System on Chip Architecture
Energy-efficient MQTT sensor node
Project Report

Matteo Fragassi, Davide Giuffrida,
Massimiliano Di Todaro, Bálint Bujtor
Department of Control and Computer Science
Politecnico di Torino

CONTENTS

LIST OF FIGURES

System on Chip Architecture
Energy-efficient MQTT sensor node
Project Report

***Abstract*—Sensor nodes are the main components of the IoT infrastructure, allowing the extraction of sensitive information from the environment. Sensed quantities can be either processed locally, implementing edge computation, or sent to external devices that are part of a "cloud". The following report showcases our take on the problem of designing a sensor node, together with the network architecture needed to ensure communication with external actors. In particular, we worked on an multi-processor MCU-MPU environment to devise a node capable of filtering out-of-window data locally and sending them to external clients, allowing for a customization of the critical window and reducing power consumption through DVFS based on power state transitions, both for MCU and MPU. Our work aims to provide a scalable and highly customizable model for application-independent sensors, playing on both the edge and the cloud computing paradigms through a combination of local processing and communication with remote devices. The product lacks all the fine-grained optimizations that would make it fit for immediate deployment, but its structure and protocols make it extensible and configurable for every kind of specific task or real-time constraint.**

I. Introduction

The IoT field has seen an extraordinary increase in investments in the last decade, following the increasing pervasiveness of technology in our lives. This trend is forecast to be pushed even further in the next years [1].

With this perspective, we developed a sensor node that utilizes the multi-processor architecture of the *STM32MP157F-DK2* board to harvest data from the environment, filter them according to some criteria and transmit them over the network via the MQTT protocol. In terms of hardware, the board is composed of two processors: the powerful Cortex-A7 (MPU) and the low-power Cortex-M4 (MCU). The MCU is an interface between the MPU and the Analog-to-Digital Converter (ADC), while the MPU works as an interface between the data-gathering system (i.e. the CM4 and its subsystem) and the user.

The system is also designed with reduced power consumption in mind. The processors can independently assume a low-power configuration depending on the activity on the node.

In the next sections, the background required to understand the discussed topics will be introduced (II), followed by the presentation of the proposed solution in all its parts (III) and the possible applications of the sensor node (IV). Finally, a series of recommendations is reported in section V.

II. Background

An ADC, the Inter-Processor Communication (IPC), the MQTT protocol and the low-power modes are the building blocks of our sensor node. This section aims to briefly introduce the concepts needed to understand each of them before fully explaining the proposed system architecture.

A. ADC

The ADC block has two successive approximation ADCs that have a configurable resolution of 8, 10, 12, 14, and 16 bits. Each of the two ADCs has up to 20 multiplexed channels, including 6 internal channels that are only connected to ADC2. The ADC input voltage range is between 0 and 3.6 Volts.

A conversion can be organized into two groups: *regular* and *injected*. A group is a sequence of conversions that can be done on any channel, either regular or injected and in any order. In particular, regular groups have up to 16 conversions, while injected groups can only have a maximum of 4 conversions. When a channel is configured as injected, it has a higher priority than regular channels. It means that it can suspend any ongoing regular channel conversion when the injected channel/group is triggered. If a single channel is converted, the operation is performed in *single mode*, while, if multiple channels are converted, the sequencer needs to be enabled and the conversion is performed in *scan mode*.

Both hardware and software can issue the conversion start to the ADC. In the former case, it can be done via timer events or GPIO input events. Except for timer events, the trigger can signal a regular or injected conversion, the latter having a higher priority.

After the ADC is started, the conversion can be performed in different modes:

- **Single**: The ADC performs once the sequence of conversions.
- **Continuous**: The ADC repeatedly converts the selected sequence. Only for regular channels.
- **Discontinuous**: The ADC converts up to 8 regular channels or 1 injected channel in the conversions sequence per conversion trigger. This means that if the sequence of conversions is longer than the number of conversions performed at each trigger, the conversion needs to start multiple times to complete the group.

The processor or the DMA can read the digital result of the conversion in a left or right-aligned 32-bit data register. Moreover, the ADC can use a synchronous or asynchronous clock source.

The ADC has many features such as self-calibration to increase the conversion accuracy, oversampling to reduce the input channel's noise and a low-power mode to decrease power consumption. However, the most exploited feature in our system is the *Analog Watchdog* (AWD). This functionality allows the ADC to detect if the result is outside a user-defined window, defined by a high and a low threshold value. If the input value is out-of-window, the AWD module creates an interrupt that informs the MCU that there is new data to transmit. This solution allows a fast and low-complexity check that would otherwise be performed in the MCU firmware.

For more information on the ADCs refer to section 29 of the reference manual [5].

B. Inter-Processor Communication

Inter-processor communication (IPC) refers to the communication between the MCU and the MPU. Because they are separated modules, each with its subsystem, a certain software stack is required on top of the provided hardware to ensure easily accessible and reliable communication. However, despite its complexity, the IPC implementation can differ depending on the requirements of the message exchanges [7]:

- **Direct Buffer Exchange Mode** (DBEM): for control messages and low data rate exchanges.
- **Indirect Buffer Exchange Mode** (IBEM): for high rate transfers and large data buffers.

For our purposes, Direct BEM is the best choice because it fits all the requirements while it allows to have a simpler RPMsg interface and a reduced memory occupation. This is possible because the RPMsg buffers contain effective data, so no other memory allocation is needed.

As mentioned, the IPC is supported by some hardware modules allowing hardware interaction between different subsystems. The *Inter-Processor Communication Controller* (IPCC) is the unit that manages the message exchange [8]. The processors can communicate using 6 channels in simplex, half-duplex or full-duplex mode and each processor has two interrupts to know if the channels are occupied (RXO) or free (TXF). However, these signals are shared by all channels for each processor. The predefined configuration on STM32MP15x boards is a full-duplex communication on channel 1 and channel 2. The MCU uses channel 1 to indicate that a new message is available while the MPU uses it to indicate that the message has been treated. Channel 2 handles the opposite messages. More details about this peripheral's implementation can be found inside section 12 of the RM [5].

Another important hardware module is the *shared memory* where the exchanged data can be stored by the sender and retrieved by the receiver [9]. The overall memory shared by the MCU and the MPU has different memory modules: MCU SRAM1/2/3/4 (384kB) and RETRAM (64kB). The region reserved for the IPC buffers in the default STM implementation is the MCU SRAM3 (64kB). The proposed application has 128KB of reserved memory inside the MCU SRAM. The memory mapping is the same as the STM example that was used as the starting code to build the MCU firmware [6] for the sensor node.

