_net_endpoint_t which are necessary to manage the TCP/UDP/SERIAL communication on the board ESP32 which uses freeRTOS as operating system.

The inclusion

After the platform has been set, it is necessary to include all the headers with zenoh-pico.h *in primis*. To do so the folders' hierarchy must be as follows:

```
zenoh-pico
|-include
|   |-zenoh-pico.h
|   |-zenoh-pico
|-src
```

This must be done as-is because the default inclusion files for the CMake, accordingly to the CMakelists.txt, are in the include folder:

```
if(SKBUILD)
  set(INSTALL_RPATH "zenoh")
  set(INSTALL_NAME_DIR "zenoh")
  set(INSTALL_INCLUDE_NAME_DIR "zenoh/include")
endif()
```

Thus, when called

```
#include "zenoh-pico.h"
```

the default location is the include folder as referred by the CMakelists.txt.

Architecture as macro-blocks

Looking at the code's architecture we recurringly find in the code the start and stop of the lease and read task at the edges of the business logic of our program in the main() function. This is because the lease and read tasks are two important tasks for the architecture of zenoh-pico. As suggested by the name of the tasks, the lease task manages the **session leasing timeout** whereas the read task operates to make **reads** from the network in an asynchronous manner.

Still looking at the code's organisation we can tear down the code's architecture to the following **macro-blocks**:

- api
- collections
- link
- net
- protocol
- session
- system
- transport
- utils
- config.h

The **api** folder has a file containing some "public" functions whose purpose goes from creating a default configuration file to check whether the key expression is canon. Also, it defines zenoh types, zenoh constants, zenoh primitives and zenoh macros.The file macros.h in the include/api folder contains, among many other things, the macros to foster an **ownership model** like the one in the Rust programming language. This custom ownership model works like this - any destructible type, thus a type which requires a free(), must start with z_owned. To move data among functions the owned type must be wrapped in a z_move function which will transfer the ownership of the owned data to the callee function, that means the callee function has the duty to free() the passed data. On the other hand, if the callee function just need to borrow data then it will take values as z_loan(data). In the latter case, the duty to free() the owned data is of the caller function.

The **collections** folder contains the private primitives to operate on collections such as lists, vectors, intmaps, strings, bytes and slice of bytes. I said "private" as the functions implementing the lists and the other collections they start with a _ that means in the C lingo that that function must be called only by the zenoh-pico core. Many of the _z_ functions are wrappers to **system calls** (such as zp_malloc which calls malloc) or to already famous function (such as _z_str_n_copy which calls strncpy).

The **link** folder contains what is needed to the link such as a zenoh UDP socket or a zenoh TCP socket or a zenoh raw ethernet socket or a zenoh websocket socket. The **manager** has the job to create the link and check whether the link is still valid. The implementation of the headers contains what is needed to open/listen/close/free/write/read-/create a new **link** on a specific zenoh socket. In the implementation there are two

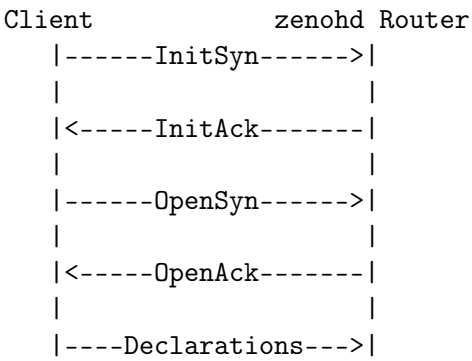
types of sending information wherever the protocol underneath zenoh support it - unicast (serial, TCP, UDP, websocket) and multicast (bluetooth and UDP).

In the **net** folder we find the primitives to declare/undeclare a resource, the scouting primitive, the declare/undeclare a publisher, the declare/undeclare a subscriber, the declare/undeclare queryable, and the query primitive to query data from the matching queryables in the system. The declaration and undeclaration is a typical step of the zenoh protocol. Moreover, this folder contains, in the session.h, the definition of the session we serialized and deserialized in the RTC Slow Memory along with the operations on the session.

The **protocol** folder contains two subfolders which are **codec** and **definitions**. The codec folder has the headers and logic to serialize or deserialize a zenoh message on or from the underneath protocol payload. The serialization and deserialization is little-endian. In the definitions folder there are the headers and logic of the internals of the zenoh messages such as how the messages are structured in the transport protocol, for example the following is the HELLO message, very important in the scouting phase to tell the node sending HELLO is reachable with a certain locator or list of locators (e.g. tcp/192.168.1.98):

```
 7 6 5 4 3 2 1 0
+-+-+-+-+-+-+-+-+
|Z|X|L|  HELLO  |
+-+-+-+---------+
|    version    |
+---------------+
|zid_len|X|X|wai| (*)
+-+-+-+-+-+-+-+-+
~     [u8]      ~ -- ZenohID
+---------------+
~   <utf8;z8>   ~ if Flag(L)==1 -- List of locators
+---------------+
(*) WhatAmI. It indicates the role of the zenoh node sending the HELLO message.
   The valid WhatAmI values are:
   - 0b00: Router
   - 0b01: Peer
   - 0b10: Client
   - 0b11: Reserved
```

In order to **open** a zenoh session the zenoh protocol sends the following **messages** on the network:



Note that the session has associated two **timers**, one timer from the client and one timer from the zenohd router. When the timer exceeds a certain configurable value

then the session is no longer valid. To configure the value after which the session is no longer valid it is necessary to check for it in the config JSON5 for the zenohd router and the config.h in the zenoh-pico's codebase. Of the two configured timers, it is taken the value of the minimum valued timer configured. The rest of the header files in the **protocol** folder are about definitions and operations on the key expressions, extensions to the zenoh protocol, the definitions of different type of buffers to send and receive to and from the network. The most important and basic buffer is the IOSli buffer which is structured as

```
typedef struct {
    size_t _r_pos;
    size_t _w_pos;
    size_t _capacity;
    uint8_t *_buf;
    _Bool _is_alloc;
} _z_iosli_t;
```

It is defined by its capacity, whether it is allocated or not, the read position of the cursor and the write position of the cursor.

The **session** folder contains the headers to operate on a zenoh session, to operate on subscriptions, to operate on list of replies to a query, the list of resources a node has to manage (e.g. "demo/temperature"), the subscriptions of the node, the current pending queries which are queries that have not received a reply final yet, registration/unregistration/operations on the resource IDs. Generally speaking, it contains the **operations** to the **elements** of the session's struct defined in the file net/session.h.

The **system** folder contains two subfolders which are **link** and **platform**. The link folder contains the header files for the operations to support the supported links by zenoh-pico. It supports the following links: bluetooth, raw ethernet, serial, TCP, UDP, websocket. The platform folder contains the header files to support many platforms. The supported platforms are Arduino (ESP32 and openCR), emscripten, ESP-IDF, freeRTOS-Plus-TCP, mbedOS, Unix, Windows, Zephyr. At last, the file platform.h contains nothing but the right includes to the right header file based on what environment variable has been defined like ZENOH_WINDOWS or ZENOH_ESPIDF. When no recognizable environment variable has been defined it is used a "gravestone include" to make the compiler compile and make good the syntax highlight, the "gravestone include" is the include with the file **platform/void.h**.

The **transport** folder is composed of 4 subdirectories which are **common**, **multicast**, **raweth** and **unicast**. The common one has the headers to send to and receive from the transport layer. The multicast has the headers to send to and receive from the multicast transport layer, moreover it contains the operations a transport layer must have to support multicasting. The same for multicast is valid for the unicast folder. The raweth folder contains the headers to transmit to and receive from the ethernet transport. The headers of which it is composed require a manager that manages the creation and the free of the transport layer.

The **utils** folder, as the name suggests, contains utilities for the zenoh framework. For instance to apply a CRC32 checksum to a vector of bytes, operations on strings, logging, operations on pointers, types of results a function may return, and at last my **header deep-sleeping.h** that defines the functions to support a deep sleeping but is not present in the Pico's codebase along with its implementation yet.

Chapter 4

The Project, the Implementation, and Experimental Evaluations

In this chapter I will discuss the **project**, the **implementation details** and the **experimental evaluations** taken from my implementation.

4.1 The project - Efficient support for deep-sleeping modes

The **deep sleeping mode** is a paramount mode on embedded systems where the device consumes way **less energy** and **power** than normal mode and nowadays in 2024 is even more important to have support for such mode. Here I will discuss the higher level details of my **project** leaving the implementation details to the next chapter. For Zenoh-Pico, I will create the support for deep sleeping and **hibernation** mode on the board ESP32 for the communication type **UDP unicast mode**.

In order to support the deep sleeping for Zenoh-Pico on the board ESP32 I had to **(1) save the session** in the RTC Slow Memory then **(2) use the same UDP port** among the deep sleeps. To have things clear in mind, I serialized before the deep sleeping the session, then deserialized and rebuilt in memory after the deep sleeping the session represented in listing 3.1 except the _mutex_inner variable:

LISTING 4.1: The Zenoh Session

```
typedef struct {
#if Z_MULTI_THREAD == 1
    _z_mutex_t _mutex_inner;
#endif  // Z_MULTI_THREAD == 1

    // Zenoh-pico is considering a single transport per session.
    _z_transport_t _tp;

    // Zenoh PID
    _z_id_t _local_zid;

    // Session counters
    uint16_t _resource_id;
    uint32_t _entity_id;
    _z_zint_t _pull_id;
    _z_zint_t _query_id;
    _z_zint_t _interest_id;

    // Session declarations
```

```
    _z_resource_list_t *_local_resources;
    _z_resource_list_t *_remote_resources;

    // Session subscriptions
    _z_subscription_sptr_list_t *_local_subscriptions;
    _z_subscription_sptr_list_t *_remote_subscriptions;

    // Session queryables
    _z_questionable_sptr_list_t *_local_questionable;
    _z_pending_query_list_t *_pending_queries;
} _z_session_t;
```

4.1.1 Saving the session

In order to save the session I had to *memcpy* the content of the session from the **heap** memory to the **RTC Slow Memory**. To do so, I created one function for each one of the lists I had to *memcpy* to the RTC Slow Memory. Moreover, I created a function for the _z_transport_t complex struct.

