POLITECNICO DI TORINO

ELECTRONIC ENGINEERING

Master's Degree Thesis

Modular Framework for MCU Programming

Study and Development

Supervisor
Prof. Guido Albertengo

Candidate
Riccardo Midena
Matricola: 233149

Internship Tutor
Santer Reply S.p.A.
Ing. Daniele MARIETTA BERSANA

Settembre 2018

In partnership with:

Concept Reply, B.U. of Santer Reply S.p.A.

Abstract

This thesis discuss about a framework implementation which aims to simplify and speed up the writing of a firmware for an embedded system. The goal is to create a system able to drop drastically the implementation time of IoT nodes. To obtain this result, the idea is the creation of a totally generic modular system which exploits several blocks run concurrently that implements functions agnostic to the data elaborated inside.

The first part of the thesis describes the theorical aspects, such as the features of an RTOS and the parameters to compare them, in order to select the more suitable one for the framework implementation. Afterwards, the most important protocols used in the realization of the blocks are presented, such as the MQTT and the I2C.

In the second part, the framework is described in depth; starting from an analysis of the basic module and the interaction with other ones, to the explanation of the different implemented blocks.

Finally, in the last part, it is shown an analysis of performance and complexity of the code in project implemented with the described framework.

Contents

Chapter 1

Introduction

In the last decade, industries have begun to evolve in what is considered the fourth industrial revolution. This revolution has a profound impact within three directions of development:

- Usage of data, computing power and connectivity: represented by the Big Data, the Internet of Things (IoT) and the cloud computing
- Analytics: the gathered data can be used with the machine learning concept to improve the machinery efficiency
- Human-Machine interaction: showing to the users real time data about the status of the system makes possible to improve the interaction with new features and higher level of control.

It changes the entire production process transforming analog and centralized workflows into digital, closely connected and decentralized ones. The goal is to involve production machines able to continuously collect data about status, locations and work status. This gives a higher control on the overall process and machinery, leading to a reduction of the costs, a better resource efficiency (by limiting their waste) and services customer oriented.

Furthermore, more connected devices could also bring to the possibility of a self-organized production process.

Cyber Physic System (CPS)

Internet driven, self-controlling and sensor aided production system are the main features on which the industrial machinery are based in what is called Cyber Physic System. [1]

CPS are complex systems that interlace the physical world, through sensors, with the virtual one, including software on the device that carries out the measurements and the connection that makes all the devices potentially connected to each other. As well as in industry 4.0, these concepts are also expanding in common devices present in every fields, starting from the traffic lights in the streets that can monitor the traffic, communicating the information to the cloud and taking in-place actions, up to smartwatches monitoring user's health parameters.

All these embedded devices have an increasingly complexity in the implementation, thus delaying production times, preventing simple upgrading of the software.

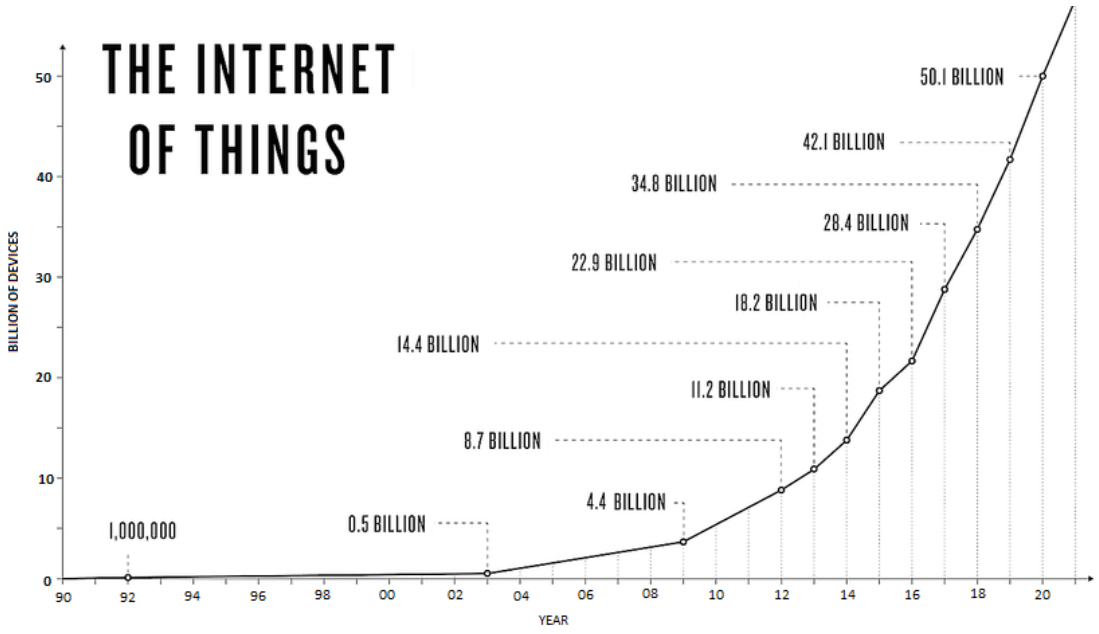


Figure 1.1: Evolution of IoT devices in time

The goal of this thesis is to simplify the creation of new connected devices (i.e. IoT nodes) realizing a framework which aims to the creation of complex systems through intuitive, fast implementable and simple code.

Project presentation

The idea is the creation of a totally modular implementation where the functions of a device are subdivided in interchangeable blocks derived from a basic structure. In this way the implementation of all the devices can be standardized and the simple usage of different blocks allows to connect each other that lead the device to sample from sensor and dispatch the measures through the network. Since the firmware in microcontrollers is typically written in C or C++, this is the programming language chosen to implement the framework, thus allowing a high portability and compatibility with different system.

Chapter 2

Background

After the advent of microcontrollers, most of the fields of our lives had started to become smarter by embed them in every kind of system. Until nowadays, the greatest number of embedded systems didn't adopt any Real-Time Operating System (RTOS), this used to be a reasonable situation since the tasks they had to fulfill were usually very simple and specific.

With the expansion of the internet of things (IoT), the microcontrollers able to establish a connection to the network are increasingly spreading, thus, in addition to the simply monitoring or control of information from external sources, they also acquire the ability to send those data (that usually need an elaboration first) to cloud servers to be used in a second moment.[2]

In order to handle all those tasks, the system need to increase the abstraction by adding a new layer between the firmware and the hardware that is represented by the operating system.

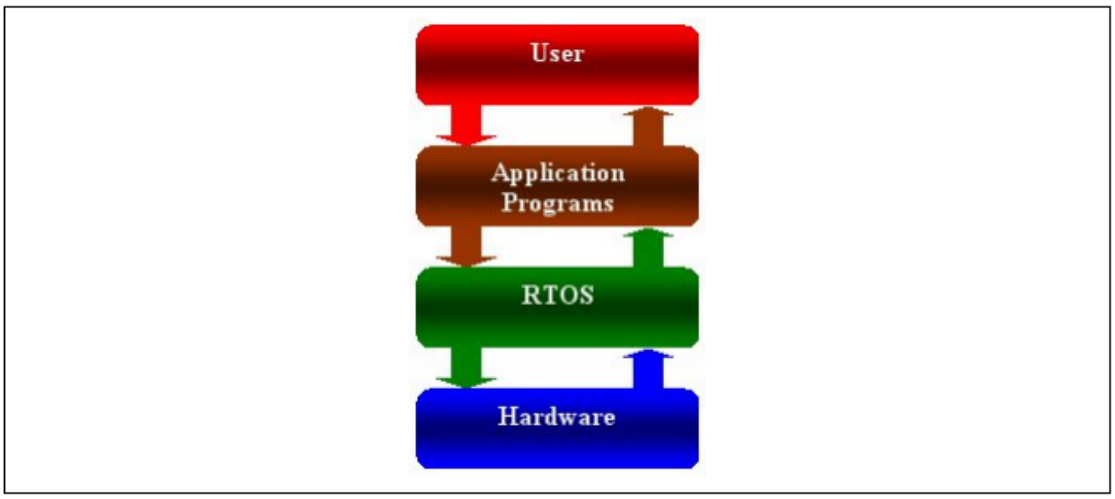


Figure 2.1: Abstraction level of a system

2.1 RTOS

" Real-Time Operating System (RTOS) is a multitasking operating system intended for real-time applications". [4]

The Operating system has an essential role in complex systems since it is the one that, running in the background on a microcontroller, provides an interface between the application program and the hardware resources in which it is executed. With the improvement of the technologies, even the cheaper microcontroller systems can handle several peripherals and perform various duties, leading to larger firmware too complex to manage.

Since those duties are usually independent each other, a way to simplify the whole software is to divide it into pieces commonly called "tasks" or "threads".

The RTOS is the system that takes care of the context switching, i.e. its save all the resources values (registers, stack, etc.) related to a single thread and then resumes the context of another thread. In this way it can manage the processing time among al the different tasks the software must execute.[3]

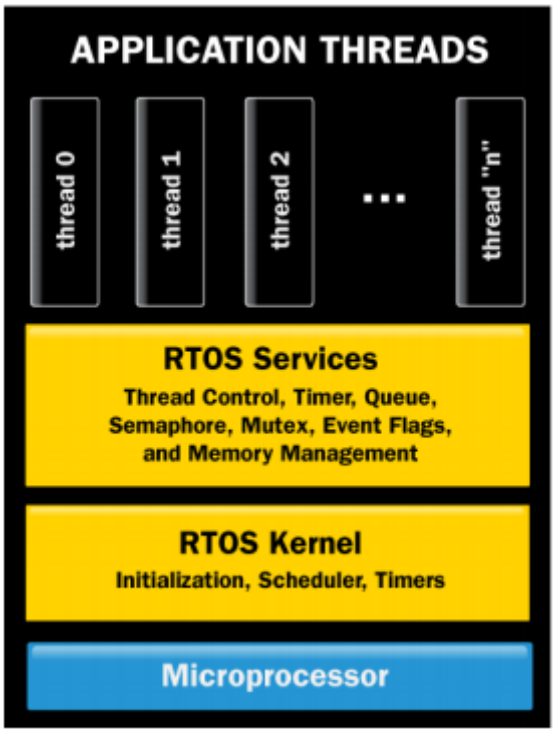


Figure 2.2: Threads managing system

2.1.1 RTOS classification

It is possible to classify in three categories the different RTOS available on the market based on how much the different thread in the system are strictly bounded to the time deadline of execution:

- Hard real-time system: the tolerance for missed deadlines is extremely small or zero. A missed deadline can lead to catastrophic and dangerous result for the system or an eventual user.

- Firm real-time system: the failure in the compliance with the deadline could be acceptable since it leads only to a wrong result that has to be discarded (e.g. cruise control).

- Soft real-time system: deadlines may be missed, and the result obtained could be used with a consequent reduction in the quality of the system.