When it comes to the software, the IPC is mainly supported by the *Remote Processor Messaging* (RPMsg) protocol, a *Virtio* based protocol that offers a high-level communication interface, and by the *Mailbox* protocol, that drives the IPCC. The implementation of these services depends on the considered subsystem and on the supported environment [10]. On the MCU side, the Mailbox is managed by the HAL_IPCC driver and the RPMsg service is implemented by the OpenAMP library. Additionally, a higher-level interface that mimics a UART communication is provided. This *virtual driver* is called Virtual UART and relies on RPMsg. On the MPU side, the Mailbox controller is implemented by the stm32_ipcc driver [12] and the RPMsg protocol is handled through the Linux RPMsg framework [11]. Another important protocol is the Remoteproc framework [13]. Unlike the RPMsg and Mailbox services, it is not used to exchange messages but to enable the IPC on the Linux side depending on the information in the firmware resource table [14]. In general, the protocol allows different platforms/architectures to remotely control (power on, load firmware, power off), monitor and debug remote coprocessors.

C. Low-power states

The main parameters which define the power consumption are *clock frequency* and *voltage level*, so by tuning their values for each power domain (e.g. processor cores, analog modules and clock tree) it is possible to control the energy consumed over time. Devices achieve the targeted energy consumption by operating in the appropriate power states ensuring the correct system functionality. Power states are classified according to their "depth". A *deeper state* offers more energy savings but usually requires more time to move to another power state. Another side effect is that to achieve a certain energy consumption, deeper states power off entire domains so that it is impossible to move to another state with some *wake-up sources*. As mentioned, to achieve maximum energy savings, it is important to operate at the deepest power state that does not interfere with the mission of the system.

The STM32MP157F-DK2 board offers different operating modes that allow to control of the clock and power distribution. These modes are classified not only by their "depth", but also depending on the affected part of the system:

- **Subsystem operating modes**
  MPU and MCU are separate subsystems that can communicate and share resources. Because of this structure, they can work in certain subsystem operating modes almost independently from each other. The MPU has 4 available states (CRun, CSleep, CStop and CStandby) while the MCU can use only 3 (no CStandby).
- **System operating modes**
  These operating modes can be considered "global power states" since they involve the entire system. The adoption of one of the 5 system modes (Run, Stop, LP-Stop, LPLV-Stop and Standby) depends directly on the operating mode of MPU and MCU combined.

A subsystem moves into an operating mode when the necessary conditions are satisfied and the entry procedure is

triggered. The mechanism is the same if the subsystem tries to move from one power mode to another. In this case, the correct exit triggers, among the entire pool of available wake-up sources, depending on how the mode was entered. For example, if a subsystem is moved into CStop mode through a WFI, it needs a wake-up interrupt to return to CRun. These concepts roughly apply to the management of system operating modes. A substantial difference is that these global states do not need explicit entry/exit procedures. The system moves in and out of an operating mode considering the states of both MCU and MPU.

According to the information provided so far, to adopt a low-power strategy it is sufficient to act appropriately on the subsystems. On the MCU, the coding of the entry/exit procedures is eased by the HAL_PWR driver and other low-level primitives. On the MPU, the low-power management is performed through the Linux suspend framework, which is based on the Generic Power Domain (GenPD) framework and the Power State Coordination Interface (PSCI) standard. When a suspension is requested, the entered system low-power state depends on the available wake-up sources. In fact, the secure OS (OP-TEE) is informed whenever a peripheral is set as a wake-up source so that its power domain is not switched off. This means that, upon a kernel request to suspend the platform, the OS knows which power mode can be reached according to the power domains that must remain active [15] [16].

Additional information can be found in section 9 of the RM [5] and on the application note AN5109 [17].

D. MQTT

The MQTT protocol is well-suited to meet the application requirements in terms of performance and power consumption [2] [3]. Some key characteristics are highlighted here:

1) **Publish/Subscribe architecture**
   Instead of a direct message exchange between a sender and a receiver, as in the traditional client-server model, MQTT decouples the communication by using an intermediate unit, called *broker*. Clients publish messages on specific channels, called *topics*, and other clients subscribe to those topics to receive the published messages.
2) **Topics**
   Messages are categorized and identified by topics. A client can act as a *publisher* of a topic, by publishing messages on that topic, or as a *subscriber*, by subscribing to receive the messages published on that topic, or both.
3) **Broker**
   An MQTT broker acts as intermediary that manages the message distribution. It receives messages from publishers and forwards them to the appropriate subscribers. This module is one of the central points of the publish/subscribe architecture since it simplifies the communication model and ensures message delivery.
4) **Quality of Service (QoS)**
   The QoS is the message delivery assurance level. MQTT provides three choices: QoS 0 (at most one delivery), QoS 1 (at least one delivery), QoS 2 (exactly one delivery). The QoS is associated with a publish/subscribe operation, so different operations can have different QoS even though they are performed on the same topic. However, it is important to underline that the QoS level for a certain message delivery caps to the publish level. This means that even if a subscriber has a QoS of 2 for a topic when a client publishes on that topic with a QoS of 1 or 0, the broker-subscriber communication will have a QoS of 1 or 0 for that message delivery [4].

Even though MQTT has been proven to be suited for low-power applications, the performance and power consumption of the system also depend on the placement of each module within the network architecture.

III. Proposed Solutions

The sensor node has been developed without a precise deployment scenario in mind, so it lacks all the fine-grained optimizations typical of a complete product. It is more of a template that can be easily adapted to most of the field-specific applications that require this kind of system.

In the following sections, the node has been broken down into more pieces to allow a better understanding of each part, even though the division is not so clear when the code is considered.

A. Data acquisition on the Cortex-M4

The analog-to-digital conversion is performed with ADC2. In the proposed implementation, the ADC is not fed by a real sensor to ease the system testing. Instead, channel 1 of DAC1 is internally connected to channel 16 of ADC2 and it is configured to generate one of five different analog values, that span the entire conversion range of the ADC. The output is changed when the ”user button 1” is pressed. This action generates an interrupt associated with EXTI14 that triggers the update of DAC1 in the main loop. In particular, ADC2 is configured to have a 12-bit resolution so the range of digital values it can produce is $(0, 4095)$ while the maximum accepted voltage is 2.9V, the same value of the positive internal reference V_{REF+}.

The provided system is designed to continuously monitor a sensor, so the ADC samples the input signal in continuous mode after the software starts. Considering that the input value of the ADC is quasi-constant, the scenario in which the signal is continuously sampled is the most challenging because the system is heavily stressed. This is the reason that led to the choice of this configuration for the proposed sensor node.

Furthermore, the ADC also performs the data filtering with the AWD feature. In particular, AWD1 is configured to monitor only channel 16 with the same resolution as the ADC so the high/low threshold has to respect the imposed value limit. Each time a value is converted, the AWD module checks if it is inside the programmed window, so if $thr_{low} < data < thr_{high}$, and in case the condition is not met an interrupt is fired. Inside the handler, the last converted value is retrieved from the DR

register of the ADC and it is added to the data buffer after being converted into the correct unit of measure; mV in our case. In case the buffer limit is reached, the ADC is stopped the next time the callback is invoked. Even though this causes new out-of-window data to be lost, it avoids flooding the MCU with interrupt requests.