Why you did not use a third-party library to serialize/deserialize the session?

The serialization and deserialization process has been done a million of times and there exist **libraries** written in C (such as Protobuf) that will get the job done for you, so, why did I decide to implement the serialization/deserialization from scratch? There are a couple keypoints of why I decided to do it by myself, being:

- **less overhead** than using a third-party library for serialization/deserialization - the serialization/deserialization has been written ad-hoc for Zenoh-Pico and it removes the overhead and the memory/disk footprint left by a third-party library such as Protobuf or Binn and it is anyway fast and efficient even though must be said that the library Protobuf is already built-in in the libraries on the ESP32.
- Zenoh-Pico uses function pointers in many occasions and some libraries do not support the serde of function pointers.
- in Zenoh-Pico there are several **user-defined structs** for which it would have been needed anyway ad-hoc functions to serde the content of the user-defined structs.

One drawback of not having used a third-party library for serde is that my implementation is **not zero-copy** when it could be zero-copy, for instance in the deserialization process I create the space to store a string in the heap memory then I memcpy from the RTC Slow Memory to the just allocated heap memory thus making a copy of the string instead of creating a pointer *char** to the beginning of the null-terminated string in the RTC Slow Memory's buffer, doing so I would have saved a copy and my deserialize function would have been faster than it is right now.

Another drawback is **less abstraction**, indeed with a third-party library I could have abstracted away more the serde problem and focus on the real problem that was how to restore the UDP unicast communication after the deep-sleep reset of the MCU.

Why RTC Slow Memory?

Someone might object why I used the **RTC Slow Memory** and not the RTC Fast Memory or the NVS (Non-Volatile Storage) or the FAT (File Allocation Table) File System or the SPIFFS (Serial Peripheral Interface Flash File System) File System. Let us proceed step by step by discarding one solution at a time while explaining the reasons behind.

- The **NVS** library operates on key-value pairs and its application scenario is about storing parameters such as WiFi parameters so this one is discarded as it is possible to save only integers, strings and binary large object (BLOB) and NVS works well only for storing many small values.
- The **FAT** File System is capable of storing MiB to GiB of files and its application scenario is about storing audio, video and other files. It has been discarded as it is relatively slower than the RTC Slow/Fast Memory.
- The **SPIFFS** File System was a worthy option as it is an excellent storage on the embedded systems world due to its capability of storing on a SPI NOR flash device targeting embedded systems. SPIFFS occupies less RAM resources than FAT File System and is only used to support flash chips with their capabilities less than 128 MiB. It has been discarded as it is relatively slower than the RTC Slow/Fast Memory even if in the **future** could be used as an extension to support multicast and peer to peer mode by saving many sessions at once.
- The RTC Fast Memory is 8 KiB like the RTC Slow Memory and it is 10 times faster than the RTC Slow Memory. It has been discarded as it can be only accessed by PRO_CPU (Core 1) but could have been a worthy solution for a faster session serialization and deserialization to and from the RTC Fast Memory. With this solution would have only been possible to save one and only one session at a time thus supporting only unicast solutions as well as using the RTC Slow Memory.

At this point, for the future, a SPIFFS File System would have been a better solution in terms of **session scalability**.

Below, I show the static attributes used to save one and only one session hence not supporting extension capability to the **multicast** or **peer to peer** which use one session per connection with their peers/end systems.

LISTING 4.2: RTC static data attributes

```
RTC_DATA_ATTR static uint8_t local_resources[DIM_LOCAL_RESOURCES];
RTC_DATA_ATTR static uint8_t remote_resources[DIM_REMOTE_RESOURCES];

RTC_DATA_ATTR static uint8_t local_subscriptions[DIM_LOCAL_SUBSCRIPTIONS];
RTC_DATA_ATTR static uint8_t remote_subscriptions
                                        [DIM_REMOTE_SUBSCRIPTIONS];

RTC_DATA_ATTR static uint8_t local_questionable[DIM_LOCAL_QUESTIONABLE];

RTC_DATA_ATTR static uint8_t pending_queries[DIM_PENDING_QUERIES];

RTC_DATA_ATTR static uint8_t transport[DIM_TRANSPORT];
```

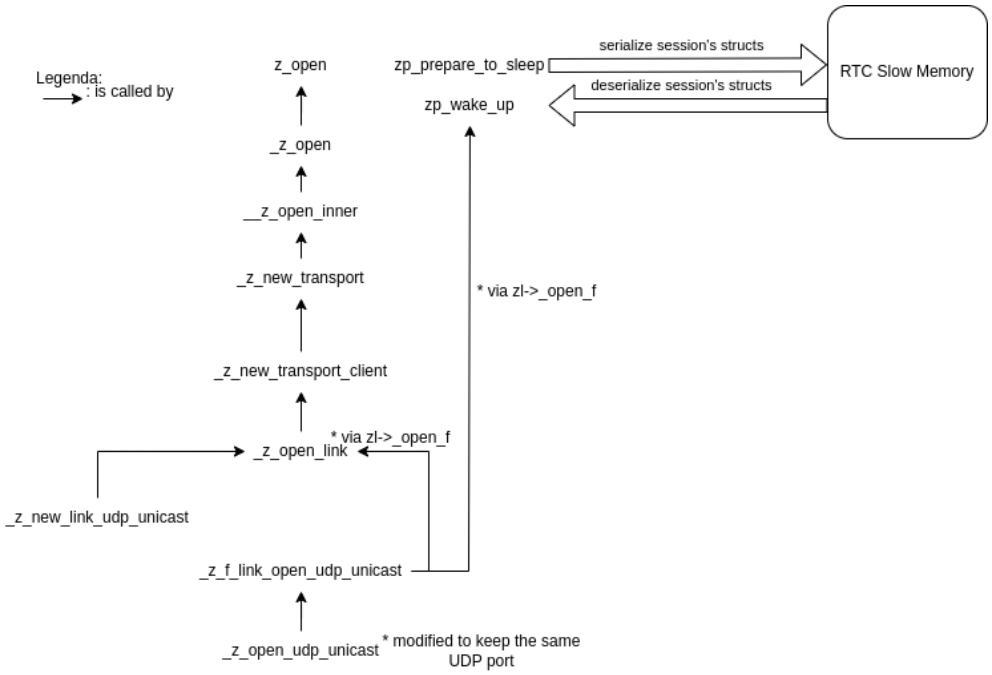


FIGURE 4.1: Function calls

```
// Zenoh PID
RTC_DATA_ATTR static uint8_t RTC_local_zid[16];

// Session counters
RTC_DATA_ATTR static uint16_t RTC_resource_id;
RTC_DATA_ATTR static uint32_t RTC_entity_id;
RTC_DATA_ATTR static size_t RTC_pull_id;
RTC_DATA_ATTR static size_t RTC_query_id;
RTC_DATA_ATTR static size_t RTC_interest_id;
```

See how the mapping is 1:1 w.r.t. the **zenoh's session** seen in Listing 3.1. The RTC_DATA_ATTR is a keyword to tell the compiler that such static variables are stored in the RTC Slow Memory. The type for serializing the lists is a uint8_t which is a byte-size vector. The DIM_X_Y is to give a **dimension** at compile time to the buffers containing the serialized lists and the zenoh transport. In case of weird errors happening at runtime, it is advisable to first check whether the dimensions are large enough for the buffers to store the whole list and not overflowing outside the buffer which will not cause an Overflow error at runtime.

4.1.2 Use the same UDP port

For the session to be correctly restored I had to **save the UDP port**. The figure 4.1 gives an **overview** of the function calls called to open a UDP unicast communication in Zenoh. The figure 4.1 helped me a lot to get where and when I had to call the function to recreate the UDP unicast communication and binding to the same UDP port for the Zenoh router to recognize me.

4.1.3 Restoring the session

To **restore** the zenoh session I *memcpy*'d from the RTC Slow Memory to the *malloc*'d structs of the element of the list the serialized content of the buffers. The result is a struct element of the list which is pushed in the correct zenoh list at runtime. It makes sense to restore the function pointers as on embedded devices it is all **statically linked** thus the position of the functions in the memory does not change between offs and ons or resets of the board.

4.1.4 Serde functions

The **se**rialization and **de**serialization functions (contracted to **serde**) are special function pointers used to serialize and deserialize user-given structs at the right time at the right position in the code. The serde functions have been very intrusive to the zenoh-pico API because they required the user to provide the serde functions via the option struct thus I had to edit many inner zenoh-pico functions to glue the serde functions passed through the option structs with the rest of zenoh-pico.