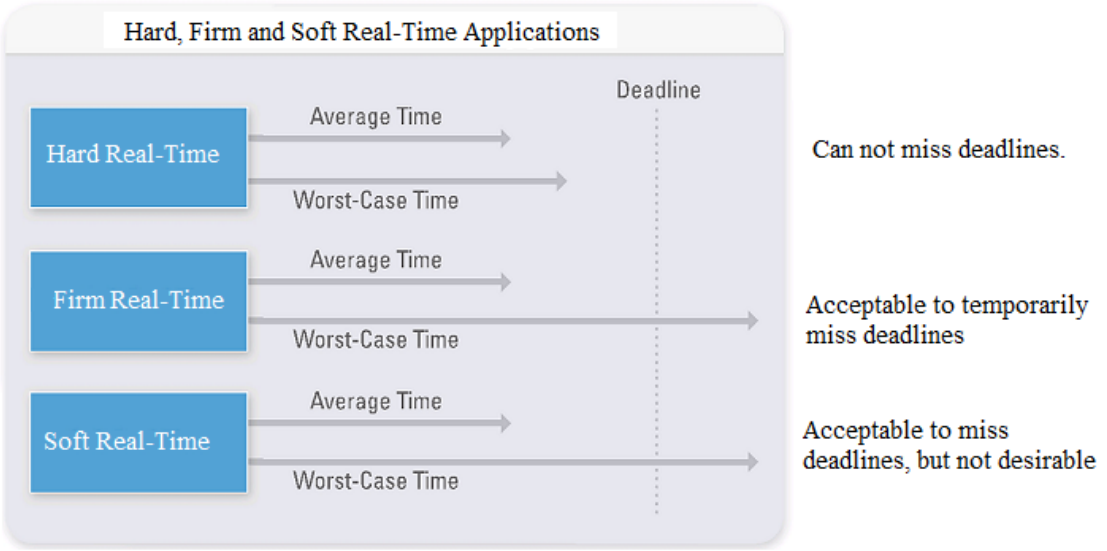


Figure 2.3: Categorization of RTOSes

2.2 Thread management

As mentioned above, the program can be designed in different ”chunks” of codes distinguished by different goals and deadlines called threads.

During the instantiation of a thread, it is associated with a priority (set to ”normal” by default) that indicates the level of importance of its execution. That means that threads with lower priority have a less critical task to execute and are generally the ones chosen to be executed last.

Managing those tasks require a mechanism called scheduler that decides which task has to be executed in order to respect the three time-critical properties: release time, deadline and execution time. Release time represent the point in time from which the task can be executed, Deadline is the point in time by which the thread must be completed and Execution time indicates the time required from the task to be completed.

Since only one task at a time can be executed, the system has to organize them in different states:

- the microprocessor is executing the task. If the processor has a single core, only one task at any given time can be in this state.

- Waiting: the task is waiting for either a temporal or external event. They can enter in this state also to wait not ready resources such as queues or mutex. In this state the task do not use any processing time.
- Sleeping: the task is created but it is not active
- Ready: the task is able to be executed but it does not since another one with equal or higher priority is running

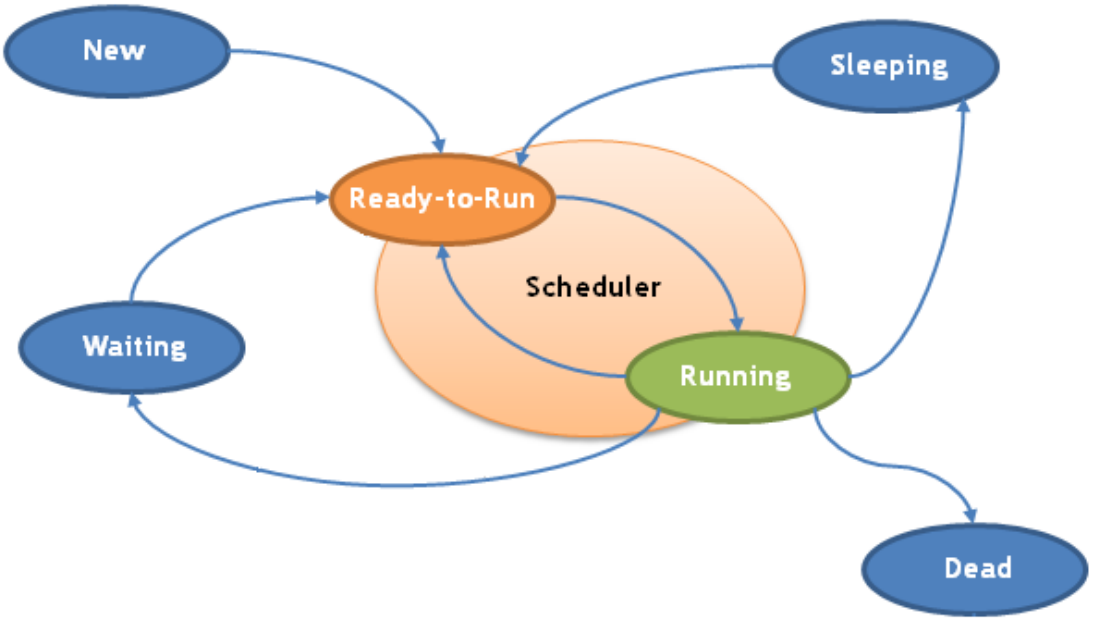


Figure 2.4: Threads states diagram

After than the thread is completed, or it receive an abort signal, its context is released.

2.2.1 Scheduler

The figure 2.4 shows how the tasks can change to the running state only if they are in the ready state. The scheduler, which is the one that decides which of the many

ready threads has to be executed, has the role to maximize the CPU utilization and minimize the waiting time.

The scheduler can be classified into two categories: non-preemptive and priority-based preemptive. In the Non-preemptive scheduling, the switch between the ready and running state can occur only when the current executing task return the control to the RTOS. Once this happen, the scheduler chooses among the different thread which is the one with higher priority. In a priority-based preemptive scheduling, the scheduler can give in any moment the control of the processor to the task with higher priority.

In case of more threads with the same priority level are waiting in the ready state, the scheduler could be implemented to follow a round robin algorithm. This algorithm allows the execution of all the task by swapping periodically the thread executed by the CPU.

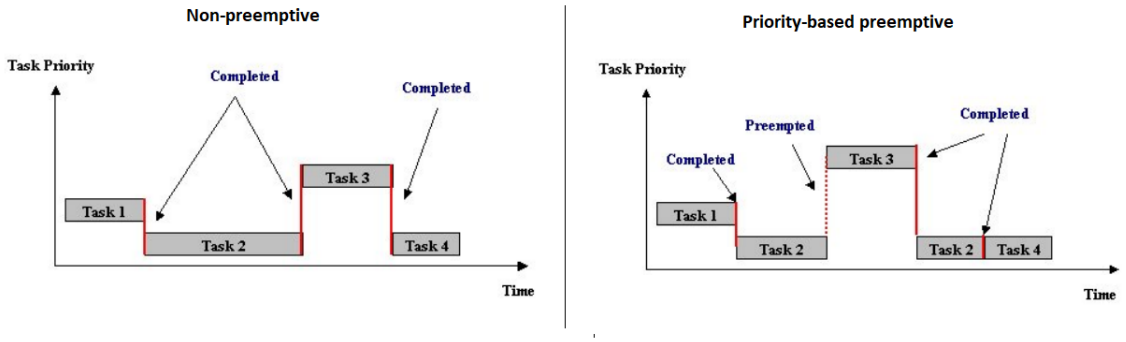


Figure 2.5: Preemptive vs Non-Preemptive schedulers

2.2.2 RTOS vs General purpose OS

General purpose operating systems such as Microsoft Windows and Mac OS can provide a suitable environment to develop and execute non-critical application. However, there are several aspects that highlight how not all the use cases can be handled properly by non-real-time system.

In a real-time system the correctness of the operations is not only determined by the correctness of the procedures it execute but also from the meeting of the

deadlines associated with each task. For these reasons, the real-time system must follow the programmer's priority more strictly and so it is characterized by the presence of a preemptive scheduler.

Furthermore, the real-time systems are often associated with a high reliability since generally they are designed to work for day, months or even year without stopping. A "watchdog" is then a feature typically included to automatically restart the microcontroller in case of failure.

2.2.3 RTOS metrics

Since the fields where microcontrollers are present are many and different from each other, it is possible to find several different RTOS on the market optimized for different task. The choice about the system on which design the project can be subdivided in two main aspect: Technical features and Commercial features.

Technical features

Scalability

The scalability is the ability to an RTOS to adapt itself to the application-specific requirements. Depending on the number of required functionalities, the RTOS should be capable of modify modular components such as file system, drivers and protocol stacks. The main reason for scaling is to optimize the usage of the memory footprint.

Portability

Often, with the usage of new technologies or the evolution of systems, the application may outgrow the hardware it was originally designed for, hence the RTOS should be able to be ported between processor architectures and specific target systems.

Performance

The performance of a RTOS denotes its ability to fulfill its timing requirements. More tasks (and so more deadlines) lead to a shorter time to accomplish them and therefore to a need for a more powerful hardware. However not only the hardware can improve the performances since these are closely related to the metrics of the

kernel such as the time of the context switching, the interrupt latency. Generally, the performance of an RTOS is measured through the usage of benchmarks producing timestamps when a system call starts and finish and then analyzing all the features.

Ecosystem

An underestimated aspect too often not considered during the choice is the environment in which the RTOS works. This can be represented from many aspects that can ease the integration and reduce enormously the time to market, but the most important are the set of development tools (e.g. debugger, compiler) and the support from the developers and the community.

These two aspects are often in contrast each other since the best support can be found in open source RTOS but at the same time these lacks a good development environment.

Safety

The safety is an aspect required especially from critical system such as medical ones. The safety metrics has to be certificate before the completion of the project and this could require time and high costs. For this reason, in the market there are some pre-certified RTOS build following the regulatory Authorities.

Reliability

Embedded system must be reliable since depending on the application, it is possible that the system requires to be operative for long period of time without any human intervention. The reliability of a system can be categorized by its downtime in a year, that is the time in which the system is not available due to a fail. This aspect is highly correlated to the safety.

Security

This aspect is becoming more and more important, especially for the Iot world. The security is the feature that protect the information elaborated by the system and blocks eventual unwanted and not by the designer behavior. This aspect is strictly correlated to the ecosystem since the security needs continuous update.

Commercial Features

Costs

This is an important aspect for the company which wants to develop new embedded system based on a RTOS. The RTOS on the market at the moment are more than 100 and the prices range from $70 to over $30000. Moreover, the RTOS vendors may also require royalties on a per-target-system basis and even new features or support could be chargeable.

Vendor

The company behind the RTOS is fundamental since it must be a reliable supplier able to support not only the current product but also its eventual evolution. For this reason, it is always preferable to choose a pro-active supplier with a good reputation.