After gathering enough data, the MCU tries to send them to the MPU. The communication is started if some conditions are satisfied:

- The MPU did not request any operation (i.e. suspension, threshold update) and acknowledged III-B all the data previously transmitted by the MCU.
- The number of out-of-window values in the buffer has to be higher than the minimum number of data that can be delivered in a single transmission.

The ADC is stopped during the transmission to prevent the modification of the buffer.

B. CM4-CA7 communication

The Inter-Processor Communication explanation in section II-B is much more detailed than the actual knowledge necessary to implement functional code starting from the `OpenAMP_TTY_echo_wakeup` example [6]. In fact, if the configuration of the different frameworks does not require any change, the communication can be completely managed with high-level interfaces. On the MPU, the RPMsg TTY driver [11] is used to expose a standard TTY interface (`/dev/ttyRPMsg<x>`) that simulates a serial link between the two processors. On the MCU, the interfaces are the OpenAMP primitives and the Virtual UART driver, both properly configured. For example, the RPMsg client name is set to `rpmsg-tty` inside `virt_uart.c` to match the name of the RPMsg service offered by the RPMsg TTY driver.

The communication is finally set up during the startup of the kernel by the Systemd service `comm_setup.service`. This service has `setup.service` and `/dev/ttyRPMsg0` as dependencies because it executes `comm_setup.sh`. The script waits that the MCU is online after `setup.sh` flashes it and then sends it a message through the RPMsg TTY interface to let the MCU know its RPMsg endpoint address. After the configuration of some peripherals, the MCU waits for a transmission coming from the master processor. Normally, the address could be delivered with any message, however, the MCU expects to receive the initial values to define the AWD window. In case the message is not parsed correctly, the firmware execution gets stuck in the `Error_Handler()` and the whole setup process needs to be repeated starting from the firmware flash.

After the setup phase, the application can actually start. The MCU, continuously polls the Virtual UART channel with the `OPENAMP_check_for_message()` function to check if there are new messages. When this happens, the execution of a previously registered Virtual UART callback is triggered. The callback verifies the dimension of the received message and copies at most 512 bytes of data from the RPMsg buffer into a buffer defined in the MCU firmware. This avoids to lose the data because a successive transmission has overwritten the RPMsg buffer. For messages larger than 512 bytes, more data transfers are necessary. Additional information can be found in section A-B.

When a message is retrieved, it is parsed and proper actions are taken depending on the message type:

- **THR**
  The message contains the new thresholds of the monitored AWD channel. The ADC is stopped to allow the configuration of the AWD but before being restarted, the MCU sends an acknowledge to the MPU to report the successful update.
- **DATA_ACK**
  The acknowledge normally notifies the MCU that the MPU has correctly processed the transmitted out-of-window data. However, it is also used to enable again the transmission of data after a suspension or threshold update operation.
- **WAIT_THR**
  The message signals that a threshold update is required. In case there is no unprocessed data on the channel, the MCU disables the transmission of new data and sends a message to inform the MPU that it is ready for the update. Otherwise, it waits for the MPU to free the channel and send the new thresholds.
- **WAIT_SUSP**
  The message warns that the MPU is entering a low-power mode. If there is no unprocessed data on the channel, the MCU waits for a certain time to avoid breaking the operating mode entry procedure and then automatically acknowledges the elaboration of the data, otherwise, it does nothing. In the latter scenario, the MPU does not go into suspension because the MCU inactivity interval is reset. This message is currently unused because the low-power mode of the MPU is disabled. Refer to section III-D for more information.

As mentioned in section III-A, the MCU autonomously start a communication only when it needs to transmit out-of-window data to the MPU.

The part of the application that runs on the MPU handles the communication with the MCU and with remote users over the network. In particular, the management of the interactions with the coprocessor depends on the performed operation:

- **Out-of-window data**
  The `/dev/ttyRPMSG0` interface is checked at each iteration of the main loop. When a new set of data is available, the MPU converts them in a convenient format and sends them to the users.
- **Threshold update procedure**
  After the threshold is received, the MPU alerts the MCU with a WAIT4THR message (WAIT_THR) and then reads the channel. If the buffer contains the RDY4OP message, the update procedure continues normally, otherwise the channel is occupied by unprocessed out-of-window data. In this scenario the data are sent to the user and no RDY4TR message (DATA_ACK) is transmitted to the MCU until the update procedure is concluded. The next step is to communicate the threshold to the

MCU and wait for an acknowledge (THR_SET). After the update is flagged as successful, the MPU sends an acknowledge to the user client. A further acknowledge is sent to the MCU to allow it to start new data transmissions.

C. Interactions on the network

The network communication involves different actors, that cover an important role in the application environment. However, the communication protocol and the network architecture are highly influenced by the adoption of low-power features on the sensor node so it is suggested to check section A-C and A-D to explore possible alternatives to the proposed solution.

Independently from the adopted protocol, the nodes that are provided to build one of the possible network infrastructures are the following ones:

- **MQTT broker**
  A standard module whose functions have already been described in section II-D. It can be placed on any node connected to the network and every implementation compliant to the MQTT protocol standard can be deployed. We provide a configuration file (`mosquitto.conf`) for the Mosquitto broker [24].
- **Board client**
  The client that runs on the MPU of the sensor node. Other than handling the Inter-Processor Communication, it also manages the interactions on the network. The script used for the deployment of the application is `ipcc-interface_noSusp.service.py`. Another version of the client (`ipcc-interface_noSusp.py`) is available for development purposes.
- **User client/s**
  The client used by the final users to interact with the sensor node. in particular, it can request threshold updates to the boards on the network and it can receive out-of-window data from the same boards. The client can be executed on every device that can access the network and use the MQTT protocol. The code for the current version of the application is implemented in `client_noSusp.py`.
- **Proxy client**
  The proxy client is needed when the MPU subsystem can be moved into a low-power mode and the Ethernet WoL feature is configured to receive magic packets. In the current version of the application it is not used because the low-power feature developed for the MPU conflicts with the network communication. Refer to section A-D for additional details.

In the current version of the application, the nodes interact using only MQTT, so a proper organization of the topics is fundamental to correctly set up a communication protocol:

- BOARD_TOPIC_STATUS
  The board client updates the user client on its status using this topic.
- BOARD_TOPIC_ERROR
  The board client alerts the user client that there has been a fatal error using this topic.
- THR_TOPIC_UPDATE
  Topic where the user client asks the board client to perform a threshold update by transmitting the new threshold.
- THR_TOPIC_ACK
  Topic used by the board client to inform the user client that the required threshold update has been successfully completed.
- DATA_TOPIC
  The board client sends the out-of-window data retrieved from the MCU to the user client through this topic.

The network communication represents a different part of the process for the two main operations implemented by the application. In fact, while for the out-of-window data the board-user interaction is the last step of the data exchange, the threshold update operation starts and finishes with an MQTT message.

For what concerns the data exchange, after the data are retrieved from the MCU, as explained in section II-B, the MPU sends them to the user client by publishing a message on DATA_TOPIC with DATA_QOS as QoS. For this topic, the appropriate QoS value depends on the delivered data. Once the data are received by the user client, they are logged in a dedicated file.