This is how the end user is expected to implement the serialize function

```
int8_t serialize(int8_t (*write)(void *writer, const char* serialized,
size_t serialized_len), void *writer, void* ctx) {
  my_args* args = (my_args*)ctx;
  char buffer[512]; size_t buffer_len = 0;
  // serialization code for args here
  return write(writer, buffer, buffer_len);
}
```

The serialize function takes as an argument a pointer to a function that returns an int8_t which is a 1-byte signed integer to signal any error may arise from the serialize function. Such pointer to a function takes a pointer to a void, a pointer to a char array and a size_t that is the length of the char array. This is because this pointer to a function is the write() function, its implementation will write the serialized content of buffer in a RTC Slow RAM buffer we have seen before.

writer is there because write can not hold state. For example it could point to a struct that holds the start and capacity of the section I am allowing write to write to. Even if write does not need state yet (it might all be static), having that opaque pointer gives me the option to add some in the future if needed. An example of how the write function can be implemented is the following:

```
int8_t _write_drop_arg(void * writer, const char * serialized, int serialized_len){
    int8_t ret = 0;

    memcpy(*(uint8_t **)writer, &serialized_len, sizeof(serialized_len));
    *(uint8_t **)writer += sizeof(serialized_len);
    memcpy(*(uint8_t **)writer, serialized, serialized_len);
    *(uint8_t **)writer += serialized_len;

    return ret;
}
```

In my case, writer points inside the RTC Slow Memory buffer thus I need a **double pointer** to it to increment the position of the writer pointer after the serialization

took place. Doing so, the caller of _write_drop_arg obtains the **updated position** of the buffer.

This is the signature of the deserialize function

```
int8_t (*deserialize)(const char* serialized, int serialized_len, void** ctx)
```

It takes the array of the serialized struct, the length of the serialized struct and eventually some context information if needed. The context information is provided via a double pointer that way if the user needs to allocate again for their state, they can replace the ctx pointer with the result of the new allocation.

In order to easily manage the two new function pointers and **glue them** with the option struct and the rest of zenoh-pico I created the new type zp_serde_functions_t

```
typedef int8_t (*zp_serializer_t)(int8_t (*write)(void *writer, const char *serialized,
int serialized_len),void *writer, void *ctx);
typedef int8_t (*zp_deserializer_t)(const char* serialized, int serialized_len,
void **ctx);

typedef struct {
    zp_serializer_t serialize;
    zp_deserializer_t deserialize;
} zp_serde_functions_t;
```

In the next chapter I will delve deeper on how I glued the options with the rest of zenoh-pico.

4.2 Implementation

The implementation can be teared down to the **two main functions** added to the zenoh-pico's API: the **zp_prepare_to_sleep** which serializes into the RTC Slow Memory the session's structs, and the function **zp_wake_up()** that deserializes from the RTC Slow Memory the session's structs and rebuilds in the heap memory the zenoh session. Then I will discuss the most difficult errors encountered when developing zenoh-pico. At last, I will even discuss how I glued the serde functions to the rest of zenoh-pico via the option struct. Let us start by dissecting the function zp_prepare_to_sleep first then I will focus on the function zp_wake_up.

4.2.1 zp_prepare_to_sleep()

The zp_prepare_to_sleep() looks like the listing below:

LISTING 4.3: The function

```
int zp_prepare_to_sleep(z_owned_session_t zs){
    _serialize_z_transport_t(zs._value->_tp, transport);

    // Saving Zenoh PID
    memcpy(RTC_local_zid, zs._value->_local_zid.id, 16);

    // Saving session counters
    RTC_resource_id = zs._value->_resource_id;
    RTC_entity_id = zs._value->_entity_id;
    RTC_pull_id = zs._value->_pull_id;
```

```
    RTC_query_id = zs._value->_query_id;
    RTC_interest_id = zs._value->_interest_id;

    _serialize_z_resource_list_t(zs._value->_local_resources,
    local_resources);
    _serialize_z_resource_list_t(zs._value->_remote_resources,
    remote_resources);

    memset(local_subscriptions, 0, DIM_LOCAL_SUBSCRIPTIONS);
    memset(remote_subscriptions, 0, DIM_REMOTE_SUBSCRIPTIONS);
    _serialize_z_subscription_sptr_list_t(zs._value->_local_subscriptions,
    _write_subscription_local, local_subscriptions);
    _serialize_z_subscription_sptr_list_t(zs._value->_remote_subscriptions,
    _write_subscription_remote, remote_subscriptions);

    memset(local_questionable, 0, DIM_LOCAL_QUESTIONABLE);
    _serialize_z_questionable_sptr_list_t(zs._value->_local_questionable,
    _write_questionable_local, local_questionable);

    memset(pending_queries, 0, DIM_PENDING_QUERIES);
    _serialize_z_pending_query_list_t(zs._value->_pending_queries,
    _write_call_arg, _write_drop_arg, pending_queries);

    return 0;
}
```

As I can see from the listing of the zp_prepare_to_sleep, first I serialize the struct _z_transport_t, then the zenoh PID, the session counters and at last the lists. It is good practice to initialize to zero the destination buffers by using *memset* as has been done in the listing above. Saving the zenoh's PID and the session counters has been pretty straightforward to implement so let's delve deeper in finer details on how the **serialization** of the lists works.

Lists' serialization

For the lists to be serialized I followed the following pattern (in pseudo-code):

```
save_the_number_of_elements_in_the_list

while list NOT empty:
    save_each_element_of_the_list
    if_found_any_subelement_then_save_it
    list = list->next
```

The resulting code looks like this:

LISTING 4.4: An example

```
int _serialize_z_resource_list_t(_z_resource_list_t *list,
                                 uint8_t *resources){
    int ret = _Z_RES_OK;
    _z_resource_t *element;
    size_t no_of_elements = _z_resource_list_len(list);
```

```
    uint8_t *_buffer = resources;

    //Serialization no_of_elements _id _key._id _key._mapping._val
    //_key._suffix _refcount
    memcpy(_buffer, &no_of_elements, sizeof(size_t));
    _buffer += sizeof(size_t);

    _z_resource_list_t *iterate_list = list;
    while(!_z_resource_list_is_empty(iterate_list)){
        element = _z_resource_list_head(iterate_list);

        memcpy(_buffer, &element->_id, sizeof(size_t));
        _buffer += sizeof(size_t);

        memcpy(_buffer, &element->_key._id, sizeof(uint16_t));
        _buffer += sizeof(uint16_t);

        memcpy(_buffer, &element->_key._mapping._val, sizeof(uint16_t));
        _buffer += sizeof(uint16_t);

        strcpy((char *)_buffer, element->_key._suffix);
        _buffer += strlen(element->_key._suffix) + 1;

        memcpy(_buffer, &element->_refcount, sizeof(uint16_t));
        _buffer += sizeof(uint16_t);

        iterate_list = _z_resource_list_tail(iterate_list);
    }

    return ret;
}
```

As I can see from the listing above, first I get the **length** of the list then I save it on the buffer and I increment the pointer to the buffer with the size of the just saved element I.E. a size_t. Hence, now the buffer points to an empty space initialized with zeros. Next, I start **iterating** over the element of the list and I save every element and sub-element of the struct _z_resource_t **in this case**. Note how after every element and sub-element have been saved I increment the buffer to always point to a free space on it. Additionally, note how every element is fixed apart from the type *char** which is a string. The type *char** is the only varying type in size but luckily I have the tools to measure with precision and deterministically its length I.E. the *strcpy* and *strlen* since it is null-terminated.

Function pointers

The following 3 functions take at least one extra argument containing one function pointer:

- _serialize_z_subscription_sptr_list_t
- _serialize_z_questionable_sptr_list_t

- _serialize_z_pending_query_list_t

This is due to the fact that the functions may treat structs containing user provided arguments thus only the user knows how to serialize and deserialize those arguments that is where serde (**ser**ialize **de**serialize functions) functions, discussed in the previous chapter, come handy. In the following listing I report one of the above function as the principles are valid for each one of the remaining 2 functions. I chose _serialize_z_pending_query_list_t as it is a special case of the other 2 functions.