2.3 MBED

As pointed out in the introduction, the framework is primarily developed so that it is agnostic to the operating system on which it is executed. However, to manage the various threads, a RTOS is necessary. The choice is made by the comparison of the different RTOSes present on the market by the analysis of the metrics described above. The chosen RTOS is MBed OS 5.

2.3.1 Arm Mbed

Arm Mbed is an ecosystem which sets the foundation to the development of low-power, connected and secure Iot devices at scale. It is meant for all those systems which contain one or more internet-connected devices based on 32-bit ARM Cortex-M microcontrollers and it born with the aim to lead of the IoT environment. Developed by Arm, one of the most influent company in the microcontroller design, it is supported by over than 60 partners such as Analog Devices Inc, IAR Systems, NXP, Huawei, Renesas, Realtek, Schneider electric, Texas Instrument, STMicroelectronics and many more.

Mbed-OS is a free and open-source RTOS, thus guaranteeing a more reliable

system and a large community improving it constantly [5]. Arm Mbed offers several services such as:

- **Mbed OS developer community:** a forum in which the community can receive support or improve the environment.
- **Mbed toolchain:** the compilation of the Mbed project could be done with the online IDE Arm provides using the ARMCC C/C++ compiler, without the installation of any environment and compiler, or through development environments such as Keil uVision, IAR Embedded Workbench, and Eclipse with GCC ARM Embedded tools.
- **Mbed Device Connector Service:** it allows to connect the IoT devices to the cloud without having to build any infrastructure ensuring a high level of security and providing easily-integrated REST APIs.
- **Pelion Device Management:** this service provides the ability to easily and flexibly manage a wide range of IoT devices. It helps connect new IoT devices on global networks, administer them and extract real-time. Associating it with the Mbed Cloud Portal it is possible to control the devices through an online dashboard.
- **Mbed OS:** the RTOS based on the open-source CMSIS-RTOS RTX on which the firmware are meant to run. It supports a multithreading real-time and deterministic software execution. Released under an Apache 2.0 licence, allows the use of Mbed OS in commercial and personal projects.
- **Mbed TSL and uVisor:** with the grown of IoT devices integrated into our life, the potential security violation has increasingly become an aspect to be taken in consideration. The idea to increase the protection and mitigate the consequences, mbed offers these two security-specific building blocks with the idea to standardize the protocols. Whereas Mbed TSL takes care about the security of the communication on the internet throught the usage of the standard protocols TSL and SSL, uVisor is a supervisory kernel that restrict the access to the hardware (e.g. memory and peripherals).

The application code supported by the Mbed system is a C/C++ firmware that uses the APIs provided by Mbed itself. These APIs are based on the same architecture and so are agnostic to the board on which they are compiled provided that the different boards have the same used interfaces. This approach leads to maximum portability of the various projects.

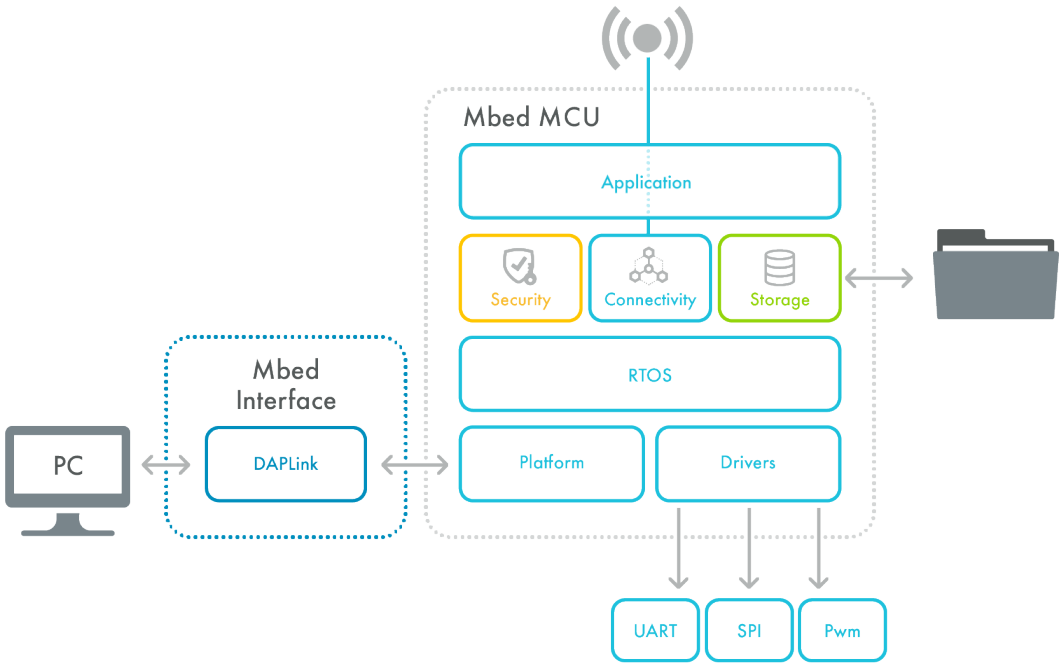


Figure 2.6: Sketch of a typical Mbed board's hardware architecture

2.4 Protocols

In order to create an IoT node it is necessary to choose a proper communication protocol to connect it to the cloud. One of the most common protocol used is the HTTP one through the REST architecture (Representation State Transfer). HTTP (Hyper Text Transger Protocol) is an ASCII protocol purely textual that bases the communication on the request-response pattern, however each communication transfers a quantity of side-information unsustainable for a continuous transmission from a microcontroller which generally has low quantity of memory.

Among all the minor used protocols such as XMPP, DDS, or STOPM, the more

relevant is the MQTT one.

2.4.1 MQTT

Developed by Andy Stanford-Clark and Arlen Nipper in the 1999, the MQTT (Message Queue Telemetry Transport) is a binary protocol based on TCP/IP in the ISO/OSI model.[6]

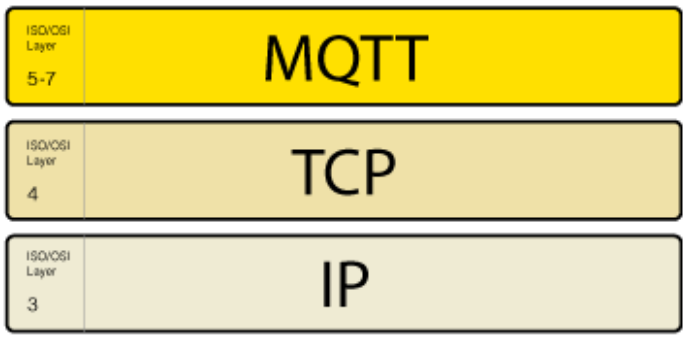


Figure 2.7: MQTT in the ISO/OSI model

The main goal of this new protocol was to perform a communication between multiple devices with a limited use of the bandwidth and a minimal consumption of energy. Unlike HTTP communication, the MQTT protocol use the Public/Subscribe paradigm which is event-driven.

Therefore, the architecture is formed by a central server called broker which has the duty of dispatch the several messages among the different clients connected with it. To know how sort the different messages, the publisher clients associate a topic to them which will be used from the broker as a routing information. All the clients interested in a specific information from another client will warning the broker with a subscription to the interested topic.

This approach leads to different advantages:

1. **Decoupling:**The star topology allows the decoupling between the publisher and the subscriber clients. This decoupling can be viewed in several levels:

- *Spatial decoupling*: the two clients has no need to know or see each other with a direct connection (e.g. they don't need to exchange their IP or the port for the communication) leading to a simpler infrastructure.

- *Time decoupling*: Since is the broker that manage the dispatching of the data, the publisher and the subscriber can run at different time and the message can still be transmitted correctly. In microcontroller this is a useful feature since it is not necessary to wait other devices and, in case of inactivity, it can go in sleep mode, thus saving energy.

- *Synchronization decoupling*: The request-response model requires to interrupt the execution of the program to grant a high efficiency. Having a broker that stores the data and pushes the information to the client allows a higher degree of freedom in the time response.

2. **Scalability:**The grown of the system do not require the knowing of the whole infrastructure since add a new client requires only one connection to the broker. Furthermore, the event-driven approach allows an highly parallelization of the operations done by the broker, hence the structure of the system can reach a very high number of clients (obviously with the grown of the system the broker server will be stressed more and more).

3. **Reliability:**The MQTT broker is provided with a buffer in which it stores the data published on a topic in case the connection with a client subscribed in it is interrupted by any circumstances. Once the connection is restored the broker send all the buffered messages.

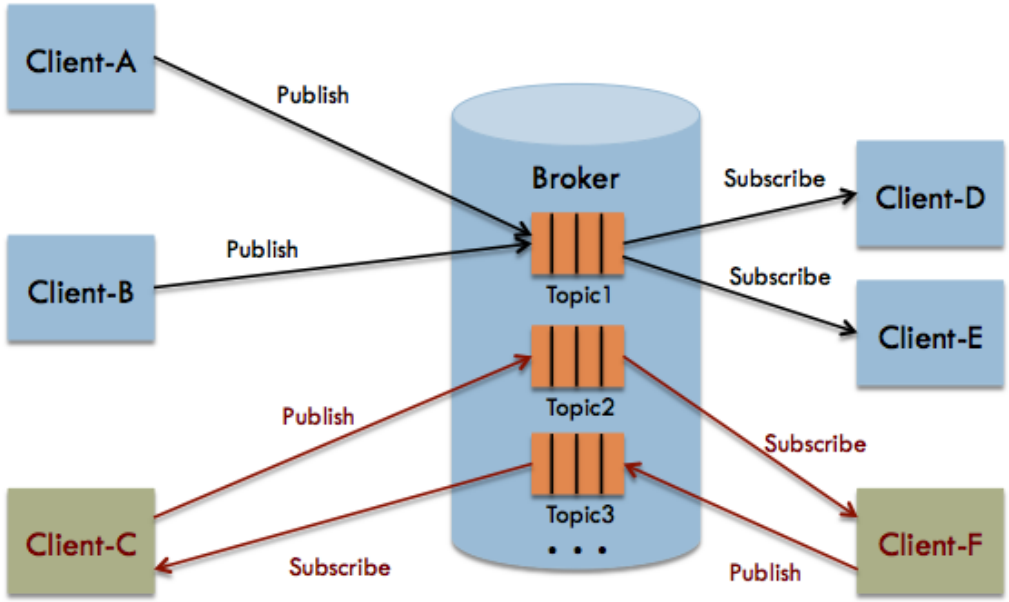


Figure 2.8: MQTT structure

MQTT Connection

The connection between the broker and the clients starts with a CONNECT message from the client and a CONNACK response. Once the connection is established, it remains active until the broker receive a disconnect message or it is not able anymore to see the client due to a network or a client issue.