Moving on to the threshold update procedure, this process starts when a user client sends a message containing the new threshold and the list of MAC addresses of the boards to update on the topic THR_TOPIC_UPDATE. At this point, the board client executes the process as described in section II-B, while the user client enters in a wait-state. In fact, it expects that all the targeted boards acknowledge the success of the update on the topic THR_TOPIC_ACK by sending their MAC address. If not all the boards publish on the topic within a certain period of time, the process is marked as ABORTED, otherwise it is labeled as FINISHED. In case of failure, the update should be issued again for the boards that caused the timeout expiration. Besides, the error log should be checked to assess if the boards have reported any problem. THR_TOPIC_UPDATE and THR_TOPIC_ACK require a QoS of 2 because sending repeated messages may cause a waste of time on the first topic and a critical update error on the second topic. In particular, the error happens when a late acknowledge incorrectly flags that a board has been updated, even though the acknowledge does not refer to the ongoing update procedure.

The MQTT client is implemented with Paho MQTT [21] on both the board and the user device. The library offers many methods to manage the communication protocol synchronously or asynchronously. In our case, the latter option has been adopted. After configuring the client to use version 5 of the MQTT protocol, the connection to the broker is established and the `loop_start()` method is called. This function launches a new background thread that handles all the network operations, like the execution of standard callbacks when specific packets are received. Among the provided callbacks, the most important for the application is `on_message()`, the function executed when a message is delivered through

one of the topics the client is subscribed to. Inside `client_noSusp.py`, this method is used to log the incoming messages and record the boards that transmitted an acknowledge while a threshold update procedure is active. On the board client (`ipcc-interface_noSusp.service.py` or `ipcc-interface_noSusp.py`) the callback mainly handles the threshold reception.

The current version of the application supports multiple boards in different local networks but only a single user client. Even though this feature is not available, some code to handle the multi-user client scenario have already been developed. For example, the board client implements a protection mechanism against concurrent threshold update requests. Another limitation of the current communication protocol is that the user client needs to know the list of online boards before it is started.

D. Power-states

The STM32MP157F-DK2 offers many solutions for low-power applications, so the system was developed to exploit some of them, without sacrificing too much in terms of performance and functionality. In the proposed implementation, even though MCU and MPU both need to be active to perform most of the operations supported by the sensor node, each subsystem adopts a unique power management strategy that depends on allocated resources and subsystem-specific tasks.

The MCU can enter CSleep when it does not communicate with the MPU. In this case, there is no ongoing thresholds update procedure, all transmitted out-of-window data have been acknowledged and the data buffer has too few elements to start a new transmission. The subsystem moves to the low-power operating mode by calling a Wait For Interrupt (`WFI`) with the `HAL_PWR_EnterSLEEPMode()` function. Since in this state all the interrupts enabled in the NVIC can wake up the MCU, the Systick interrupt must be disabled before leaving CRun and enabled after a proper wake-up. The subsystem should be reactivated only when the MPU requests to update (IPCC_RX0 interrupt) the thresholds and the AWD detects out-of-window data (ADC2 interrupt). In CSleep, the MCU clock is stopped but the clock of the subsystem peripherals may still be active.

When it comes to the MPU, as already explained in section II-C, the choice of the operating mode depends on the configuration of specific registers and on the selected wake-up sources. In our case, when the MPU is suspended, the Ethernet peripheral (ETH) can request a threshold update and the IPCC can request the transmission of new data. Since these interrupts are both associated with EXTI events (respectively event 70 and event 61), they can wake up the MPU after it enters CStop with a `WFI`. In the ST example, the Mailbox (IPCC) and the Ethernet peripheral do not have any wake-up capability by default. These features are enabled at the startup of the kernel by two Systemd services:

- `setup.service`
  The service executes a script (`setup.sh`) that activates the Mailbox wake-up event.
- `wol@end0.service`
  The service enables Wake-on-Lan (WoL) in unicast mode. This means that the Ethernet module wakes-up the system when it receives a network packet addressed to the board. Refer to section A-D for more WoL solutions.

The low-power mode is activated when the MPU does not receive out-of-window data from the MCU for a certain time and when it has no pending MQTT message to send to the user client/s. The MPU subsystem suspension process follows these steps:

1) The MCU is notified that the MPU is starting the suspension process
2) The MPU waits for the response of the MCU:
   a) The MCU has no request for the MPU
      The suspension can proceed normally
   b) The MCU can transmit new data to the MPU
      The data exchange is executed, the MCU inactivity timeout is reset and the suspension process is stopped.
3) The MPU is suspended

The subsystem suspension is requested with a `systemctl suspend`. After a MCU/user request, the MPU is woken up but, without further actions, the application does not know which operation it needs to handle. For this reason, after resuming the execution, the application retrieves the last ETH wake-up event from `/sys/kernel/debug/wakeup_sources` and compares it with the previously recorded ETH event that caused the subsystem to exit from suspension. A different event time means that the MPU was woken up through the ETH peripheral, otherwise the wake-up event was triggered by the IPCC.

The subsystems power management does not affect the system, which always remains in Run mode. Pushing the energy savings even further would compromise the functionality of the sensor node. Please refer to section A-D for a more detailed explanation of the power configuration.

The MPU power management as described above is implemented in `ipcc-interface.py`. Unfortunately, an unsolved issue caused to remove from the system the possibility to suspend the MPU. In the current state, the only power-optimized subsystem is the MCU and the correct MPU client is `ipcc-interface_noSusp.service.py`. Section A-D addresses the error more in detail.

IV. Possible applications

In a world where every aspect of human life is monitored, sensors are fundamental and so it is to develop data acquisition chains like the proposed one.

The most prominent sector that could benefit from our sensor node is the IoT industry, more precisely edge-computing applications. The dual processor architecture would prove beneficial because the weaker but low-power M4 core could gather data (e.g. temperature, humidity) and notify the powerful A7 core only when the values of a monitored quantity required to perform some operations (e.g. irrigate the monitored plantation).

The automotive field could be another application area. In fact, a modern vehicle has over 200 ECUs that operate and communicate constantly. In such scenarios, it is critical to have an effective communication so that, even if the entire sensor node cannot be directly used, some takeaways of the inter-process communication development could be applied.

V. Conclusions and Recommendations

The generic sensor node implemented in this project allows us to evaluate the usability and effectiveness of the STM32MP157F-DK2 multi-processor architecture in terms of performance, power efficiency and flexibility to use it in different industries and applications.

The less powerful MCU has been used to sample an analog signal and filter the values with the AWD feature of the ADC, according to a user-programmable window. Moreover, the MCU is also in charge of notifying the more powerful MPU after enough out-of-window data have been gathered. This operation, together with the AWD threshold retrieval, has been enabled by the IPC, which connects the otherwise separate subsystems of MCU and MPU. The powerful MPU has been equipped with an MQTT client in order to transmit out-of-window data to remote users and to listen for incoming thresholds. In addition, the different power states offered by the board have been exploited to achieve optimal power consumption. This may lead to extend the system's life, a fundamental quality for a real product.