LISTING 4.5: The function

```
int _serialize_z_pending_query_list_t(_z_pending_query_list_t *list,
int8_t (*write_call_arg)(void *writer, const char *serialized,
int serialized_len),
int8_t (*write_drop_arg)(void *writer, const char *serialized,
int serialized_len),
uint8_t *pending_queries){
    int ret = _Z_RES_OK;
    _z_pending_query_t *element;

    size_t no_of_elements = _z_pending_query_list_len(list);
    uint8_t *_buffer = pending_queries;

    //Serialization
    memcpy(_buffer, &no_of_elements, sizeof(size_t));
    _buffer += sizeof(size_t);

    _z_subscription_sptr_list_t *iterate_list = list;
    while(!_z_pending_query_list_is_empty(iterate_list))
    {
        element = _z_pending_query_list_head(iterate_list);

        //_key._id _key._mapping._val _key._suffix _id _parameters _target
        //_consolidation _anykey _pending_replies _callback _dropper
        //_serialize _deserialize _call_arg_len _call_arg _drop_arg_len
        //_drop_arg
        memcpy(_buffer, &element->_key._id, sizeof(uint16_t));
        _buffer += sizeof(uint16_t);

        memcpy(_buffer, &element->_key._mapping._val, sizeof(uint16_t));
        _buffer += sizeof(uint16_t);

        memcpy(_buffer, element->_key._suffix,
            strlen(element->_key._suffix) + 1);
        _buffer += strlen(element->_key._suffix) + 1;

        memcpy(_buffer, &element->_id, sizeof(uint32_t));
        _buffer += sizeof(uint32_t);

        // Since there is no _parameters_len I assumed _parameters is
        // null-terminated.
        memcpy(_buffer, element->_parameters,
            strlen(element->_parameters) + 1);
```

```
        _buffer += strlen(element->_parameters) + 1;

        memcpy(_buffer, &element->_target, sizeof(z_query_target_t));
        _buffer += sizeof(z_query_target_t);

        memcpy(_buffer, &element->_consolidation,
            sizeof(z_query_consolidation_t));
        _buffer += sizeof(z_query_consolidation_t);

        memcpy(_buffer, &element->_anykey, sizeof(_Bool));
        _buffer += sizeof(_Bool);

        _serialize_z_pending_reply_list_t(element->_pending_replies,
            &_buffer);
        // &_buffer to update the buffer pointer from the callee to
        // the caller
        //(I.E. this function)

        memcpy(_buffer, &element->_callback, 4); //cannot be NULL
        _buffer += 4;

        // // troubleshooting IllegalInstruction
        if(element->_dropper != NULL) memcpy(_buffer, &element->_dropper, 4
        _buffer += 4;

        if(element->serde_functions.serialize != NULL)
            memcpy(_buffer, &element->serde_functions.serialize, 4);
        _buffer += 4;

        if(element->serde_functions.deserialize != NULL)
            memcpy(_buffer, &element->serde_functions.deserialize, 4);
        _buffer += 4;

        if(element->serde_functions.serialize != NULL)
            element->serde_functions.serialize(write_call_arg, &_buffer,
                element->_call_arg);

        if(element->serde_functions.serialize != NULL)
            element->serde_functions.serialize(write_drop_arg, &_buffer,
            element->_drop_arg);

        iterate_list = _z_pending_query_list_tail(iterate_list);
    }

    return ret;
}
```

The listing above does what is usually done to serialize lists – it saves the number of elements first, then all the elements and sub-elements of the list are serialized one after the other. What is different in this listing w.r.t. the previous listing showing the serialization is the following:

- in this listing we serialize a list inside a list because the element _z_pending_query_t contains a list called _pending_replies.
- in this listing we use the serde functions.

In order to serialize a list inside a list we had to leverage the double pointers to the serialization's buffer, indeed I call

```
_serialize_z_pending_reply_list_t(element->_pending_replies, &_buffer);
```

The _serialize_z_pending_reply_list_t dereference the variable _buffer (like this: *_buffer) each time it is needed to update the buffer pointer. At the end of the serialization process of this function, the buffer's pointer points to a buffer zone that is free. So, I can say that the callee updates the buffer pointer of the caller by using double pointers.

Additionally, in this function I use the serde functions given by the end-user to serialize and deserialize (in this case to serialize) its custom structs. The serde functions are given a value by the end-user thus I check if they are NULL or not. If they are not NULL that means the user gave a value to the serde functions hence the element contains user-provided structs. In the latter case, I **invoke** the serialization function and keep updated the buffer pointer by giving a pointer to the buffer's pointer.

```
if(element->serde_functions.serialize != NULL)
    element->serde_functions.serialize(write_call_arg, &_buffer, element->_call_arg);

if(element->serde_functions.serialize != NULL)
    element->serde_functions.serialize(write_drop_arg, &_buffer, element->_drop_arg);
```

Before using the serde functions, I check whether the serde functions are not NULL, if so I save the function pointers in the serialization buffer so I can retrieve after the DEEPSLEEP_RESET the function pointer to both the serialization function but most importantly the deserialization function.

```
if(element->serde_functions.serialize != NULL)
    memcpy(_buffer, &element->serde_functions.serialize, 4);
_buffer += 4;

if(element->serde_functions.deserialize != NULL)
    memcpy(_buffer, &element->serde_functions.deserialize, 4);
_buffer += 4;
```

Note that the size of the function pointers is 4 bytes as we are on a 32-bit addressing system I.E. address of 4 bytes (32-bit).

_z_transport_t serialization

```
int _serialize_z_transport_t(_z_transport_t tp, uint8_t *transport){
[...]
    memcpy(_buffer, &link._open_f, 4);
    _buffer += 4;
    memcpy(_buffer, &link._listen_f, 4);
```

```
    _buffer += 4;
    memcpy(_buffer, &link._close_f, 4);
    _buffer += 4;
    memcpy(_buffer, &link._write_f, 4);
    _buffer += 4;
    memcpy(_buffer, &link._write_all_f, 4);
    _buffer += 4;
    memcpy(_buffer, &link._read_f, 4);
    _buffer += 4;
    memcpy(_buffer, &link._read_exact_f, 4);
    _buffer += 4;
    memcpy(_buffer, &link._free_f, 4);
    _buffer += 4;
[...]
}
```

What is interesting of the function _serialize_z_transport_t is that at a certain point in time it must save function pointers in the RTC Slow Memory. Such function pointers have an address like 0x4XXXXXXX. You might be wonder why, between deep sleeping resets, the function pointers are **stable** in memory thus not changing. This is due to the embedded world favoring **static linking** that means the function pointers, called labels, from the different libraries are all statically linked thus their pointers are not changing in memory between deep sleeping resets and the labels are translated into an address stable in memory. This is good for our use case because no dynamic linking means once the linker has created the final firmware's image all the functions will have their pointer pointing always to the same zone of memory. Moreover, note that the function _open_f will be used by the zp_wake_up() to bind to the previously used UDP port.

4.2.2 zp_wake_up()

The function zp_wake_up() does something more than the function zp_prepare_to_sleep() does. It does:

- deserializes the session from the RTC Slow Memory
- it recreates the session in the heap memory
- it binds to the previously used UDP port
- it assigns the session field of the struct _z_transport_t
- it initializes the struct _z_transport_t which includes its buffers and its mutex

In this subsection I will discuss in details what and how the zp_wake_up() performs its tasks.

First, the zp_wake_up() looks like:

LISTING 4.6: The function

```
z_owned_session_t zp_wake_up(){
    z_owned_session_t zs = {
        ._value = (_z_session_t *)z_malloc(sizeof(_z_session_t))
        };
    memset(zs._value, 0, sizeof(_z_session_t));
```

```
    if (zs._value != NULL) {
        _deserialize_z_transport_t(&zs._value->_tp, transport);
        zs._value->_tp._transport._unicast._session = zs._value;
        // Restoring Zenoh PID
        memcpy(zs._value->_local_zid.id, RTC_local_zid, 16);

        // Restoring session counters
        zs._value->_resource_id = RTC_resource_id;
        zs._value->_entity_id = RTC_entity_id;
        zs._value->_pull_id = RTC_pull_id;
        zs._value->_query_id = RTC_query_id;
        zs._value->_interest_id = RTC_interest_id;

        zs._value->_local_resources =
        _deserialize_z_resource_list_t(local_resources);
        zs._value->_remote_resources =
        _deserialize_z_resource_list_t(remote_resources);

        zs._value->_local_subscriptions =
        _deserialize_z_subscription_sptr_list_t(local_subscriptions);
        zs._value->_remote_subscriptions =
        _deserialize_z_subscription_sptr_list_t(remote_subscriptions);

        zs._value->_local_questionable =
        _deserialize_z_questionable_sptr_list_t(local_questionable);

        zs._value->_pending_queries =
        _deserialize_z_pending_query_list_t(pending_queries);

        __init_transport_t(&zs._value->_tp);

        // open UDP socket with the same port
        zs._value->_tp._transport._unicast._link._open_f
        (&zs._value->_tp._transport._unicast._link);
    }
    return zs;
}
```

As I can see from the listing of the zp_wake_up(), first I **deserialize** the struct _z_transport_t, then I **assign** the session field of the transport unicast field, then I **restore** the zenoh PID, the session counters, the lists, I **initialize** the _z_transport_t, I **bind** to the UDP socket with the previously used port. Restoring the zenoh's PID and the session counters has been pretty straightforward to implement so let's delve deeper in finer details on how the deserialization of the lists works, then we will see how the initialization of the _z_transport_t works and at last how the UDP socket is bound to the previously used port.

List's deserialization

The dual of the serialization is the deserialization. The pseudo-code used to deserialize and rebuild in the heap memory the session's structs looks like this:

```
read_how_many_no_of_elements_previously_stored

iterates_no_of_elements_times:
    read_element
    create_the_heap_in_memory
    fill_the_heap_in_memory
    read_its_sub_elements
    create_the_heap_in_memory
    fill_the_heap_in_memory
```

Here I report the counterpart of the function used to serialize: deserialize.

LISTING 4.7: An example

```
_z_resource_list_t * _deserialize_z_resource_list_t(uint8_t *buffer){
    _z_resource_list_t *list = _z_resource_list_new();
    size_t no_of_elements;
    uint8_t * _buffer = buffer;

    //Deserialization no_of_elements _id _key._id _key._mapping._val
    // _key._suffix _refcount
    memcpy(&no_of_elements, _buffer, sizeof(size_t));
    _buffer += sizeof(size_t);

    for(size_t i = 0; i < no_of_elements; i++){
        _z_resource_t *element =
            (_z_resource_t *)malloc(sizeof(_z_resource_t));

        memcpy(&element->_id, _buffer, sizeof(size_t));
        _buffer += sizeof(size_t);

        element->_key = *((_z_keyexpr_t *)malloc(sizeof(_z_keyexpr_t)));

        memcpy(&element->_key._id, _buffer, sizeof(uint16_t));
        _buffer += sizeof(uint16_t);

        memcpy(&element->_key._mapping._val, _buffer, sizeof(uint16_t));
        _buffer += sizeof(uint16_t);

        size_t len = strlen((char *)_buffer) + 1;

        element->_key._suffix = malloc(len);
        memcpy(element->_key._suffix, _buffer, len);
        _buffer += len;

        memcpy(&element->_refcount, _buffer, sizeof(uint16_t));
        _buffer += sizeof(uint16_t);

        list = _z_resource_list_push(list, element);
    }
```

```
    return list;
}
```

Simply, in the above function I restore the previously saved "number of elements" number, then I iterate n times deserializing then building step by step the struct, in this case the struct is _z_resource_t. Obviously, the buffer must point to the right position in order to restore the right value from the RTC Slow Memory. At last, I push the just created and initialized element in the list and I continue the iteration until I finish the n-th iteration.