The CONNECT message contains different fields that configure the relation from the two devices:

- *Client ID*: This identifier is unique for each client and identifies it to the broker
- *Clean session*: this is a flag which warns the broker that the session established by the client is a persistent one or not. In a persistent session if the connection is interrupted, the brokers save the information of the client such as its subscriptions, the messages with quality of service (QoS) above 0 and the existence of the previous session. In this way all the messages that the client was not able to receive are not lost and when the session is established

again, this will not require a big effort from the client that otherwise should to subscribe again to all the topics it is interest in. This latter feature is excellent for clients with limited resources.

- *Username and Password*: these fields are used to create a first weak level of security enabling the authentication of the client for the authorization of the operations

- *Will Message*: Useful to detect a failure from the client, this attribute is composed by different subfield such as Topic, QoS and Message. This is Last Will and Testament (LWT) message. The broker stores this message until an error occur in the connection with the client that set it. If the client disconnects without a correct procedure, the broker send the LWT message to all other devices subscripted to the topic indicated in this field on behalf of the client disconnected, otherwise it is discarded.

- *KeepAlive*: this field represent in seconds the maximum time interval in which the client and the broker has to communicate to ensure that the connection is still present. To ensure the respect of this timing, the client commits to send a periodic PING message to the broker.

Excepting for the client ID, the clean session and the keep alive, all the other fields are optional and can be omitted.

MQTT Operations

The whole operating mechanism is based on the publish and subscribe operations.

PUBLISH

The publish operation starts with a PUBLISH message from the client sending to the broker the attributes for that particular operation. The field of a PUBLISH operation are:

- *Packet ID*: it is present only where the QoS level is higher than 0 and it is used to uniquely identify the packet sent to the broker or by the broker to other clients.

- *Topic Name*: it is a character string used by the broker to filter the messages. This is a case-sensitive attribute and permit the usage of all characters, but it must correspond perfectly during the subscription. The MQTT protocol allows to create hierarchical topics through the forward slash character to simplify the managing of the subscription (e.g. home/kitchen/temperature).
- *Quality of Service (QoS)*: It is an agreement between the client that publish a message and the one that subscribe to receive it for assure the correct transmission.
- *Retain flag*: if set, this flag tells the broker to retain this message as the last known good value. This message will be sent to each new client subscribing to that topic. This is useful to inform the new subscriber immediately after its subscription about some sort of information (e.g. status of the publisher client, last value, etc.).
- *Payload*: this represent the real message of interest by the subscriber. The MQTT is data agnostic that means it is possible to send any kind of data (e.g. images, JSON, strings, XML, etc.). This allows the creation of generic function since the structure of the payload is determined by the use case.
- *Dup flag*: Used only for messages with QoS greater than 0, this flag is set when the message is being send more than once due to a missing acknowledge response.

SUBSCRIBE

The subscription of a client to a topic requires a SUBSCRIBE message with only two fields:

- *Package ID*: as the in the publish message, this identify uniquely the message.
- *List of subscription*: each subscription is characterized by the Topic and its QoS level. Unlike the publish topic, in the subscribe topic messages there are two wildcard characters that let the subscription to more than one topic simultaneously:
  - '+' : single-level wildcard replaces one topic level

– '# ' : multi-level wildcard covers many topic levels (the one where the symbol is used and all its sub-topic)

Quality of Service (QoS)

As mentioned above, the correctness of the communication can be assured by the QoS procedure. Each connection can be associated with 3 level of QoS:

1. ***QoS 0 - at most once***

   This level does not guarantee in any way the correct delivery of the messages. It is used when the loss of a message is not critical, when the connection is mostly stable or when it is required a high speed in the communication since it is composed by the only message from the client to the broker or vice versa.

Figure 2.9: MQTT QoS 0

2. ***QoS 1 - at least once***

   The sender waits for the acknowledge message from the receiver with the id of the package just sent. If the acknowledge is not sent back, an internal timer expires, and the message is sent again with the DUP flag set. This level of QoS is used when it is required the reception of the messages and the receiver can handle eventual duplicated ones.

Figure 2.10: MQTT QoS 1

3. ***QoS 2 - exactly once***

   This QoS level guarantee that the message is received only once. It implements a handshake protocol that requires time, so the drawback is that it slow the communication. It is used when it is critical for the application to receive exactly once each message.

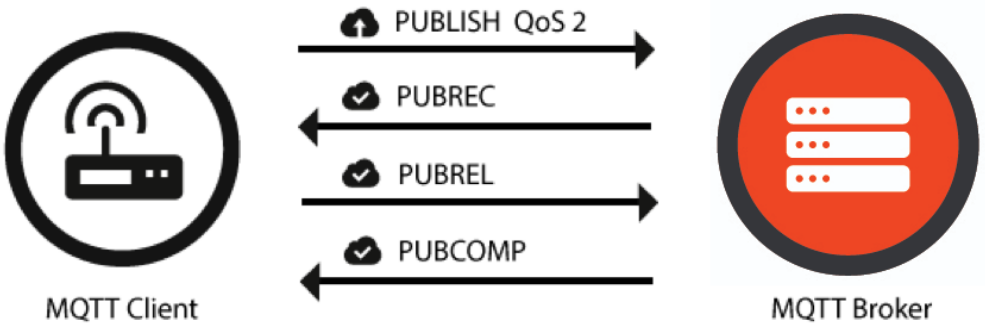


Figure 2.11: MQTT QoS 2

Chapter 3

Framework

The basic idea behind the project is to build a framework, using the C/C++ language, which helps with the creation of a network composed by several IoT nodes.

The final goal of this thesis is to reduce dramatically the time required to develop a fully working proof of concept for an IoT node that gather data from sensors, process the information, and send them to a remote server or execute in-site actions. The purpose can be achieved by the creation of a microcontroller-independent framework composed of functional blocks.

The implementation must be totally generic since the realized node has not any defined typical use case structure. This means that it must be able to execute every type of task it is implemented for being independent from the data types it interacts with (even with custom ones).

From the analysis of several commercial IoT systems it is possible to notice how there are common factors in the operation executed. Most of them can be grouped in three main categories:

- Sampling operations: the device receive some information from analog and digital sensors or other devices (field devices, high level protocols, etc.).

- Elaborating operations: the device elaborates those information to create different results, such as average, maximum or minimum calculation, filtering or grouping

- Output operations: the resulting information are dispatched on a network or are exploited in-place on actuators

All these operations are completely independents each other except for the information they exploit and so it is possible to create a fully modular structure where each module is represented by a thread handled by the RTOS on which the program run. The modularity of the system simplifies the implementation or modification of the firmware by reducing it to the addition or removal of blocks. The concurrency of the operations gives the possibility of independent development of new features, enhancing the possible growth of the project with time.

The final structure is therefore composed by input blocks, middle blocks, output blocks, configuration blocks (i.e. blocks that do not interacts with the data but configure the device) and an orchestrator which is handles the interactions among the various threads.

In figure 3.3 is shown a possible architecture obtained using the framework proposed in this thesis. In particular, the structure is made up by several input block, two middle blocks for data processing, a configuration block and two output blocks. It is interesting to notice the different path followed by data obtained from the sensors: some of them are processed by the middle blocks, others are forwarded directly to the output modules.

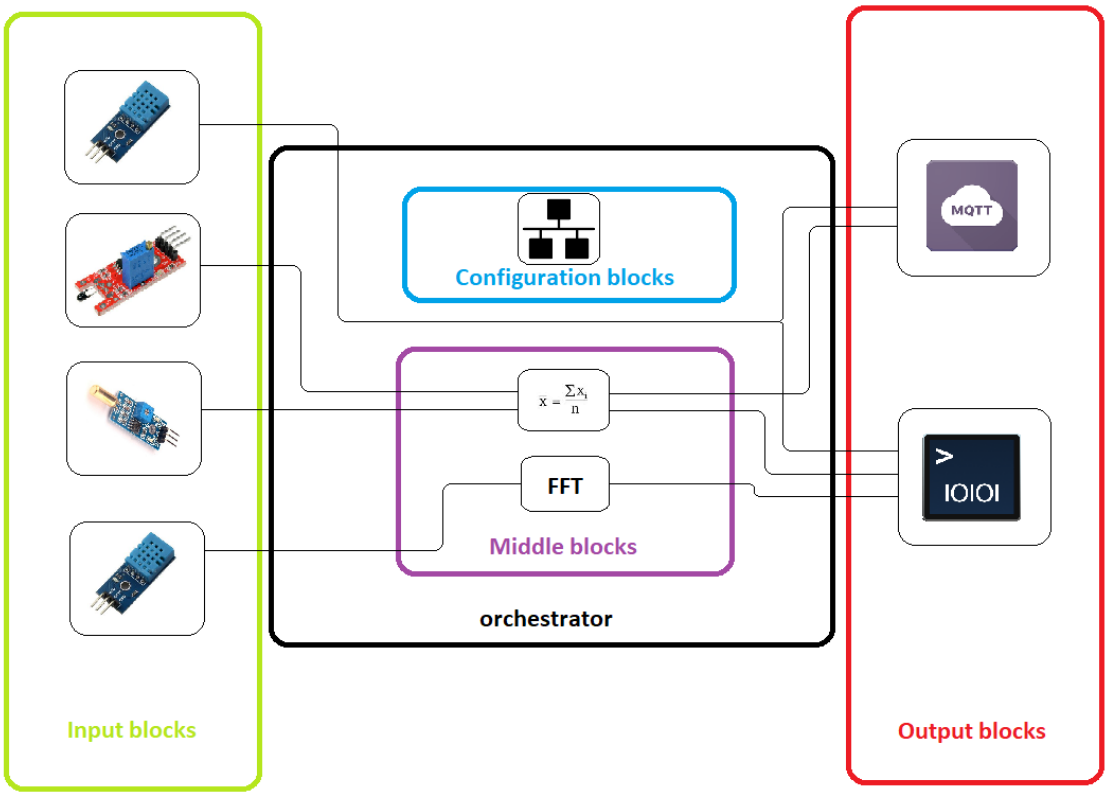


Figure 3.1

3.1 Block structure

The different type of blocks has several common characteristics and features, therefore is convenient to implement them as children of a single General parent block exploiting the C++ inheritance. This structure is convenient for several reasons:

- Standardization of the modules, for the developer is easier to use and understand how the blocks works knowing that most of the methods and members are shared
- Easier implementation of general function, exploiting the inheritance it is possible to pass a child class to functions that accept the general block as input
- Creation of new blocks, the process of development of new blocks is easier

due to the shared structure

3.1.1 General Block

The General block implements the data structure and interfaces that are inherited by all modules. Therefore, the data model and the interfaces used for the connection to other modules are already provided, simplifying dramatically the communication among blocks.