It is possible to state that the system is highly customizable and can be further tailored for a specific application to achieve even higher performance and better power efficiency. For example, the DAC can be substituted as a signal generator with any analog sensor. The energy savings are allowed by features like the AWD, which unburdens the MCU from processing data that are not required by the user. The energy-efficiency is further improved by the fact that the processor only communicates when enough data has been harvested. The selected inter-processor architecture enables reliable communication between the two nodes by using handshake and resource-locking techniques. Using different message types allows us to add new features and use cases to the IPC based on the application's needs. The publish/subscribe architecture of the MQTT protocol provides a reliable, extendable and easy-to-use solution to communicate with remote actors while having a power consumption comparable to other network protocols. The MQTT topics enable the tailoring of the information distribution based on the interest of each actor. This also allows easy addition of new features if needed.

Even though the proposed solution can already serve as a basis for similar applications, it can be further improved. In particular, the MPU suspension error detailed in section A-D should be solved and the MCU error handling should be integrated into the existing communication protocol before deploying the node in a real-world scenario.

Appendix A
Insights on the sensor node

As explained in section III, the sensor node can be configured to meet application-dependent requirements different from the ones proposed in the presented system. To ease this process, some suggestions are provided below, with additional information on the application and its environment. The system division adopted in section III is kept for clarity. In addition, inside each section, the paragraphs are grouped (Error, Modification, Information) depending on their purpose.

A. Data acquisition on the Cortex-M4

`[ERROR]`

- ADC IRQ Handler

  ADC interrupts do not work in the original STM example [6]. The error can be solved by changing the interrupt handler name in `main.h` to `ADC2_IRQHandler`, if ADC2 is used.

`[MODIFICATION]`

- ADC configuration

  In case **single mode conversion** is needed, the original project *Open_AMP_TTY_echo_wakeup* [6] is a good start because it is configured to convert one channel when the ADC is triggered by TIM2 (timer). Moreover, the **DMA** is used in *circular mode* to move the data from the ADC to a buffer.

  The DAC1 output channel **internal connection** to the ADC is **exclusive to ADC2** (page 1497 of the RM [5]) so if other quantities that can be converted only by ADC2 (i.e. V_{DDCORE}, V_{REFINT}, etc) need to be monitored, ADC2 should be used in *scan mode* instead of single mode.

  When **more sensors** are attached to the sensor node, each of them uses at least an ADC channel. In this scenario, **Analog Watchdog** 1 cannot be used to monitor all these signals, unless they have to be in the same range of values. In fact, AWD1 can monitor one channel or all of them but with a single window. In case the range of values of each signal is different, the choice is between AWD2 and AWD3, since they can monitor several channels with different windows. However, their resolution is limited to 8 bits.

- **Minimum number of elements per transmission**

  This value depends on many application-specific factors, like severity and power consumption. If the presence of out-of-window data can cause serious damage the user should be immediately informed of the event, so the data have to be transmitted as soon as they are available. However, if the data transmission can be delayed, the MPU is less active, potentially causing a significant energy save over time, especially when low-power states are exploited.

`[INFORMATION]`

-

B. CM4-CA7 communication

`[ERROR]`

•

`[MODIFICATION]`

- RPMsg customization

  The RPMsg *buffer size* is hard-coded to 512 bytes [7] and the **number of buffers** is 16 per *vring* in the original STM project example [6]. The number of vrings is fixed to 2, one to receive and one to transmit. These implementation details can be observed on the Linux side by unflattening the device tree under `/boot/stm32mp157f-dk2-m4-examples.dtb`:

```
vdev0vring0@10040000 {
    compatible = "shared-dma-pool";
    reg = <0x10040000 0x1000>;
    no-map;
    phandle = <0x87>;
};

vdev0vring1@10041000 {
    compatible = "shared-dma-pool";
    reg = <0x10041000 0x1000>;
    no-map;
    phandle = <0x88>;
};

vdev0buffer@10042000 {
    compatible = "shared-dma-pool";
    reg = <0x10042000 0x4000>;
    no-map;
    phandle = <0x89>;
};
```

  As expected the buffers memory occupation is 16kB, i.e. $16 * 512$ bytes.
  Among the mentioned parameters, the only one that can be changed is the number of buffers. This value can be found inside the file `openamp_conf.h` [14].

`[INFORMATION]`

- **CM4 log in Linux**

  The Remoteproc framework [13] allows to dump the MCU firmware traces on the Linux side. In the original STM example [6] this feature is already available, however, in case it needs to be activated, it is sufficient to define `__LOG_TRACE_IO_` as a preprocessor directive.

- MCU error handling

  MCU errors that compromise the coherency of the application are handled by calling the `Error_Handler()`. However, in the current state, the error is only reported in the MCU log before entering an infinite loop. A possible improvement could be to integrate the error handling in the CM4-CA7 communication protocol by adding an error message that forces the MPU to perform a non-clean exit, thus restarting the entire application.

C. Interactions on the network

`[ERROR]`

•

`[MODIFICATION]`

- **Alternative network architecture**

  In the current version of the application, the network communication relies completely on a **publish/subscribe** approach. However, with some network architectures, some of the nodes could also interact using a classic **client/server** message exchange. In this case, the actors presented in section III-C would play a different role and, consequently, the topics used as part of the MQTT-based communication would be different as well. A possible architecture that mixes the publish/subscribe and the client/server approaches is described in the following paragraph.
  The network nodes that are changed by the new communication protocol are:
  - **Proxy client**
    The board and the user client mainly interact through the proxy rather than directly. In fact, this client coordinates all the network communication during the threshold update procedure.
  - **Board client**
    The client mainly interacts with the proxy present in its local network using a low-level network protocol instead of MQTT. Because of this, the board client may not need to implement an MQTT client, depending on how the out-of-window data are transmitted. The topic is detailed below.
  - **User Client**
    The user client performs the same tasks of the current implementation but it publishes on topics whose only subscribers are the proxy clients.

  The differences in the implementation of the nodes are a consequence of the different network protocol adopted for the main operations of the application.
  The threshold update starts when the user client publishes a message for the proxy client on MQTT_TOPIC_NEWTHR. The payload of the message contains the list of MAC addresses of the boards that have to be updated and the new threshold values. If the message is correctly formatted, the proxy wakes up the target boards, otherwise it publishes an error on the PROXY_TOPIC_ERROR, to notify the user client about the error. At this point, the proxy waits a wake-up acknowledge from the boards and, once this message is received, it delivers the new threshold. After the update has been completed, each board sends an acknowledge that the proxy client publishes on MQTT_TOPIC_NEWTHR to inform the

user client about the update status. This topic should have a QoS of 2 to avoid the same problems addressed for THR_TOPIC_UPDATE and THR_TOPIC_ACK in section III-C.
The out-of-window data exchange can still be implemented relying completely on MQTT. The data could be transmitted to the user client on DATA_TOPIC without the involvement of the proxy, however, the board client would need to implement an MQTT client. In case this was not possible, the board client could just send the data to the proxy and let it forward them on DATA_TOPIC. The topics PROXY_TOPIC_STATUS and PROXY_TOPIC_ERROR are still needed to let the user client know about any useful runtime information on the proxy client. Moreover, the board-proxy interactions inside the local network can be implemented with every protocol that is considered to be appropriate for the task.
The presented architecture is more complex to implement than the one that relies only on MQTT. However, if the application supported many user clients, it would allow to have a cleaner network communication (i.e. organization of the MQTT topics, local interactions with the boards, etc). This aspect would be very important in case the application was further developed, for example by adding new operations.