__init_transport_t

The __init_transport_t function has been very simple to implement, I just copied and pasted the content of **src/transport/transport.c::_z_transport_unicast**. What the code does is:

1. initialize the mutexes.
2. initialize the read and write buffers.
3. initialize the defragmentation buffers.

And, it fallbacks in case of allocation failure giving to the variable ret the proper value.

Bound UDP socket to the previously used port

In order for the zp_wake_up() to bound to the previously used port I had to call the saved function pointer _open_f and edit it.

```
// open UDP socket with the same port
zs._value->_tp._transport._unicast._link._open_f
    (&zs._value->_tp._transport._unicast._link);
```

The function _open_f is just an alias for _z_f_link_open_udp_unicast

```
zl->_open_f = _z_f_link_open_udp_unicast;
```

which calls our function of interest _z_open_udp_unicast.

```
_z_open_udp_unicast(&self->_socket._udp._sock, self->_socket._udp._rep, tout);
```

Let us see how I edited the function _z_open_udp_unicast for the socket to correctly bind to the previously used UDP port:

LISTING 4.8: The function _z_open_udp_unicast

```
int8_t _z_open_udp_unicast(_z_sys_net_socket_t *sock,
const _z_sys_net_endpoint_t rep, uint32_t tout) {
    int8_t ret = _Z_RES_OK;

    struct sockaddr_in sin;
    sin.sin_port = 0;
    sin.sin_family = rep._iptcp->ai_family;
    sin.sin_addr.s_addr = INADDR_ANY;
```

```
socklen_t slen = sizeof(sin);

sock->_fd = socket(rep._iptcp->ai_family, rep._iptcp->ai_socktype,
rep._iptcp->ai_protocol);

if(RTC_source_port == 0){// first time that the socket is used
    if (sock->_fd != -1) {
        z_time_t tv;
        tv.tv_sec = tout / (uint32_t)1000;
        tv.tv_usec = (tout % (uint32_t)1000) * (uint32_t)1000;
        if ((ret == _Z_RES_OK) && (setsockopt(sock->_fd, SOL_SOCKET,
                        SO_RCVTIMEO, (char *)&tv, sizeof(tv)) < 0)) {
            ret = _Z_ERR_GENERIC;
        }

        if (ret != _Z_RES_OK) {
            close(sock->_fd);
        }

        // get currently used port
        bind(sock->_fd, (struct sockaddr *)&sin, slen);
        if(getsockname(sock->_fd, (struct sockaddr *)&sin, &slen) < 0){
            ESP_LOGI("ERROR",
                "_z_open_udp_unicast:␣unable␣to␣getsockname()\n");
            ret = _Z_ERR_GENERIC;
        }
        RTC_source_port = ntohs(sin.sin_port);

        if (ret != _Z_RES_OK) {
            close(sock->_fd);
        }
    } else {
        ret = _Z_ERR_GENERIC;
    }
} else {// bind to the previously used port
    if(sock->_fd != -1){
        z_time_t tv;
        tv.tv_sec = tout / (uint32_t)1000;
        tv.tv_usec = (tout % (uint32_t)1000) * (uint32_t)1000;
        if ((ret == _Z_RES_OK) && (setsockopt(sock->_fd, SOL_SOCKET,
            SO_RCVTIMEO, (char *)&tv, sizeof(tv)) < 0)) {
            ret = _Z_ERR_GENERIC;
        }

        sin.sin_port = htons(RTC_source_port);
        bind(sock->_fd, (struct sockaddr *)&sin, slen);
    } else {
        ret = _Z_ERR_GENERIC;
    }
}
```

```
    return ret;
}
```

In the listing above, first I prepare the general struct sockaddr_in to store information about the socket.

```
struct sockaddr_in sin;
sin.sin_port = 0;
sin.sin_family = rep._iptcp->ai_family;
sin.sin_addr.s_addr = INADDR_ANY;
socklen_t slen = sizeof(sin);
```

Note that the port is initialized to zero meaning has not been assigned any value to it as the port 0 is not a well-known port. To the sin_family is assigned the value rep._iptcp->ai_family and the source address s_addr is INADDR_ANY (0.0.0.0) meaning it assumes the address of any network interface present in the microcontroller. After that, a network socket is opened with the given parameters.

Then, I check whether RTC_source_port, a variable stored in the RTC Slow Memory, is 0 or not. If it is zero that means is the first time that the socket is used. In such case some timers are set with the function *setsockopt* and I get the used UDP port by the socket with the function *getsockname*. The information about the socket, along with the used port, is stored in the struct sockaddr_in. At runtime the value of sin.sin_port **was zero** even after the socket was initialized and the *getsockname* performed. This is due to the fact that the freeRTOS's operating system **assigns** a UDP port number to the socket **only upon packets sent**. Since I needed that information earlier in the code I had to *bind* the socket to its file descriptor thus receiving a UDP port number before any packet was sent on the network. The used port is saved in the variable RTC_source_port for the next DEEPSLEEP_RESET to switch to the if's body containing the code to restore the previous UDP port.

Indeed, if the RTC_source_port is different than zero that means the socket had previously binded to a UDP port. Such port is stored in the RTC Slow Memory's variable RTC_source_port. Then, I *bind* the file descriptor to the previously used port assigning to the variable sin.sin_port the value RTC_source_port.

4.2.3 The most difficult errors

Having to deal with serialization and deserialization brought **plenty of errors** during the development phase. Here I discuss the **most difficult errors** I had to deal with in the development phase of the project. Consider that the development phase had proceeded by testing time by time the single serialization and deserialization functions as they were ready to be tested and not, like discussed in this work, taken separately just for the tidyness' sake.

assert_failed: block_trim_free tlsf.c:502

One of the most annoying errors I had to deal with was the assert failed: block_trim_free tlsf.c:502 (block_is_free(block) && "block must be free")

```
assert failed:
    block_trim_free tlsf.c:502 (block_is_free(block) && "block must be free")


Backtrace:
```

```
0x400819aa:0x3ffbfd00 0x40088ac9:0x3ffbfd20 0x4008f6b9:0x3ffbfd40
0x4008d5d4:0x3ffbfe60 0x4008d073:0x3ffbfe80 0x40082951:0x3ffbfea0
0x400829af:0x3ffbfec0 0x400829e6:0x3ffbfee0 0x4008f6c9:0x3ffbff00
0x400821fd:0x3ffbff20 0x4015589d:0x3ffbff40 0x4009245d:0x3ffbff70
0x40093258:0x3ffbffa0 0x400936e1:0x3ffbffe0 0x40093797:0x3ffc0020
0x40090fca:0x3ffc0060 0x4008bc65:0x3ffc0090
```

This error happened when I was using the function _deserialize_z_questionable_sptr_list_t for the example z_queryable.c. Note that at the beginning of the development phase I was calling the two functions zp_prepare_to_sleep() and zp_wake_up() without a DEEPSLEEP_RESET between them for testing purposes. The code looked something like:

```
zp_prepare_to_sleep(s);

z_owned_session_t zs = zp_wake_up();
```

The error occurred just sometimes after a random time at the end of the code execution for no apparent reason and dissecting the backtrace via the usage of the command

```
/path/to/xtensa-esp32-elf-addr2line -pfiaC -e firmware.elf 0x400819aa:0x3ffbfd00
```

did not give much insight about the reason of the error. Zeroing the buffer containing the serialized list for queryables seemed to solve the problem but I had just zeroed what I had previously saved with the zp_prepare_to_sleep() so zeroing the buffer did not make much sense. I asked for information about the error on the ESP32's forum and the user ESP_Sprite told me it was a buffer overflow or something similar. Then I started again my research on the error and found the following piece of code:

```
element = (_z_questionable_sptr_t *) malloc(sizeof(_z_questionable_sptr_t));
memset(element->ptr, 0, sizeof(_z_questionable_t));

element->ptr->_key = *((_z_keyexpr_t *)malloc(sizeof(_z_keyexpr_t)));
```

Do you note something strange here? Yes, by what ESP_Sprite told me I noticed I was setting to zero a zone of memory that was not *malloc*'d. I corrected the error with the following edit:

```
element = (_z_questionable_sptr_t *) malloc(sizeof(_z_questionable_sptr_t));
memset(element, 0, sizeof(_z_questionable_sptr_t));

element->ptr = (_z_questionable_t *)malloc(sizeof(_z_questionable_t));
memset(element->ptr, 0, sizeof(_z_questionable_t));
```

After such edit, the error assert_failed: block_trim_free tlsf.c:502 disappeared.

InstrFetchProhibited

I stumbled upon this kind of error very frequently during the development phase. From the ESP32's documentation: "This CPU exception indicates that the CPU could not read an instruction because the address of the instruction does not belong to a valid region in instruction RAM or ROM.