The implementation of the general block include different elements, among which the most relevant are:

- Circular queue: The Mbed libraries provide the queue structure, but it is implemented through a dynamic allocation and is a standard FIFO queue that does not allow to retrieve any data from it but only following the entry order. The circular queue included in this framework is implemented statically with a fixed number of elements set during the allocation of the block through the constructor. It allows the usage of any element in any position of the queue, and this lead to two main advantages that can improve the performances:

  - The blocks can retrieve subsequent data even if other blocks did not get theirs yet. That means that no blocks can stall due to others blocks but only if the queue is empty.
  - It is possible to create a structure where the handling of data does not require a mutex that lock the queue since both the provider and the consumer of the data are perfectly aware of which element use avoiding concurrent write or read operations.

  Each element inside the queue is associated with a counter that starts from the number of blocks that will request that data and decrease by one each time someone retrieve it. Once it is equal to 0 the element can be replaced.

- Destination Number: indicates the number of blocks connected to this block waiting for the data.

- Destination Address: an array built dynamically which contains the addresses of all the consumer block ”subscribed” to the provider block. The memory allocation is done only during the starting setting and so do not influence the run-time performance of the device.

- Source Link: a map structure used from the destination blocks, designed to associate the pointer of the source blocks, with a structure composed by three parameters that represent the state of the relation between the two connected blocks:

  – DataReadyNum: that indicates the number of data available in the associated source block

  – NextDataIndexQueue: it is an index that point to the next data in the queue of the source block

  – InputQueueLength: that indicates the length of the circular queue in the source block to accomplish wrap the NextDataIndexQueue every lap.

  Through an iterator that scans the key value (i.e. the pointer of the source block) the map is checked periodically with the polling technique and, if available, the data are retrieved.

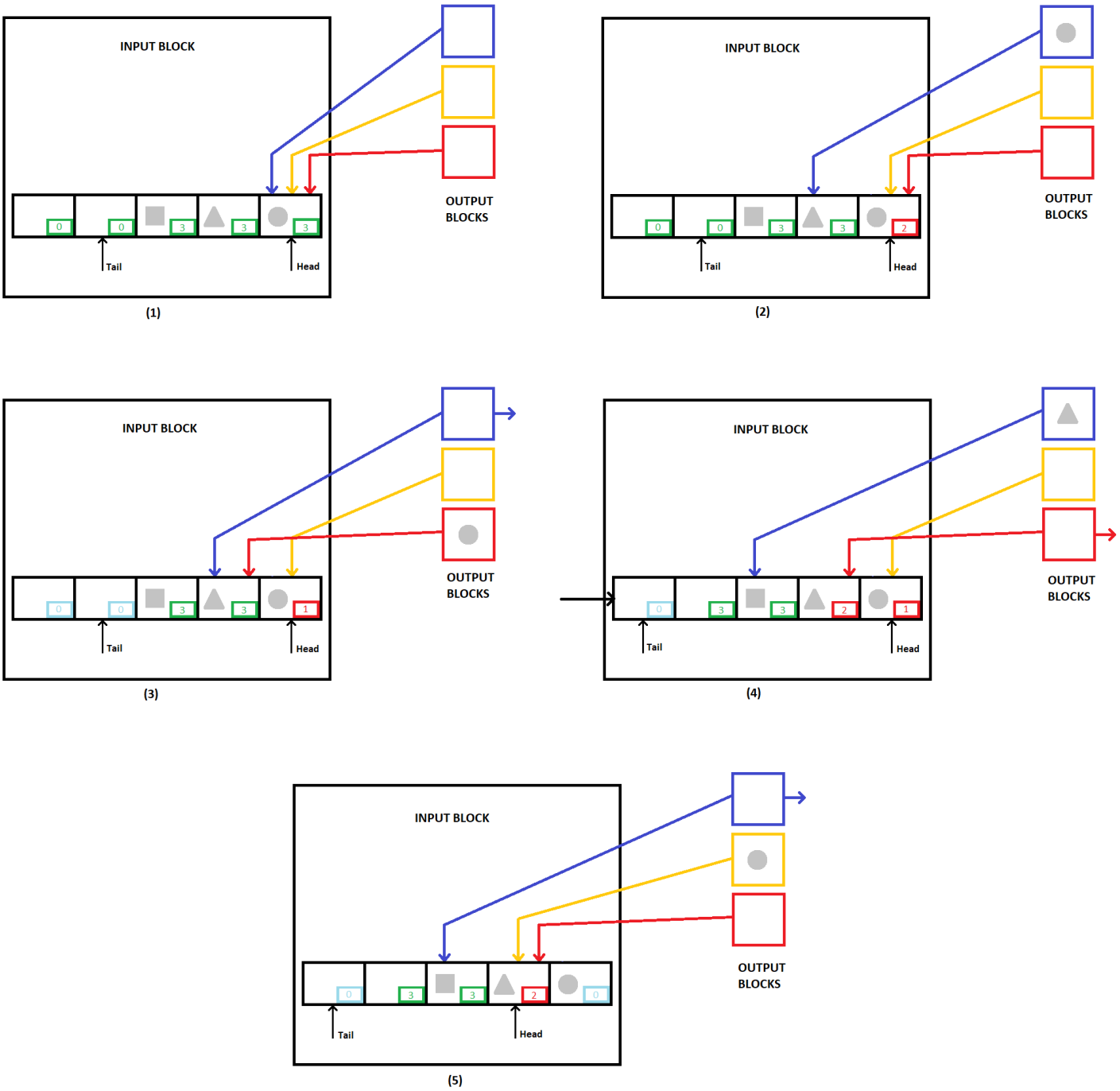


Figure 3.2: Steps executed during the data retrieving

- Thread pointer: the pointer that associate the block with the function it has

to execute.

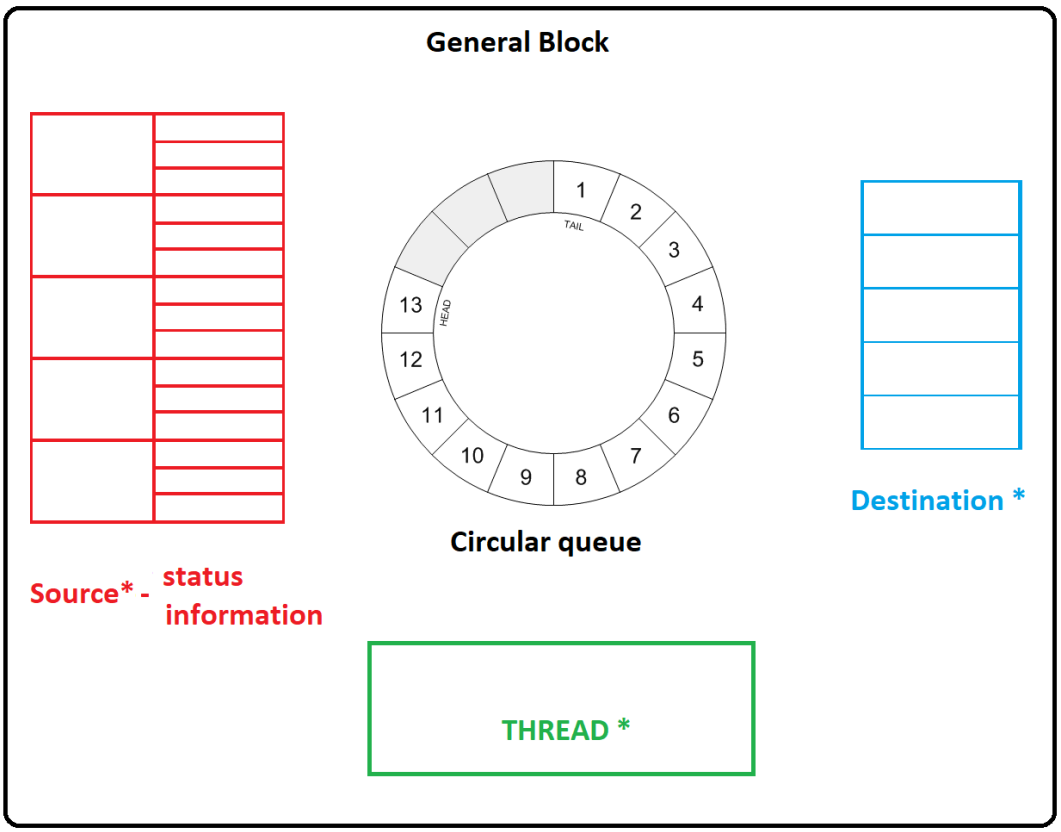


Figure 3.3: General block structure

Starting from the General block it is possible to implement the three main blocks.

Input Block

The input block is the one in charge to provide the data from the external world.

Since compliance with the sampling timing is one of the critical factors on efficiency and reliability of the system, this is the only block that must sets its priority to a higher level than the other threads. The RTOS, therefore, execute the context switch to give to the module the possibility to be executed as soon as it is ready (unless it is serving another sampling block).

The sampling method can be turned by working on the data structure implemented in the queue and adapting the thread with slight variations. The main method used are:

- Single sampling: a single sample is taken periodically based on the time set during the instantiation of the block.
- Burst sampling: in this case the sampling is done periodically but more than one single sample is taken at a time. This can be easily implemented with the creation of an array in the internal structure of the queue and with the addition of a loop repeated a number of time equal to the number of samples wanted in the burst. This sampling method is generally used with a following elaboration implemented in a middle block, such as average, fft and so on.

Unlike the middle and the output blocks which by default present a queue length equal to the single element, this block starts with ten elements unless the instantiation modifies it.

Middle Block

This block is the only one that execute a thread able to call an external function in order to elaborate the data retrieved from the previous block in the chain before saving them in its own queue.

it is a hybrid between the input and the output block since it must provide the data to the following block and at the same time be able to retrieve them from the previous one.

Since the data managed by this category of modules is not critical from a timing point of view, the priority of its thread is set to normal and it is stopped in case an input block is ready for execution.

Output Block

This block is associated with the aim of the device, that is the actuation of in local or the dispatching of the collected data to the network it is connected to.

As the middle block its thread is associated with a normal level of priority.

3.1.2 Orchestrator

This is the main block of the firmware and it is the only piece of code to by modified by the programmer to build the wanted structure.

The structure of the orchestrator could be divided in 4 main slices:

- Block instantiation: this create the basic structure of the task the devices must execute
- Block connection: all the blocks can be connected in any way, but this link is done from the destination blocks to the source ones.
- Thread instantiation: all the function the blocks have to execute are instantiated and configurated and linked to them
- Blocks start up: the blocks are started and the device begin to work properly.

Before the main returns, this main function presents a join function that stop its execution until a signal from an ended thread wake it up (that is not possible since the threads are implemented in infinite loops).