`[INFORMATION]`

- Avoid **race conditions on variables**

  When the `loop_start()` function is called, it launches a new background thread that manages all the library network operations. The callbacks run on this thread too, so it is important to correctly handle accesses to the variables that are modified by the main and the background thread in order to maintain application-level coherency. The fact that Python is currently capable of running only one thread at a time, does not prevent to incur in race conditions [25] [26]. For this reason, the board client and user client use a lock object provided by the `threading` library.

- **Network architecture with MPU low-power mode**

  The current version of the application does not support any MPU low-power function because of the Ethernet DMA error described in section A-D. Despite this problem, the network nodes that would be necessary to support that feature are provided anyway. However, the nodes implementation and the network architecture depend on the adopted Wake-on-Lan configuration:
  - **Unicast packets**
    In case the WoL was set up in unicast mode, the actors and the network organization (i.e. topics, communication protocol, etc) would be the same presented in section III-C. Moreover, the script used to implement the user client would still be `client_noSusp.py` but the board client would be `ipcc-interface.py` instead of `ipcc-interface_noSusp.py`, as mentioned in section III-D.
  - **Magic packets**
    In case the WoL was set up to be responsive to magic packet activity, the network configuration described in section III-C would require another actor: the proxy client. In this scenario, before sending the new threshold to a board, the user client should ask the proxy client to wake up that board by sending it a magic packet. It means that there should be a proxy client in the local network of each board that is part of the application.
    Furthermore, because the nodes interact using MQTT, the additional complexity of the network communication needs to be supported with new topics:
    - PROXY_TOPIC_STATUS
      The proxy client updates the user client on its status using this topic. The QoS should be set to 1.
    - PROXY_TOPIC_ERROR
      The proxy client alerts the user client that there has been an error using this topic. The QoS should be set to 1.
    - PROXY_TOPIC_WAKEUP
      Topic where the user client informs the proxy client of the boards it needs to wake up. The QoS should be set to 2, as explained in section III-C for the topic THR_TOPIC_UPDATE.

    The proxy client is implemented in `wol_proxy.py` while the board client and the user client are respectively `ipcc-interface.py` and `client.py`. In this case, the user client is different because it performs more steps than the other version during a threshold update operation. In fact, after sending the list of boards to the proxy, the user client waits a certain time before publishing the threshold on THR_TOPIC_UPDATE. This is done to avoid that the message is not received by a targeted board because it was disconnected from the broker.

D. Power-states

`[ERROR]`

- **MPU suspension error**

  As anticipated in section II-C, the MPU cannot be suspended due to an unsolved error. The part of the system involved with this issue is the Ethernet DMA.
  The error can be observed during the DMA initialization of the Ethernet peripheral after the subsystem wakes up from a suspension. It is also necessary that an MQTT connection was established before moving the MPU to a low-power state, even if the connection was not active when the MPU was suspended. The `dmesg` output of a suspension followed by a wake-up sequence is reported below:

```
Disabling non-boot CPUs ...
CPU1 killed.
Enabling non-boot CPUs ...
CPU1 is up
stm32-dwmac 5800a000.ethernet:
Failed to reset the dma
stm32-dwmac 5800a000.ethernet end0:
stmmac\_hw\_setup:
DMA engine initialization failed
stm32-dwmac 5800a000.ethernet end0:
Link is Down
stm32-dwmac 5800a000.ethernet end0:
Link is Up - 1Gbps/Full -
flow control off
```

After the error, the network communication through Ethernet is restored when a watchdog times out. These are part of the corresponding log messages:

```
------------[ cut here ]------------
WARNING: CPU: 0 PID: 0
at net/sched/sch_generic.c:525
dev_watchdog+0x208/0x27c
NETDEV WATCHDOG: end0 (stm32-dwmac):
transmit queue 0 timed out
```

The issue does not depend on how the suspension is started or the MPU is woken up. Launching a `systemctl suspend` on a terminal or from `ipcc-interface.py` leads to the same result. Waking up the subsystem with a magic packet, a unicast packet, the board wake-up button and an IPCC interrupt always triggers the error. The Ethernet DMA initialization also seems to fail independently from the state of the MQTT client. Some tests have been executed in different scenarios, all with the same result:

1) The MQTT client has been disconnected from the broker and the client background operations, managed through `loop_start()` and `loop_stop()`, have been stopped before moving the MPU in low-power mode.
2) The MQTT client is not disconnected from the broker and the background operations (i.e. the thread started by `loop_start()`) have not been stopped before the suspension.

Besides, even if the error can be reproduced, the way it modifies the application behavior is not consistent. This can be demonstrated by the fact that sometimes the network connection remains up and the board is able to perform some network tasks even when the DMA fails. For example, the MQTT client may be able to successfully perform some publish operations and get stuck while trying to publish other messages. A similar scenario occurs when, after waking up, the system executes a threshold update and correctly sends the acknowledge back to the user client. However, if new out-of-window data need to be transmitted, the application proceeds like if it sent them, even though the user client has received nothing.
At the moment, further debug efforts are required to precisely know the source of the issue. However, it is not possible to debug the problem with STM32CubeIDE because the inspection of Ethernet registers, like ETH_DMA, is disabled.
The next steps could be to investigate the compatibility of the MQTT Python library (Paho MQTT [21] [22]) and the ST network library [20] and to analyze the device errata sheet to check any compatible ETH error [19].
In case the issue could not be solved, a possible alternative to the solution adopted in the sensor node is to synchronously manage the network activity by using `loop_read()`, `loop_write()` and `loop_misc()`. A practical example on how to use these functions can be found inside the Paho MQTT repository [22] under `examples/loop_select.py`. A brief but useful description of the library can be found on the README of the Github repository while a more complete documentation is on the dedicated Eclipse page [23].

- Exit from **Stop** and **LP-Stop**

  The exit from Stop and LP-Stop is affected by an error. Refer to the device errata sheet for the necessary workaround [19].