Usually, this means an attempt to call a function pointer, which does not point to valid code. PC (Program Counter) register can be used as an indicator: it will be zero or will contain a garbage value (not 0x4xxxxxx)."

Here, I discuss what does it mean to not check the errors in code. See the following code of the function _serialize_z_pending_query_list_t:

```
if(element->_call_arg != NULL) element->serde_functions.serialize(write_call_arg,
&_buffer, element->_call_arg);

if(element->_drop_arg != NULL) element->serde_functions.serialize(write_drop_arg,
&_buffer, element->_drop_arg);
```

It should not happen that element->_call_arg != NULL and element->serde_functions.serialize == NULL because the user is expected to provide serde functions along with their custom structs. I did not check this case so the above code raises a InstrFetchProhibited CPU exception when element->_call_arg != NULL because it is invoked the function element->serde_functions.serialize which by looking at the PC of the register dump it was around the value 9500 (not 0x4XXXXXXX). The following edit solves the error:

```
if(element->_call_arg != NULL && element->serde_functions.serialize != NULL)
    element->serde_functions.serialize(write_call_arg, &_buffer, element->_call_arg);

if(element->_drop_arg != NULL && element->serde_functions.serialize != NULL)
    element->serde_functions.serialize(write_drop_arg, &_buffer, element->_drop_arg);
```

Working on copies: _deserialize_z_transport_t

I thought I was mastering pointers very well, I was wrong and this subsection demonstrates so: when I was working on the function _deserialize_z_transport_t, due to the many sub-elements it is composed of, I decided to structure the code like I did for the serialization counterpart by creating a struct variable for each sub-element and work on each struct variable for tidyness' sake, like this:

```
[...]
_z_transport_unicast_t unicast = res._transport._unicast;

memcpy(unicast._remote_zid.id, _buffer, 16);
_buffer += 16*sizeof(uint8_t);

memcpy(&unicast._sn_res, _buffer, sizeof(_z_zint_t));
_buffer += sizeof(_z_zint_t);
memcpy(&unicast._sn_tx_reliable, _buffer, sizeof(_z_zint_t));
_buffer += sizeof(_z_zint_t);
memcpy(&unicast._sn_tx_best_effort, _buffer, sizeof(_z_zint_t));
_buffer += sizeof(_z_zint_t);
memcpy(&unicast._sn_rx_reliable, _buffer, sizeof(_z_zint_t));
_buffer += sizeof(_z_zint_t);
memcpy(&unicast._sn_rx_best_effort, _buffer, sizeof(_z_zint_t));
_buffer += sizeof(_z_zint_t);
memcpy(&unicast._lease, _buffer, sizeof(_z_zint_t));
_buffer += sizeof(_z_zint_t);
```

```
memcpy(&unicast._received, _buffer, sizeof(_Bool));
_buffer += sizeof(_Bool);
memcpy(&unicast._transmitted, _buffer, sizeof(_Bool));
_buffer += sizeof(_Bool);
[...]
```

Do you see the problem? In the above code the assignation

```
_z_transport_unicast_t unicast = res._transport._unicast;
```

is like doing

```
int a = 5;
int b = 6;

a = b;
```

thus the future changes in the variable "unicast" do not affect the struct variable res._transport._unicast. I should have used a pointer but I opted out to rewrite the whole res._transport._unicast anywhere it was needed.

```
memcpy(tp->_transport._unicast._remote_zid.id, _buffer, 16);
_buffer += 16*sizeof(uint8_t);

memcpy(&tp->_transport._unicast._sn_res, _buffer, sizeof(_z_zint_t));
_buffer += sizeof(_z_zint_t);
memcpy(&tp->_transport._unicast._sn_tx_reliable, _buffer, sizeof(_z_zint_t));
_buffer += sizeof(_z_zint_t);
memcpy(&tp->_transport._unicast._sn_tx_best_effort, _buffer, sizeof(_z_zint_t));
_buffer += sizeof(_z_zint_t);
memcpy(&tp->_transport._unicast._sn_rx_reliable, _buffer, sizeof(_z_zint_t));
_buffer += sizeof(_z_zint_t);
memcpy(&tp->_transport._unicast._sn_rx_best_effort, _buffer, sizeof(_z_zint_t));
_buffer += sizeof(_z_zint_t);
memcpy(&tp->_transport._unicast._lease, _buffer, sizeof(_z_zint_t));
_buffer += sizeof(_z_zint_t);
```

LoadProhibited, StoreProhibited

From the ESP32's documentation: "These CPU exceptions happen when an application attempts to read from or write to an invalid memory location. The address which has been written/read is found in the EXCVADDR register in the register dump. If this address is zero, it usually means that the application has attempted to dereference a NULL pointer. If this address is close to zero, it usually means that the application has attempted to access a member of a structure, but the pointer to the structure is NULL. If this address is something else (garbage value, not in 0x3fxxxxxx - 0x6xxxxxx range), it likely means that the pointer used to access the data is either not initialized or has been corrupted."

The register dump looks like:

```
Guru Meditation Error: Core  0 panic'ed (LoadStoreError). Exception was unhandled.
```

```
Core  0 register dump:
PC      : 0x400d68c2  PS      : 0x00060c30  A0      : 0x800d6a15  A1      : 0x3ffba020
A2      : 0x3ffc5904  A3      : 0x400d65b8  A4      : 0x50000224  A5      : 0x3ffc58e4
A6      : 0x00000000  A7      : 0x00000000  A8      : 0x400d1e5c  A9      : 0x3ffb9fd0
A10     : 0x00000022  A11     : 0x3ffba1e0  A12     : 0x000000ff  A13     : 0x0000ff00
A14     : 0x00ff0000  A15     : 0xff000000  SAR     : 0x00000004  EXCCAUSE: 0x00000003
EXCVADDR: 0x400d1e5c  LBEG    : 0x400014fd  LEND    : 0x4000150d  LCOUNT  : 0xffffffff


Backtrace: 0x400d68bf:0x3ffba020 0x400d6a12:0x3ffba050 0x400d2232:0x3ffba070
0x401573b7:0x3ffba0c0 0x4008bc65:0x3ffba0f0
```

It happened to me that I received a LoadProhibited error with EXCVADDR equal to, more or less, the value 1000. That was because at a certain point in the code I was accessing a field of the session in the zs._value->_tp._transport._unicast._session which I forgot to initialize. The following edit in the function zp_wake_up() solved the problem:

```
zs._value->_tp._transport._unicast._session = zs._value;
```

4.2.4 Glueing the serde functions with the rest of zenoh-pico

The process of integrating the new serialization and deserialization functions has been very intrusive to zenoh-pico.

I added the serde type to the proper structs which are:

- _z_subscription_t
- _z_questionable_t
- _z_pending_query_t

Since zenoh-pico uses the pattern options to provide feature extensibility, I added the serde functions type to the following zenoh-pico options because their respective operations operate on user-provided data to be serde for deep-sleeping's purpose.

- z_subscriber_options_t
- z_pull_subscriber_options_t
- z_queryable_options_t
- z_get_options_t

Such options will be used by the end user to provide user-defined serde functions for user-defined data. Then, I changed the _z_declare_subscriber signature in order to be able to fit the serde functions in and assign the passed serde functions to the inner struct _z_subscription_t. The function is a private one so is possible to add/remove parameters without breaking the API. After that, I changed the function's signature of _z_query to accept zp_serde_functions_t as an argument and the serde type is assigned to the struct type _z_pending_query_t. The same goes for _z_questionable_t with the function _z_declare_queryable.

EXAMPLE	PREPARE_TO_SLEEP [μs]	WAKE_UP [μs]	RAM [bytes]
z_get.c	883	16668	128392
z_pub.c	1074	16732	128480
z_pull.c	1225	16660	128576
z_sub.c	1058	16641	128464

TABLE 4.1: Baseline measurements.

4.3 Experimental evaluations

The experimental evaluations have been conducted on the **4 examples** I created as a demonstration: z_get.c, z_pub.c, z_pull.c, z_sub.c. Must be noted that as of today is still missing the example z_queryable.c as it has a minor issue causing it to crash on an InstructionFetchError after a couple of DEEPSLEEP_RESET.

4.3.1 Baseline measurements

I measured the time needed to prepare to sleep, the time needed to wake up, and the RAM occupied by the session on each one of the above examples. These are of interests because the session is composed, among many other things, by different lists and with each one of the example I get a gauge of how much RAM is occupied by the different lists (and the session itself) and the time needed to serialize and deserialize the different lists. The **results** are reported in the table 4.1:

To get the results in table 4.1, in order to measure the time needed, I simply **wrapped** the functions of interest I.E. zp_prepare_to_sleep and zp_wake_up with the system call gettimeofday() which measures the microseconds ([us]) since the Unix epoch time (1st January 1970). Note that gettimeofday() has a slightly higher overhead w.r.t. the function esp_timer_get_time() but at the time of testing I did not know the existence of such function.

```
struct timeval tv_start;
struct timeval tv_stop;

gettimeofday(&tv_start, NULL);
zp_wake_up();/zp_prepare_to_sleep();
gettimeofday(&tv_stop, NULL);
long elapsed_time = tv_stop.tv_sec * 1000000 + tv_stop.tv_usec -
(tv_start.tv_sec * 1000000 + tv_start.tv_usec);
printf("elapsed time: %ld[us]\n", elapsed_time);
```

Whereas to get the size of DRAM allocated by the zp_wake_up:

```
before = heap_caps_get_free_size(MALLOC_CAP_8BIT);
zp_wake_up();
after = heap_caps_get_free_size(MALLOC_CAP_8BIT);
printf("bytes allocated %zd\n", before - after);
```

I must consider that the experiments have been conducted on lists of at least 1 element and at maximum 1 element. Taken that, I see that the zp_wake_up() function almost takes a constant time around 16ms and 17ms to deserialize then create the session in the heap memory whereas the zp_prepare_to_sleep() takes between

1.04ms and 1.225ms to serialize the session's struct in the RTC Slow Memory. The difference between the zp_prepare_to_sleep() and the zp_wake_up() is of a factor of x10000. This is **due to the fact that** the zp_wake_up(), other than deserializing the session, has to create the session in the heap memory by calling many times the onerous system call malloc().