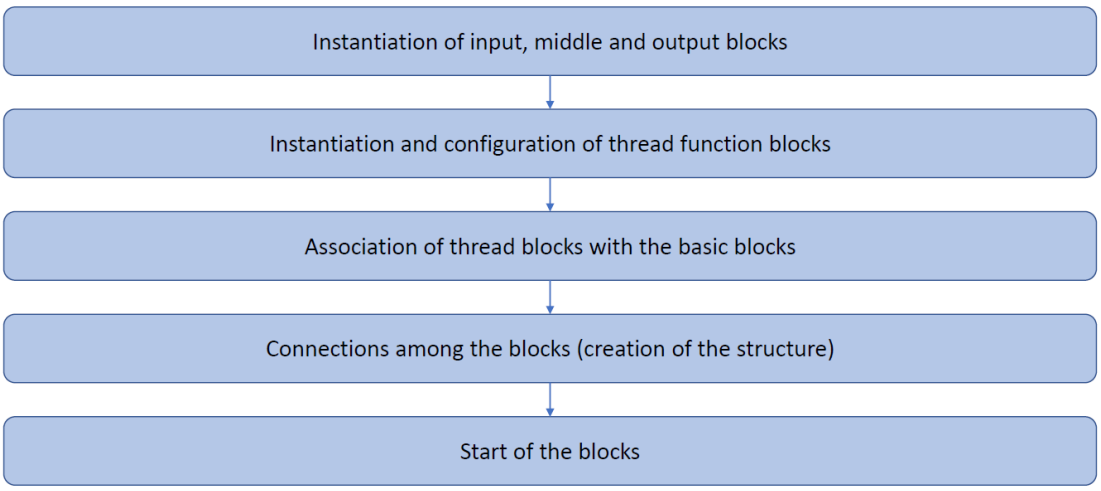


Figure 3.4: Steps to create the firmware

3.1.3 Thread Implementation

In order to define a standard in the declarations, the threads are implemented as derived classes from an abstract parent one. Besides the pivotal method of the thread, this abstract class exhibit the methods to link the thread classes to the different blocks and the elaboration function used by the middle block. This association allows the threads to communicate with the block it is associated to. This relation, consent the thread the use of the method declared in the general block structure such as insert the data in the queue or check whether there is a data available in previous block in the chain and retrieve them. The threads are implemented with an infinite loop that present at the end a wait command to return the control to the thread manager of the RTOS. By default, this wait time is set to 0 ms in the output and middle blocks, that means that they are executed continuously. This time implementation consents the user to run the block task at its maximum speed. However, the instantiation of more blocks could lead to a worsening of efficiency, since the threads have the same priority and a scheduler implemented following the round robin algorithm would make numerous and expensive context switches. Therefore, the developer has the possibility to set a higher value of waiting time for both middle and output blocks. The advantages from the higher value are both the increasing speed of execution of the tasks, which are not slowed by the continuous context switch, and a possibility to move the device in a low power mode. This decision is taken by the RTOS, which can switch off the device peripherals and slow down the clock frequency if no threads are ready to be executed. Save energy is an important feature to an IoT device since often they are battery-supplied.[6][9]

3.2 System Configuration

The interactions among blocks are designed following a subscribe scheme. The modules can be compared to nodes of a network where the consumer blocks subscribe to the producer ones and each time a new data is available the latter warns the former. Once the user provides the information on the number of blocks and their configuration, the firmware execution can be divided in steps:

- The firmware at first create the different blocks. The memory needed for the whole system is instantiated based on the parameter set by the user or on the default value. Therefore, knowing the number of blocks that the user wants to instantiate, the system allocates the memory required, reducing the resource occupation to the minimum.
- The second step is represented by the thread instantiation and configuration (e.g. set the topic of the mqtt block, set the sampling time etc.)
- The last step before the start of the program is the linking of the various block, creating a chain from the input to the output, and associating them to its corresponding thread (e.g. I2C, SPI, UART thread for input blocks, average for the middle ones etc.). During this process the blocks exchange their ID pointer in order to recognize each other in the running phase of the program. For each subscription, the producer blocks increment the number of destinations it has to warn for incoming data.

Once the threads are active, they start running continuously. If the user configured the system with waiting times long enough (i.e. no need of fast operations), the threads can be found all in the waiting state and the system can decide to go in a sleep mode to save energy. During the running state, the blocks behave in three ways:

- Input blocks: they sample the data and warn all the blocks connected to them (known thanks to the ID pointer previously stored) when a new data is ready.
- Output blocks: their thread ask periodically to the basic block if there is any data ready from the block they are associated with. In case no data are available, the thread enters in the wait state, otherwise the basic block retrieves the correct data known its correct position in the source queue thanks to a variable associated with that block as shown in the figure 3.4.
- Middle block: These blocks are an hybrid from both the input and the output blocks. Therefore their associated threads operate as output ones to retrieve the data and as input ones to warn the following blocks.

In case of a malfunction, the queue could be filled completely, and this would lead to the dropping of new data.

3.3 Main Frameworks Features

The framework has to be optimized for two main features, which are the genericity of the possible operations and the high efficiency obtained by the low computational cost of the operations.

Generality A system is considered generic when it is able to adapt all its functions to any kind of data type it interacts with such as real numbers, integer value and so on. There are many different ways to make a system generic. The main analyzed approach are:

- Polymorphism: this approach involves the implementation of several function for each new data type allowing the software to understand run-time which of them use. The advantage is the simplicity of the system, the drawbacks are: the overload of the system which has to create a virtual table for each class and the low compatibility with custom data type since it is necessary to implement a new function to each of them
- Side variable: this method is implemented with the addition on the data structure of a new variable which indicates the type of the data. This allows an easy implementation of new types but increase the memory usage and generic function can still be incompatible with custom types.
- Template: This is the more complex structure but the one with more advantages. The templates are classes or methods implemented specifically for the generic programming. The template functions are not written for specific type but are solved during compile time analyzing the type passed in the code. They are, therefore, implemented in a complete static way without any overload of the system run-time. With their usage, any user custom type is perfectly compatible with any function of the standard blocks(if they make sense).

Based on this analysis, the framework has been implemented exploiting the template structure.

Code Lightness The main obstacles in the efficiency of a firmware design are due to the minimization of the overload caused by some functions. These aspects are important in normal computer application but can result essential in embedded system devices given the constrain due the low resources it has.

As mentioned above, the template are ad-hoc classes since they are written in the code by the compiler by replacing the types indicated during their instances, thus avoiding any kind of run-time operation.

One of the main operations that influence the lightweight of the code is the instantiation of dynamic memory. The dynamic memory brings four main problems:

- *Memory leak*: the program cannot present any error in the memory management, since a leak can lead in short time to a total usage of the memory(it is usually small in size)

- *Handling memory condition*: each time a new block of memory is required, the system has to check the available memory and if there is not any free remaining space, the system has to be reset or has to rise an exception blocking the execution of the program

- *Fragmentation*: in case of multiple different sized memory requests, it will result in fragmentation leading to a possible waste of memory

- *Time*: this is closely connected to the memory fragmentation. The more the memory is fragmented, the slower new memory allocation will be. This issue could lead to an overall slowdown of the system. For these reason in the whole project the dynamic memory request during the execution of the program are avoided. Another important improvement is achieved with the implementation of a non-FIFO queue. This gives the possibility to the output block of picking up the exact data without waiting that every other block has retrieved the last data. For example, if a consumer blocks has already taken from the queue the n-value, it doesn't have to wait for the other blocks

subscrived to the same producer before reading the n-1 value. This allows even the removal of the mutex locking during the reading and the writing of the data from/to the queue, since it is not possible to write and read at the same time the same element of the queue. These could in fact slow down the system since it would block the entire queue.

Chapter 4

Block Implementation

This chapter will describe some of the most interesting blocks implemented to create a first prototype of a working IoT node.

4.1 Configuration Blocks

The configuration blocks do not belong to the sampling-elaboration-utilization chain of data and therefore do not present the structure described in the previous chapter. They are simple classes that aid the functioning of the blocks or set some property or configuration parameter of the device.

4.1.1 Ethernet Block

This block aims to configure the network setting of the device and return the pointer to object of the created connection. This pointer could be used from any other block that need a connection with the network. The pointer, if the connection results unreachable, tries periodically to establish a new one until it receive a positive response.

During its instantiation, if no parameters are added, the Ethernet block enables by default the DHCP on the board and will connect automatically. In case of necessity,it is possible to configure statically through the IP, the Netmask and the Gateway addresses.

4.2 Input Blocks

These blocks are the only timing critical ones. For this reason, their associated thread has to be as lightweight as possible, in order to obetain a higher sampling time (necessary especially when the number of sampling blocks start to grow).

To optimize the accuracy of the sampling time, the input blocks should implement a method to measure the execution time of the thread loop and subtract it to the set sampling time to obtain the resulting measuring period. This is necessary in the Mbed system since this soft RTOS exploit the ”wait” method to temporize each thread without considering its execution time that would be added to each loop causing a drift in time.

Among the already implemented input blocks, the most important are:

4.2.1 Inter Integrated Circuit (I2C) Block

Since the devices working with the I2C protocol are many, the idea is to implement a general I2C block which inherits all the characteristics of the generic input block. This combined with all the functions suitable to manage an I2C communication, are derived by child blocks. In addition, the block sets a static mutex that sets the maximum user of the bus to a single child block at a time, thus not creating conflicts in case two blocks try to sample simultaneously.[10]

The child to be implemented are the thread, which can exploit the existing libraries already present in the Mbed community and associate them to the method of the General block. In addition, it is possible to use the functions defined in the parent block to create new ones. This way, the creation of new blocks can be straightforward and the developer can wrap already working library (fast and reliable).

In the case presented in this thesis, the child thread is associated with the X-NUCLEO-IKS01A1 hardware.

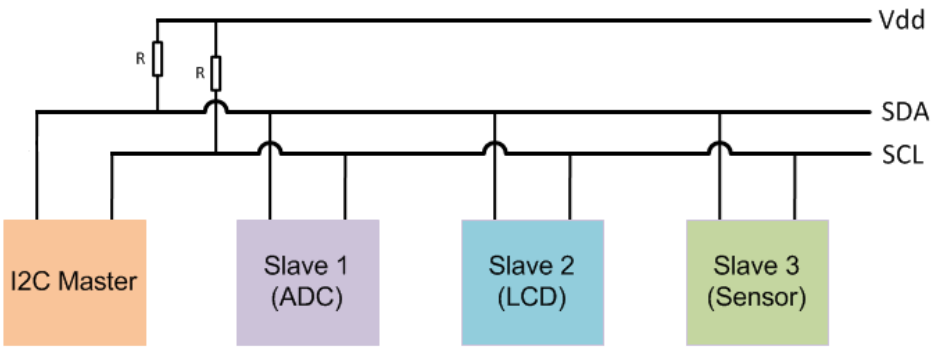


Figure 4.1: Generic I2C structure

X-NUCLEO-IKS01A1 Block

This board can measure: temperature, acceleration, rotational motion, humidity, pressure and magnetism.