`[MODIFICATION]`

- **MCU low-power modes**

  In case the application has stricter low-energy requirements but less restrictions on the adopted operating mode, the MCU can enter the CStop state. The function `Check_Sleep()` already implements the sequence of operations to perform upon entry/exit but it may still need to be modified to fit application-specific details. Additional documentation on subsystem-specific power management procedures can be found inside the power drivers files `stm32mp1xx_hal_pwr.c` and `stm32mp1xx_hal_pwr_ex.c`. Check also the *Power management issues* paragraph to know the risks associated to a different power management strategy.

`[INFORMATION]`

- In-depth analysis of the application **power management**

  As mentioned in II-C, the active wake-up sources are used to decide whether the system can enter or not an operating mode. In the proposed application the PD_Core cannot be switched-off because the ETH is a wake-up source. This means that, if ETH was the only wake-up source of the application, the system could enter Stop or LP-Stop but not LPLV-Stop [16] [15]. The LPDS bit inside the PWR_CR1 register selects the operating mode between the two that are available. Its value can be checked in debug mode with STM32CubeIDE. Because the bit is set to 1 the selected operating mode in case the system was able to go in low-power mode would be

LP-Stop. The fact that the system cannot enter in LPLV-Stop can be empirically demonstrated by checking that LVDS is set to 0 inside PWR_CR1.
The current system power mode can be verified through the PWR_ON and PWR_LP pins.
Even though the system is able to enter in LP-Stop, it always remains in Run mode III-D. In fact, when the system is in Stop/LP-Stop, the ADC is not active, so it cannot sample the signal of the monitored sensors, and both the ADC and the IPCC have no wake-up capability (table 34 of the RM [5]). To prevent the system from moving to Stop/LP-Stop, it is sufficient to keep either MCU or MPU in CRun/CSleep.
The current implementation uses the MCU in CSleep and the MPU in CStop because the CA7 consumes more power than the CM4. However, in case we wanted to put the MCU in CStop and the MPU in CRun/CSleep, it is important to remind that, in CStop, only EXTI events can wake-up the subsystem and that the subsystem peripherals clocks are stopped (section 9.6.6 of the RM [5]). This means that the ADC may not be enabled and that, even if it was, its interrupts could not be used to wake up the MCU because they are not associated to any EXTI event (section 21.1/.3 of the RM [5]). In case the power
To verify that the MPU enters in CStop after a `systemctl suspend` it is sufficient to check the value of PDDS, CSTBYDIS and STPREQ_Px. PDDS and CSTBYDIS are fields of the PWR_MPUCR register while STPREQ_P[1:0] is the only entry of RCC_MP_SREQSETR. By inspecting the registers in debug mode, it is possible to observe that PDDS and CSTBYDIS are both set to the appropriate values (0 and 1) before the MPU is suspended. If just one of them was set to the appropriate value, the CStop entry condition would still be satisfied. Unlike the previous bits, STPREQ_P is set after the suspension is issued. Each of these bits indicates if the associated MPU processor allows the entry in CStop mode for the MPU domain when the complete MPU is in WFI (page 728 of the RM [5]). In our case, they are both set to 1.

- **Monitor power and clock**

  When the system is in a low-power operating mode, some clock and power domains are turned-off to save energy. To check that the operating mode is the expected one, it is sufficient to observe the appropriate registers and verify that all the entry conditions are satisfied. A complementary approach to the problem is to sample the analog signals that are affected by the power-state change. The solution to adopt depends on the monitored signal:

  - STM32CubeMonitor-Power allows to analyze the power/low-power consumption of the target board. However, it requires additional hardware (i.e. carrier board, probe, etc) to perform the voltage measurements [18].
  - V_{DDCORE} supplies the digital core domain and depends on V_{DD}. In Stop, LP-Stop and LPLV-Stop the regulators supply V_{DDCORE} to retain the content of registers and internal memories, while in Standby mode its domain is powered-off. As already mentioned in section A-A, this signal can be easily monitored through ADC2.
  - If a precise measurement is not required, another alternative is to use the Power Voltage Detector (PVD) to monitor V_{DD} and/or the Analog Voltage Detector (AVD) to monitor V_{DDA}. These modules set an appropriate flag when the traced signal goes below/above the configured threshold.
  - Some clock domains are turned-off as soon as the system moves into Stop so it can be important to monitor their activity by redirecting specific clock signals on external pins.

  For additional information on these topics check section 9 of the RM [5].

- **MPU wake-up source recognition**

  The MPU can be woken up by two different wake-up sources: the Ethernet module and the IPC Controller. The generation of an interrupt by either of these peripherals is recorded inside the file `/proc/interrupts` together with the associated EXTI event. The entries of interest are:

  - `end0` interface (ETH): Ethernet global interrupt (ID:93 of the GIC) and Ethernet wakeup interrupt (PMT) [ETH1_WKUP] (EXTI event 70)
  - IPCC: IPCC_TX0 free interrupt (CPU0) (ID:133 of the GIC) and IPCC interrupt CPU1 (CPU0) (EXTI event 61)

  Although checking the fired interrupts may seem a good method to identify the peripheral that woke up the MPU, it is difficult to distinguish between the interrupt that actually reactivated the subsystem and normal interrupts. Normal Ethernet interruptions may be received right after the MPU is woken up by an MCU message and viceversa. The file `/sys/kernel/debug/wakeup_sources` is a better choice because it is updated only on wake-up events. However, in the proposed sensor node, Mailbox (IPCC) wake-up events are not recorded even though there is an entry. In case more than two wake-up sources with this problem were present, it would be impossible to spot the unit that generated the event with this file.

- **Wake-on-Lan mode**

  As explained in section III-D, the Ethernet peripheral is a wake-up source of the MPU subsystem. To be more specific, when the MPU is in low-power mode, the generation of a wake-up event is enabled by the Wake-on-Lan feature that, on the STM32MP157F-DK2, supports both unicast and magic packets. These modes are selected inside the Systemd service `wol@end0.service` respectively with the *u* and the *g* flag, also at the same

time. Neither of the available solutions is the best in all scenarios. The most appropriate one has to be chosen depending on the implemented network architecture.
Unicast packets may be better when the user client communicates directly with the boards through MQTT, like in the proposed implementation, because it does not need another node in the boards local network (i.e. the proxy client) to wake them up. However,the unicast mode causes the wake-up of a board on every unicast packet it receives, not necessarily one to flag a threshold update. Besides, each time the MQTT session expires, the broker sends a DISCONNECT packet to the client that has been disconnected, waking up the board. A new connection is established, even though it may not be necessary at that time. In this scenario, the board wastes energy since the MPU could have remained in low-power mode. The longer the maximum inactivity of a board the higher the energy wasted. Obviously, a proxy client is an additional node with a certain power consumption. This means that, to choose the best solution in each scenario, an accurate power analysis should be performed.
When the alternative architecture described in section A-C is considered, the unicast mode and the magic packet mode are not so different in terms of power consumption because the proxy client is needed anyway.