4.3.2 Varying the load of the session

Here I measured how long does it take for the zp_prepare_to_sleep() and the zp_wake_up() to serialize and deserialize the session **varying the session's size** (I.E. its lists as the other elements are not varying).

I started my measurements with the example **z_pub.c** by creating 6 different topics to publish to named "demo/example/zenoh-pico-pub-{number}", with a lower number of publication topics there were no notable differences. I noticed the zp_prepare_to_sleep() time increased to 1.102ms (30μs of difference w.r.t. table 4.1) whereas the zp_wake_up() time is almost stable at 16.7ms. I repeated the same measurements with 10 different publication topics and the zp_prepare_to_sleep() time increased to 1.251ms while the zp_wake_up() time is marked at around 17.5ms which are respectively around 200μs and 800μs more w.r.t. the measurements taken for the z_pub.c on the table 4.1.

For what regards the example **z_sub.c** I subscribed to 6 different topics first, then to 10 different topics as I did with the z_pub.c example but this time with the subscriptions instead of the publications. I declared the different subscribers on "demo/example/zenoh-pico-pub-{number}". For the 6 different topics, the zp_prepare_to_sleep() time increased to 1.271ms whereas the zp_wake_up() increased to 16.691ms which are respectively 213μs more and 50μs more w.r.t. the measurements taken for z_sub.c on the table 4.1. For the 10 different topics I incurred on an InstrFetchProhibited error due to the fact that I was reading from outside the buffer and the runtime did not raise any error about reading from outside the buffer. I was reading the value 0xff000000 that was assigned to the function of the callback which resulted in an error when it was called (indeed, on the stack backtrace of the error the PC value was 0xff000000). In order to solve the error, I incremented the RTC Slow Memory buffer's size to 1024 bytes by dimensioning the #define DIM_LOCAL_SUBSCRIPTIONS 1024. For what concerns the measurements, I measured 1.378ms for the zp_prepare_to_sleep() and 16.738ms for the zp_wake_up() which are respectively 320μs and 97μs more w.r.t. the baseline measurements taken for the z_sub.c in the table 4.1.

For the **z_get.c** I sent the first 6 queries to "demo/example/zenoh-pico-pub-{number}". It took to the zp_prepare_to_sleep 883μs and 16.663ms to the zp_wake_up which are, w.r.t. table 4.1, almost the same results. This is due to the fact that the session's list affected by the z_get.c is the list _pending_queries which are the queries emitted to the network but have not received a reply final. Since before the end of the deep sleep cycle the reply final are received the list is empty thus it takes to our 2 functions the same time as the baseline measurements taken for the example z_get.c. The same logic is valid for sending 10 different queries to "demo/example/zenoh-pico-pub-{number}".

At last, for the example **z_pull.c**, I declared subscriber on 6 different topics named, as always, "demo/example/zenoh-pico-pub-{number}". For the zp_prepare_to_sleep it takes 1.486ms whereas for the zp_wake_up it takes 16.656ms which are respectively 261μs more and almost the same for what concerns the zp_wake_up w.r.t. table 4.1. Then, I declared 10 different topics named "demo/example/zenoh-pico-pub-{number}". It took 1.717ms for the zp_prepare_to_sleep and 16.528ms for the

EXAMPLE	6 topics [μs]	10 topics [μs]
z_get.c	0	0
z_pub.c	28	177
z_pull.c	261	492
z_sub.c	213	320

TABLE 4.2: Time differences zp_prepare_to_sleep().

EXAMPLE	6 topics [μs]	10 topics [μs]
z_get.c	0	0
z_pub.c	0	800
z_pull.c	0	0
z_sub.c	50	97

TABLE 4.3: Time differences zp_wake_up().

zp_wake_up which are respectively 492μs more and almost the same for what concerns the zp_wake_up. With this example, the zp_wake_up works fine but the third z_declare_pull_subscriber fails and it is unable to declare the subscriber. I did not investigate deeper due to lack of time.

To sum up the obtained results for each example, in the table 4.2 I report all the differences in time for what concerns the function zp_prepare_to_sleep w.r.t. baseline measurements on table 4.1. In the table 4.3 I report all the differences in time for what concerns the function zp_wake_up() w.r.t. baseline measurements on table 4.1.

By looking at the table 4.3 I do not notice any great difference, indeed almost all the fields are 0s. This is quite strange as I expected a remarkable increment of the values in terms of time in us needed because of the *mallocs*. It might be due to caching I guess, really, I have no clue about.

On the other hand, by looking at the table 4.2 I notice a great increment between the usage of 6 topics first then 10 topics. In particular, apart from the example z_get.c, the example z_pub.c has an increment of x6.32, the z_pull.c has an increment of x1.88, the z_sub.c has an increment of x1.50. The major increment has been acquired by the z_pub.c which is x6.32 times the time for 6 topics.

4.3.3 Size

In this section I reported the **total image size** of the firmware produced by each one of the examples for the deep sleeping and not deep sleeping. The results have been annotated in the table below:

EXAMPLE	Deep Sleeping [bytes]	No Deep Sleeping [bytes]	%
z_get.c	816309	773661	5.22
z_pub.c	816053	773657	5.20
z_pull.c	816805	774109	5.23
z_sub.c	816677	773953	5.23

The measures have been detected by the end of the building process of PlatformIO which outputs the SRAM used and the flash size of the final firmware's

image. Consider that the flash size of the no deep sleeping has been taken on zenoh-pico v0.11.0.0 whereas the flash size of the deep sleeping has been taken on zenoh-pico v0.10.0.0 but at the end does not change that much for what concerns the used zenoh-pico version. These measurements are very important as constrained devices in the embedded systems world are **constrained** in flash size. Indeed, with my deep sleeping implementation I just occupy an average of 5.22% more in the flash than Zenoh Pico without deep sleeping which is great.

4.3.4 Average Power

We have seen in the chapter 2 a formula to compute the average power consumed by a microcontroller, reported here:

$$P_{avg} = P_{always\_on} + P_{sleep} + \frac{E_{active}}{T_{wkup}}$$

The problem is the above formula needs to know the time to wake up which is pretty complex to acquire in the ESP32 but I studied a simpler method to gain the same result. But first, let us discuss why we need the average power.

Usually, microcontrollers are connected to **external batteries** which provide energy to the MCU. The electric charge (I.E. an energy) stored in batteries is measured in mAh (milli Ampere-hour) and we can calculate the time needed to drain the electric charge in the battery by remembering that

$$E = P * t$$

thus

$$t = \frac{E}{P}$$

Moreover, we know that

$$E_{battery} = mAh * V * 3600s$$

where mAh is the capacity of the battery, V is the voltage at which the ESP32 is powered (it is 5V) and 3600s are the seconds in an hour. Given the current consumed at each sleeping mode by looking at the figure 2.16, we can compute the time needed to drain the battery in normal mode and in deep sleeping mode because

$$P = V * I$$

250 mAh

We suppose to have an external battery with a capacity of 250mAh @ 5V and our ESP32 is operating with a dual core chips at a normal speed of 80MHz (thus current of 20mA - 31mA). Let us do the math:

$$t_{drain} = \frac{E_{battery}}{P_{MCU}} = \frac{250mAh * 5V * 3600s}{V_{MCU} * I_{MCU}} = \frac{250mAh * 5V * 3600s}{5V * 20mA} = 45000s = 750h$$

Our ESP32 will drain 250mAh of battery in 750 hours or in 31.25 days.

Now, let us do the same but we will take as an example the deep sleeping example z_pub.c, we will compute the average power then we will compute the t_{drain}. The

z_pub.c takes a certain amount of normal activity where it connects to the wifi, establishes the zenoh session then publish data before going to deep sleep for 3000ms. Then it wakes up, it connects to the wifi, it restores the zenoh session then publish data before going to sleep for 3000ms for a total normal activity of around 6193785us I.E. 6.193785s. In the long term what interests us is the P_{avg} without the zenoh session establishment thus the zenoh session wake up. This is due to the fact that, apart from the first cycle, in the rest of the cycles there is no need to perform the session establishment because of the zenoh wake up. We consider in deep sleeping a current of 10μA I.E. the Amperage of when only the RTC Timer and the RTC Memories are powered on. Let us compute the P_{avg}:

$$P_{avg} = \frac{V_{active} * I_{active} * 6.193785s + V_{deep\_sleeping} * I_{deep\_sleeping} * 3s}{6.193785s + 3s}$$

$$= \frac{5V * 20mA * 6.193785s + 5V * 10uA * 3s}{9.193785s} = 0.067W = 67mW$$

as we can see the P_{avg} is a weighted average based on the time in normal mode and the time in deep sleeping mode. Then,

$$t_{drain} = \frac{E_{battery}}{P_{avg}} = \frac{250mAh * 5V * 3600s}{67mW} = 67164.18s = 1119.4h = 46.64days$$

Notice how, introducing a short deep sleeping of 3s we see an increment in the autonomy of the battery of **slightly more than 2 weeks**. The deep sleeping is incredibly important for constrained devices leveraging the power of a battery thus the importance in having such support in zenoh-pico.