The implemented block gives the possibility to choose the environmental variable to be measured locking the bus at each measurement. This influence the maximum sampling speed since more are the measurement, lower is the sampling frequency. The implementation exploits an opensource library designed for this hardware.

4.2.2 Serial Peripheral Interface (SPI) Block

Alike the I2C blocks, its implementation is composed by a general class structure that implements all the function of the SPI library wrapping them to the usage in the general block.[11]

The SPI structure is composed by three common signals: MISO, MOSI and clock. However, unlike the I2C, each instantiated block should not interact with similar ones since more devices can be grouped in the same thread and so a mutex is not necessary. Nevertheless, to simplify the addition of new devices (in the possible eventuality of a modification of the overall structure) a mutex is still implemented, associated with the possibility to connect the common signal of different blocks,

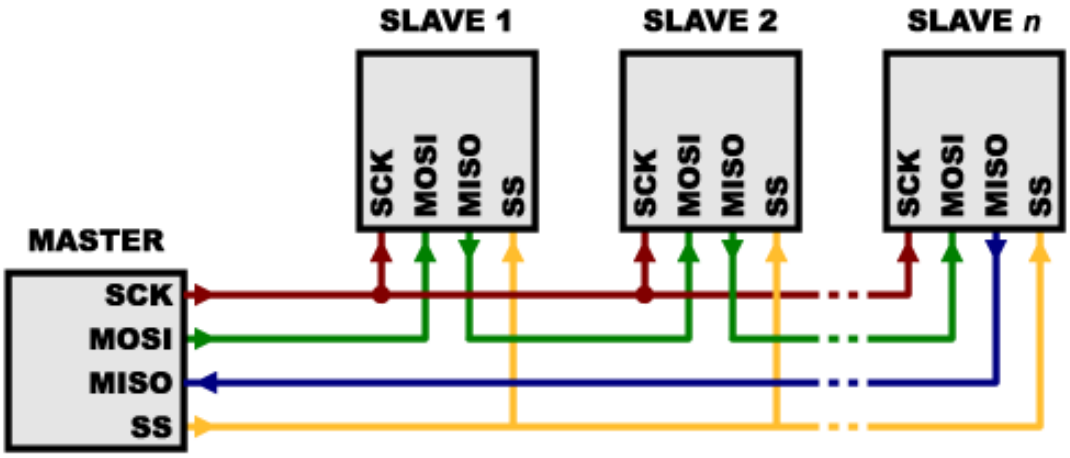


Figure 4.2: Generic SPI structure

thus allowing the addition of new devices interfaced with already present ones.

4.2.3 Universal Asynchronous Receiver-Transmitter (UART) Block

This is a simple block that can receive commands through the serial port. It is used generally for testing purpose.

It is possible to associate this thread to the output block but since the common ”printf” operation has the same behavior it is not recommended.

4.3 Middle Blocks

Besides the thread function, this block presents the implementation of the virtual function defined in the parent abstract block.

The most important block implemented are:

4.3.1 Avarage Block

This block works through a static variable that each time a data is retrieved is incremented with that value. Once the addition has been executed a number of

time equal to a configurable parameter, the average is computed and inserted in its block queue. Then, the counter is reset and the sum is set equal to 0.

It is also possible to execute a weighted arithmetic mean through the usage of a data structure composed by the data and its weight.

4.3.2 Filtering Block

This block allows the usage of part of the data coming from an already instanced block. This is useful when it is necessary to split the same data to two or more blocks but some values are superfluous or when it is possible to reuse the sampled data, thus avoiding the instantiation of another input blocks (that having an higher priority could slow down others sampling). A common use case is to receive multiple parameter measures from a single sensor that have to be processed differently. With a middle block is possible to split these measures to different output blocks.

It is possible to combine any number of middle blocks to obtain more complex elaborations.

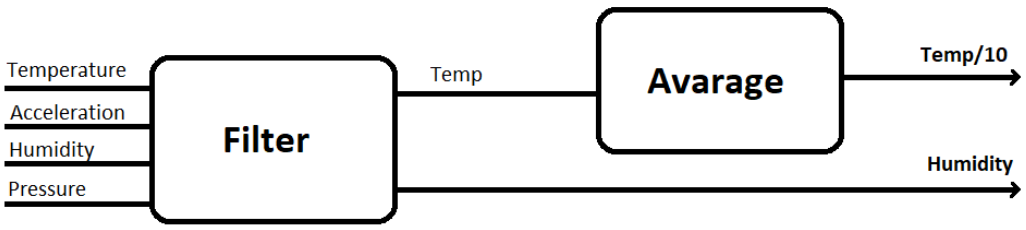


Figure 4.3: Combination of two middle blocks in cascade

4.4 Output Blocks

The output blocks can be subdivided in two categories: local actuations or dispatching of the data.

4.4.1 Serial Monitor Block

Like the UART block described above, this is a block built mainly for test purpose.

Each time a new data is received, it is written on the serial port. If the data has a custom structure a single letter is written as acknowledge

4.4.2 General Purpose I/O (GPIO) Block

This block can control an output GPIO based on a value retrieved from producer or middle blocks.

The implementation allows the user choose any GPIO available and the value or message to be interpreted as enable or disenable for the output. With the inclusion of the Mbed library, it is possible to drive the LED present in the board using as GPIO name the ”ledx” variable, where x is the number of the LED.

4.4.3 MQTT Block

This is the block used to dispatch the information over the network as a publisher client device. For this reason, it exploits the Ethernet block described above. The MQTT commands are implemented with the inclusion of the Eclipse Paho-MQTT client library.

The parameter that can be set are:

- the QoS level(by default is set to level 0)
- the topic where the message has to be published
- the length of the message (fixed by default but modifiable in case of shorter message to save some memory
- the client ID

- the hostname of the broker
- the port where the message has to be send (set to 1883 by default since it is the usual port used in the MQTT protocol)

The structure of the block is divided in several parts: a first one where the data is recovered and the creation of the connection to the broker is established and activated, a preparation of this data represented by a customizable function able to convert the data if it must be modified to make it compliant with different protocols used by the broker (such as creating a JSON file that are very common given their lightness). It is possible to use this function even to minimize the number of connection with the broker. In fact, setting a specific paramether, the grouping of consecutive data (before the dispatching phase) can be enabled. Obviously, if not necessary, simply leaving the grouping parameter empty disable this feature. At last, the payload of the message to be sent is created and sent according to the configured QoS level.

Once the message has been sent, the communication channel is closed to be recreated later. This because the messages to be sent are generally not continuous and in this way it is possible to avoid sending the KeepAlive message that would slow down the whole system and prevent a possible sleep of the device. The connection is implemented so that if this is not available, the block would continue to try to re-establish it via the ethernet block functions with a periodic loop set to one second.

From different measures performed on how the QoS can influence the timing it was noticed how, compared with the QoS 0 level dispatching time, the QoS 1 takes about 150% of time and QoS 2 takes almost 235%.

Chapter 5

Development Environment

The whole project is developed on the Eclipse integrated development environment (IDE), a free software created by the Eclipse Foundation, a not-for-profit corporation formed by a consortium of several software vendors such as IBM, HP and Intel.

The creation of environment able to operates with the project for the Mbed system needs several steps[7]:

- Installation of the Mbed Command Line tool (Mbed-cli) which enable the download from GitHub of the basic packages to create, modify and update the Mbed project
- Opening of the project in the IDE and adapt the default settings to the

new environment, and therefore setting the Mbed compiler and the toolchain (which is the GCC_ ARM).

- Set the target of the project according to the board chosen to run the firmware. The board, obviously, must be Mbed compatible.
- Since the Mbed compilation is done with the gnu++98 C++ standard language, to compile a project implemented with the framework proposed within this thesis project, it is necessary to change it to gnu++11.

The targets used to test the different firmware are the Nucleo-F767ZI and the NUCLEO-F401RE

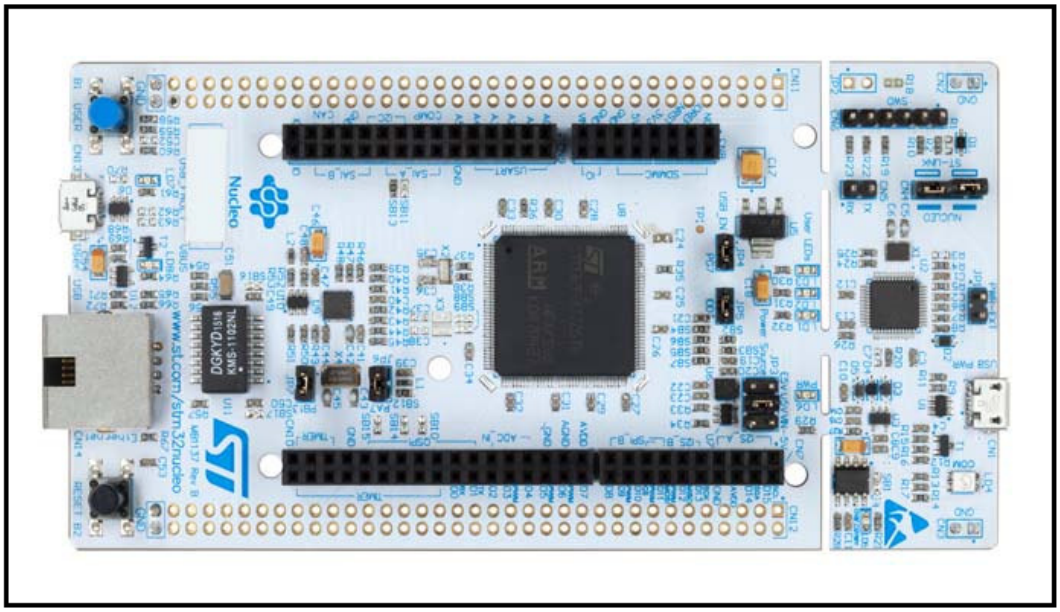
5.1 Nucleo-F767ZI

It belongs to the 32-bit microcontroller integrated circuit STM32 family by STMicroelectronics. It is based on an ARM Cortex-M7F CPU core and can reach a maximum frequency of 216 MHz.

The flash memory size is 2MB and the SRAM one is 512KB.