APPENDIX B
CLIENTS DEPLOYMENT

While the user client, the MQTT broker and the proxy client can run virtually on every device, the board client has to be deployed on the STM32MP157F-DK2 board. As explained in section III-C, the network/IPC interface is implemented on `ipcc-interface_noSusp.py` but the script that is actually used in the final product is `ipcc-interface_noSusp.service.py`. In fact, the board client is automatically launched at the startup of the Linux environment with the Systemd service `app_interface.service`. This service depends on `comm_setup.service`, because the IP communication needs to be already set up, and on `network-online.target`, because the client requires to access the network. When the dependencies are satisfied, the application is executed and, ideally, runs forever. In case of protocol errors that could compromise the internal coherency of the application, the custom exception AppException is raised. The error is logged on the board, the user client is warned and a non-clean exit is performed. The last step is especially important because the service is configured to launch again the application, for a maximum of 3 times per day, if it fails. When the restart limit is hit, the user needs to manually fix the cause of the failures. The exceptions that are not explicitly handled are not notified to the user but are still logged on the journal accessible with `journalctl`.

A. Tips and tricks

- In case there are multiple network interfaces, it is suggested to add the flag `--any` to the service `systemd-networkd-wait-online.service`. This allows to exit the service as soon as one interface is connected, otherwise, the default behavior is to wait for all interfaces to be active.
- If a service hits the restart limit, the command `systemctl reset-failed` is useful to reset the failure counter, otherwise Systemd will continue to consider the previous relaunches.

B. TTY interface management

When the board client is started as Systemd service, the TTY used to communicate with the MCU wrongly becomes the controlling terminal. This causes to interpret certain values in the wrong way. For example, the application process is killed with a SIGINT when it reads 0x2 from the CM4-CA7 interface. This probably happens because the service has no controlling terminal since `stdin` is linked to `/dev/null` by default. The problem is solved by opening the TTY interface with the O_NOCTTY attribute.

The error can be observed by monitoring the signals of the process with the command

`strace -p <process_id> -e 'trace=!all'`

when O_NOCTTY is not used. The process id can be found near the name of the executed command with `systemctl status app_interface`. An example of the output of `strace` is reported below:

```
strace: Process 3870 attached
--- SIGINT {si_signo=SIGINT,
si_code=SI_KERNEL} ---
--- SIGINT {si_signo=SIGINT, si_code=
SI_USER, si_pid=3870, si_uid=0} ---
+++ killed by SIGINT +++
```

REFERENCES

[1] MarketsandMarkets, *IoT Market by Offering... - Global Forecast to 2029* . https://www.marketsandmarkets.com/Market-Reports/internet-of-things-market-573.html

[2] Jara Ochoa HJ, Peña R, Ledo Mezquita Y, Gonzalez E, Camacho-Leon S., *Comparative Analysis of Power Consumption between MQTT and HTTP Protocols in an IoT Platform Designed and Implemented for Remote Real-Time Monitoring of Long-Term Cold Chain Transport Operations*. https://www.ncbi.nlm.nih.gov/pmc/articles/PMC10224120/

[3] Flespi, *HTTP vs MQTT performance tests*. https://flespi.com/blog/http-vs-mqtt-performance-tests

[4] Cedalo, *Understanding MQTT Quality of Service*. https://cedalo.com/blog/understanding-mqtt-qos/

[5] STMicroelectronics, *Reference Manual: STM32MP157xxx advanced Arm®-based 32-bit MPUs*. https://www.st.com/resource/en/reference_manual/rm0436-stm32mp157-advanced-armbased-32bit-mpus-stmicroelectronics.pdf

[6] STMicroelectronics, *STM32CubeMP1 MPU Firmware Package*. https://github.com/STMicroelectronics/STM32CubeMP1/tree/master/Projects/STM32MP157C-DK2/Applications/OpenAMP/OpenAMP_TTY_echo_wakeup

[7] STMicroelectronics, *Exchanging buffers with the coprocessor*. https://wiki.stmicroelectronics.cn/stm32mpu/wiki/Exchanging_buffers_with_the_coprocessor

[8] STMicroelectronics, *IPCC internal peripheral*. https://wiki.stmicroelectronics.cn/stm32mpu/wiki/IPCC_internal_peripheral

[9] STMicroelectronics, *STM32MP15 RAM mapping*. https://wiki.stmicroelectronics.cn/stm32mpu/wiki/STM32MP15_RAM_mapping#Memory_mapping

[10] STMicroelectronics, *Coprocessor management overview*. https://wiki.stmicroelectronics.cn/stm32mpu/wiki/Coprocessor_management_overview

[11] STMicroelectronics, *Linux RPMsg framework overview*. https://wiki.st.com/stm32mpu/wiki/Linux_RPMsg_framework_overview

[12] STMicroelectronics, *Linux Mailbox framework overview*. https://wiki.st.com/stm32mpu/wiki/Linux_Mailbox_framework_overview

[13] STMicroelectronics, *Linux remoteproc framework overview*. https://wiki.st.com/stm32mpu/wiki/Linux_remoteproc_framework_overview

[14] STMicroelectronics, *Coprocessor resource table*. https://wiki.st.com/stm32mpu/wiki/Coprocessor_resource_table#How_to_add_RPMsg_inter-processor_communication

[15] STMicroelectronics, *Power overview*. https://wiki.st.com/stm32mpu/wiki/Power_overview

[16] STMicroelectronics, *STM32MP1 series low power management*. https://www.st.com/content/ccc/resource/training/technical/product_training/group1/6b/eb/9d/db/1b/80/4b/c9/stm32mp1-series-low-power-management/files/stm32mp1-series-low-power-management.pdf/jcr:content/translations/en.stm32mp1-series-low-power-management.pdf

[17] STMicroelectronics, *AN5109: STM32MP15x lines using low-power modes*. https://www.st.com/resource/en/application_note/dm00449434-stm32mp1-series-using-low-power-modes-stmicroelectronics.pdf

[18] STMicroelectronics, *STM32CubeMonPwr*. https://www.st.com/en/development-tools/stm32cubemonpwr.html

[19] STMicroelectronics, *STM32MP151x/3x/7x device errata*. https://www.st.com/resource/en/errata_sheet/dm00516256-stm32mp151x-3x-7x-device-errata-stmicroelectronics.pdf

[20] https://community.st.com/t5/stm32-mcus-wireless/seeking-mqtt-implementation-example-for-stm32h7b3i-dk-with-built/td-p/638617

[21] Paho MQTT, *Python library documentation*. https://pypi.org/project/paho-mqtt/

[22] Eclipse, *Paho MQTT Github repository*. https://github.com/eclipse/paho.mqtt.python

[23] Eclipse, *Paho MQTT documentation*. https://eclipse.dev/paho/files/paho.mqtt.python/html/client.html

[24] Eclipse, *Mosquitto*. https://mosquitto.org/

[25] Python, *Global Interpreter Lock (GIL)*. https://docs.python.org/3/glossary.html#term-global-interpreter-lock

[26] Stack Overflow, *GIL not sufficient to have application-level consistency in Python*. https://stackoverflow.com/questions/105095/are-locks-unnecessary-in-multi-threaded-python-code-because-of-the-gil