4.3.5 Performance Monitor

Performance monitor technique is used to profile functions. It is of good importance as it gives an **insight** on the total cycles used, the memory reads/writes performed, the total instructions. These basics performance counters are of interest to us because the higher they are the more battery is used so a further step would be to decrease somehow the performance counters above.

I tried to measure the performance counters but in order to measure the performance counters of a function such function must be executed and in my case I cannot execute the function without actually having a z_owned_session_t type (for the zp_prepare_to_sleep) or having an established transport (for the zp_wake_up) thus resulting in runtime errors with Core 0 panics.

Conclusion

Through this master thesis we have seen a lot of interesting concepts for what concerns the world of the Internet of Things. We gave a precise definition of what an embedded system is then we deeply studied the board ESP32 in particular its technical details. We introduced the environments and tools used to operate on embedded systems' boards. Then we learnt the basics and most important concepts about Zenoh and Zenoh-Pico, we delved deeper in the Zenoh-Pico's architecture and now we have a clearer idea on what a middleware is and what are its purposes along with how Zenoh-Pico is structured internally. We presented the project for this master thesis which contains the pillar concepts on which the master thesis project has been built. We have delved deeper in the implementation rationale and details of the implementation then we conducted some experimental evaluations on the implementation.

Obtained results

The experimental evaluations have been conducted on the **4 working examples** I created as a demonstration: the **z_get.c**, **z_pub.c**, **z_pull.c**, and **z_sub.c**. The only one not working example is the **z_queryable.c** example which has a minor issue causing a crash on an InstructionFetchError error after a couple of DEEPSLEEP_RESET sent by the CPU.

In order to get the main results about the time taken I used the POSIX function *gettimeofday()* which returns the system time in μs and has a resolution of 1μs. I use the function *gettimeofday()* before the target function and after the target function saving the result in two *struct timeval* then making the difference to measure, in μs, the time taken by the target function. Note that *gettimeofday()* has a slightly higher overhead w.r.t. the function *esp_timer_get_time()* thus it is not the best function to measure time.

To measure the RAM occupied at a certain instant in time I used the FreeRTOS's function *heap_caps_get_free_size(MALLOC_CAP_8BIT)* which returns the free number of bytes in the heap memory. By doing the difference before and after a target function you get how many bytes that target function allocates in the heap memory.

I carried out measurements about the time taken by the two main functions added to the API: *zp_prepare_to_sleep()* and *zp_wake_up()*. I measured the time took from these functions to perform their operations and the *zp_prepare_to_sleep()* takes a minimum of 883μs and a maximum of 1225μs whereas the *zp_wake_up()* takes a minimum of 16641μs and a maximum of 16732μs. The RAM occupied is almost constant for every tested example. Additionally, the difference between the two main functions is of a factor of x10000 due to the fact that the *zp_wake_up()* has many system calls to *malloc()*. These baseline measurements are reported in the table 4.1. It must be taken into account that the baseline measurements have been conducted on session's lists with just 1 element.

Varying the load of the session by adding 6 topics first then 10 topics, on each example, for what regards the time differences of the function *zp_prepare_to_sleep()*, I

noticed the example z_get.c being the fastest one with the minor time shift w.r.t. the baseline measurements carried out with 1 topic only. The second fastest example is z_pub.c w.r.t. time differences in the *zp_prepare_to_sleep()*. The time differences for the function *zp_prepare_to_sleep()* are reported in the table 4.2. I did not notice great differences w.r.t. the time differences in the *zp_wake_up()* apart from the example z_pub.c taking 800μs more for the example with 10 topics w.r.t. the z_pub.c for the example with just 1 topic or the z_sub.c taking 50μs more with 6 topics then 97μs more with 10 topics w.r.t. the baseline measurements. The time differences for the function *zp_wake_up()* are reported in the table 4.3.

About the **experimental evaluations** of the implementation, I made calculus on what was the average power consumed by a programme on the ESP32 implementing an active-sleeping pattern (about 6 seconds active, 3 seconds of deep-sleeping) and I noticed that by performing a 3 seconds of deep-sleeping I gain about **2 weeks more** as battery duration on a battery with 250mAh of charge w.r.t. an active-only program running on the ESP32. This is an important result describing the real importance of the support to deep-sleeping for a middleware like Zenoh-Pico.

Moreover, my implementation for the support to deep-sleeping is only about **5.22% greater** than the Zenoh-Pico without the support to deep-sleeping which is a very good result also considering that I did not use a third-party library for the serde implementation.

Open problems

This analysis has a limit concerning the time taken by the *zp_wake_up()* as I was not able to **accurately** take the results with the given method which used the function *gettimeofday()* at the beginning and at the end of the function to take the time needed by a function to perform its task. I expected different results by using 6 topics then 10 topics because the *zp_wake_up* has many *malloc* system calls which are time consuming and by incrementing the number of the topics I increment the length of the inner list in the session struct thus I increment the *malloc*s needed to restore the list in the heap memory. This is an open problem and needs further analysis.

I had not been able to profile the two API functions with the **performance counters** via the performance monitor technique. The problem arose because for a function to be analyzed with performance counters it must be executed. In my case the functions to be analyzed need an active transport and an active Zenoh session thus executing such functions for performance monitoring results in runtime errors with Core 0 panics.

I had a problem with the usage of my support to deep-sleeping implementation for the example z_queryable.c which **crashes** on an InstructionFetchError after a couple of DEEPSLEEP_RESET by the CPU. Also this problem needs further investigation.

Also, analyzing the packets sent with the tool Wireshark I noticed that the **key expression is not compacted to a number** when sent on the wireless as happens with the normal Zenoh protocol. This problem needs further investigation. It might be because the user has to declare at each DEEPSLEEP_RESET the publisher/subscriber as discussed in the next paragraph.

One issue regarding the user experience with the two APIs provided to support the deep-sleeping is that the user has to declare the publisher/subscriber not only once but each time it is needed. I should implement another mechanism to save the

proper struct created by the declaration on the RTC Slow Memory then retrieve the struct and place it in the correct space when needed.

The serde functions to serde user-provided structs have not been tested thoroughly.

Future extensions

With the current implementation of the support to the deep-sleeping mode on Zenoh-Pico for the board ESP32 I have only taken into account the support for **UDP unicast communications** for simplicity's sake because a first prototype about only one type of transport would have been faster and easier to implement. A future extension would be to support the deep-sleeping mode for the UDP multicast transport type. Then, we could add the support to the TCP communication transport even though it will be tough as TCP has to keep some timers client-side and the client will go to sleep so we must leverage what the platform has to offer to the programmer to retain the needed information about timers while the board is asleep.

Another important implementation is to make and create the infrastructure in the code for the support to deep-sleeping to be platform-agnostic. At the moment, the support to deep-sleeping mode is tightly coupled with the board ESP32 as the implementation targets the board **az-delivery-devkit-v4 ESP32**. Also, the current implementation makes heavy use and made some modifications to portions of code that relate to the board ESP32-only. This is because zenoh-pico is structured in a way that PlatformIO sets an environment variable containing the current framework and board used (Zephyr, Arduino, ESP-IDF) and based on that value, at compilation time, the C pre-processor compiles only the needed portions of code. Doing so the zenoh-pico middleware acquires the platform-agnostic property. This heavy modification to the code to make the support to deep-sleeping mode is intrinsic to the property we want to acquire i.e. the deep-sleeping mode, since every board has its peculiarities to support the deep-sleeping mode and to support memory that retains information while the board is sleeping (RTC Slow/Fast Memory in the case of the ESP32 board).

We could conduct another experiment where we use either protobuf or the Binn library to serialize and deserialize the components of the session and see among the three paths which one is the best and adopt the best way to accomplish the task in terms of time efficiency, space efficiency, usability and maintainability.

Wrapping things up

To **wrap things up**, the support to deep-sleeping mode is of paramount importance in the embedded systems world if the device must be fed with energy from an external battery with limited capacity (I made an example with the battery capacity of 250mAh). UDP is the favorite protocol at the transport layer and supporting both UDP unicast and UDP multicast would be beneficial for the end users using zenoh-pico in need of a support to the deep-sleeping mode because their system is powered with a battery limited in its capacity.

Bibliography

Afanasyev, Alex (2018). "A Brief Introduction to Named Data Networking". In.
Bellavista, Paolo (2023). "IoT". slides 1 to 92.
Benini, Luca (2023). "Trends Architecture".
Conti, Francesco (2023). "MCU I/O Interfaces MCU Interconnects InterruptTimers".
Inamdar, Amey (2020). "ESP32 Programmers' Memory Model". https://blog.espressif.com/esp32-programmers-memory-model-259444d89387.
Kulkarni, Sandeep (2014). "Logical Physical Clocks and Consistent Snapshots in Globally Distributed Databases". https://cse.buffalo.edu/tech-reports/2014-04.pdf.
ZenohTeam-Blog (2024). "Zenoh Blog". https://zenoh.io/blog/2024-01-31-zenoh-flow-getting-started/.