It is a development board and so presents many peripherals and interfaces to the external world. The most important are:

- A series of 144 GPIO. Part of them are connected to a ZIO connector that make the board compatible with Arduino and its shields
- Four I2C channels
- Six SPI channels
- Three CAN interfeces
- Three 12-bit ADCs with 24 channels
- Two 12-bit DAC
- An Ethernet interface
- A UART/USART interface



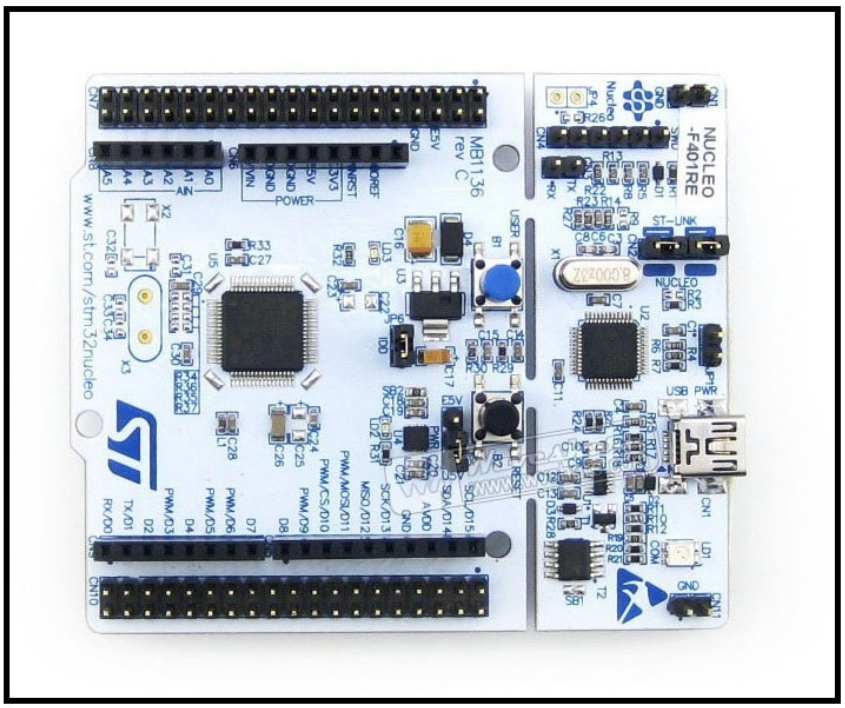
5.2 Nucleo-F401RE

It belongs to the STM32 F4-series, the first group of STM32 microcontrollers based on the ARM Cortex-M4F core and the first series to have DSP and floating-point instructions. The max CPU frequency it can reach is 84 MHz and the size of the memories are 512 KB for the flash one and 96 KB for the SRAM.

The interface toward the external environment are:

- 50 GPIOs, part of them implemented with the ArduinoTM extension connectors allowing the compatibility with many add-ons, another part with STMicroelectronics Morpho extension pin headers for full access to all STM32 I/Os.
- Three SPI channels
- Three I2C channels
- A 12-bit ADC with 16 channels
- Four UART/USART interfaces

This board has been used to simulate the different implementation even in less powerful environment of the one implemented with the NUCLEO-F767ZI.



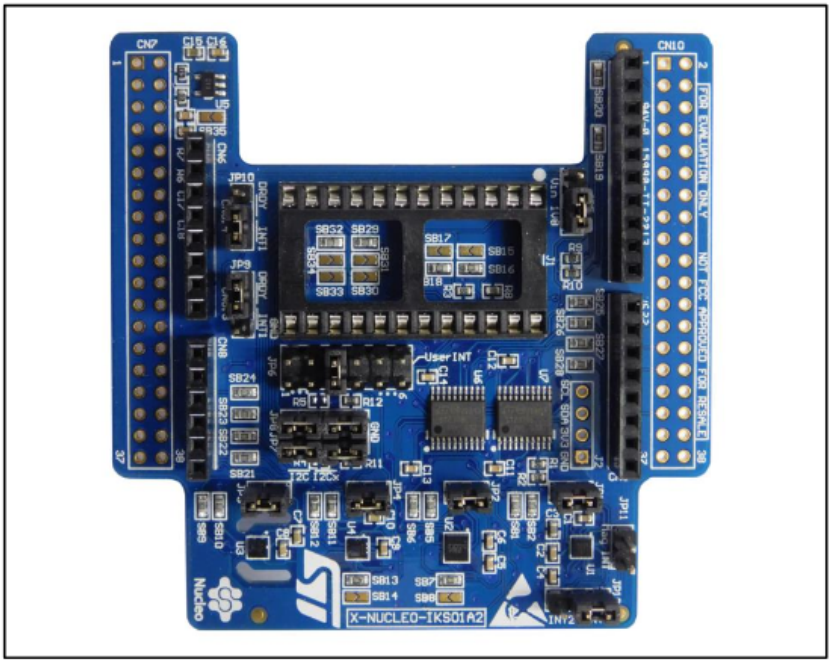
5.3 X-Nucleo-IKS01A1

The X-NUCLEO-IKS01A1 is a motion MEMS and environmental sensor evaluation board system. Its sensors are based on the I2C protocol and its pinout make it compatible with Arduino and most of the STM32 boards.

The sensor it presents are:

- LSM6DS0: MEMS 3D accelerometer ($\pm 2/\pm 4/\pm 8$) g + 3D gyroscope ($\pm 245/\pm 500/\pm 2000$)dps
- LIS3MDL: MEMS 3D magnetometer ($\pm 4/\pm 8/\pm 12/\pm 16$)gauss

- LPS25HB: MEMS pressure sensor, 260 - 1260 hPa absolute digital output barometer
- HTS221: capacitive digital relative humidity and temperature



Chapter 6

Testing Implementation and Results

This chapter describes and analyze the environment set up to test the IoT nodes realized through the use of this framework. This analysis will focus on various factors such as:

- the failure rate, that is the amount of failure (i.e. unexpected behavior) in a given period of time
- the efficiency, indicated by the execution time of the various blocks
- the complexity of the system that would lead to a possible lower sampling frequency

The MQTT block has been tested using mosquitto, a lightweight open source message broker, installed on a Linux system. With mosquito, it is possible to create a MQTT broker and a subscriber client on the same machines.

Where possible, the test firmware have been run on both boards NUCLEO-F401RE and NUCLEO-F767ZI without variation, in the other cases instead the tests are made with slight modifications.

6.1 Complex System Implementation

The first test made simulates a complex system in order to stress the hardware and analyze the behavior among a high number of instantiated blocks. The used blocks are:

- three simple timers: their execution has little impact on the system since the measures are done locally in the hardware
- four X-NUCLEO-IKS01A1 input blocks: set with different sampling time (all above 500 ms) and configured since two of them retrieve all the possible environmental values and the other two retrieves only the temperature and the humidity
- two average blocks: connected to the two different type of I2C, they execute the average of ten consecutive measures
- three MQTT blocks: set to different topic
- two LED block: which turn on the LEDs on the board based on the values on the average computed from the previous blocks

All blocks are connected with multiple connection, that means output blocks retrieve data from more than one single source. The correct functioning of the implementation has been tested through a side program running on the PC working as a broker. This program reads the data arrived on the subscriber MQTT client and compare them (which are associated with a known ID) with expected values. When a value is incorrect, this is written in a file opened in append mode. The correctness behavior of this program can be checked simply disconnecting the board from the network, so that once the queues are completely filled, it starts to drop the latest data sampled resulting on an unexpected measurement.

This test was run without interruption for two days.

Since the NUCLEO-F401RE board does not have any means to connect to the network, the MQTT blocks are been substitute with a combination of middle blocks and serial monitor blocks. The middle blocks were used to filter the data since the

serial monitor require time to execute its operation and this could have slowed down the system too much.

The results obtained highlight the correct functioning of the implementation since no errors were found but analyzing the timing of the samples on the less powerful board it was possible to notice a slight drift due to the serial monitor utilization.

6.2 Environment Measure System

The second test is done to test the efficiency of the blocks (i.e. their computational cost) and the complexity of the code associated to a real use case of an IoT node. The implemented system is composed from:

- 3 X-NUCLEO-IKS01A1 input blocks: two of them sample all the environment parameters with a sampling time equal to 1000 ms, the other one sample only the temperature, the humidity and the pressure.

- 1 Timing input block: it samples the time every 500 ms, it was implemented to visualize an eventual time drift on the operations since is a very fast block that execute almost instantly.

- A filter middle block: connected to one of the I2C block to select only the temperature, humidity and acceleration. In this way it is possible to compare it with the samples already filtered

- 4 MQTT output blocks: to send the data to the broker and analyze them

The block diagram is shown in the Figure 4.

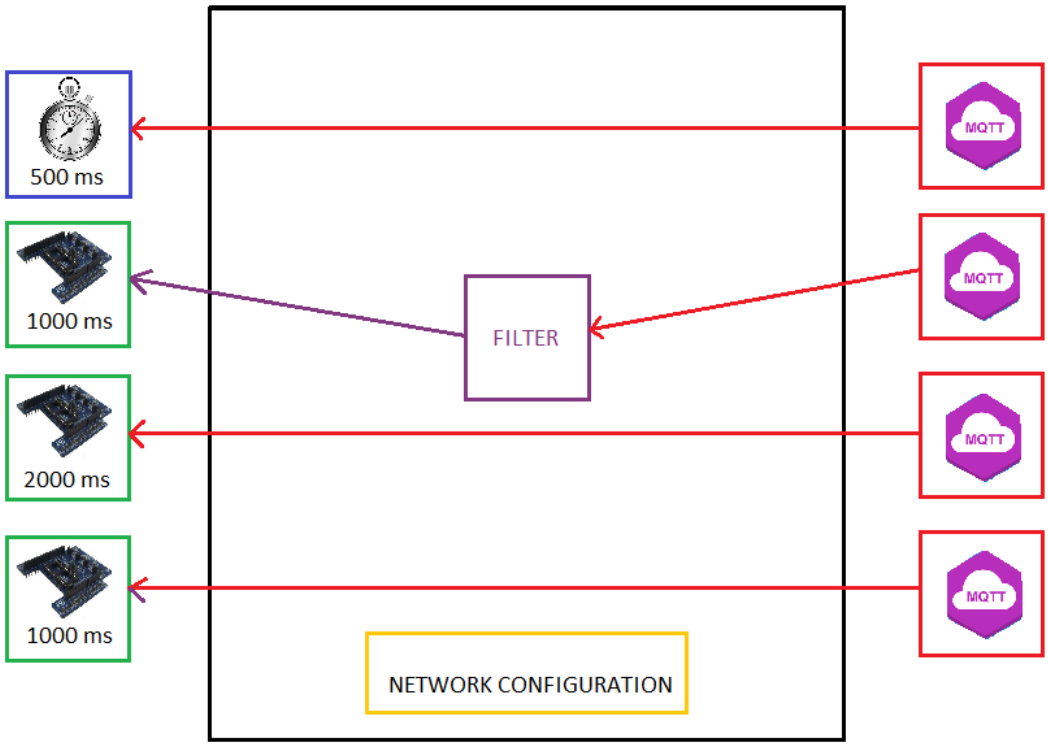


Figure 6.1: High Level Structure

The messages received from the MQTT subscriber are shown in the Figure 6.2. Here it is possible to notice how the sampling present a slight time drift, this is due the fact that the sampling time on the I2C board overlap since they are equal or have common submultiple.

As done in the previous test, the firmware is upload on both the available boards and for the NUCLEO-F401RE, the MQTT blocks are replaced with the serial monitor ones.

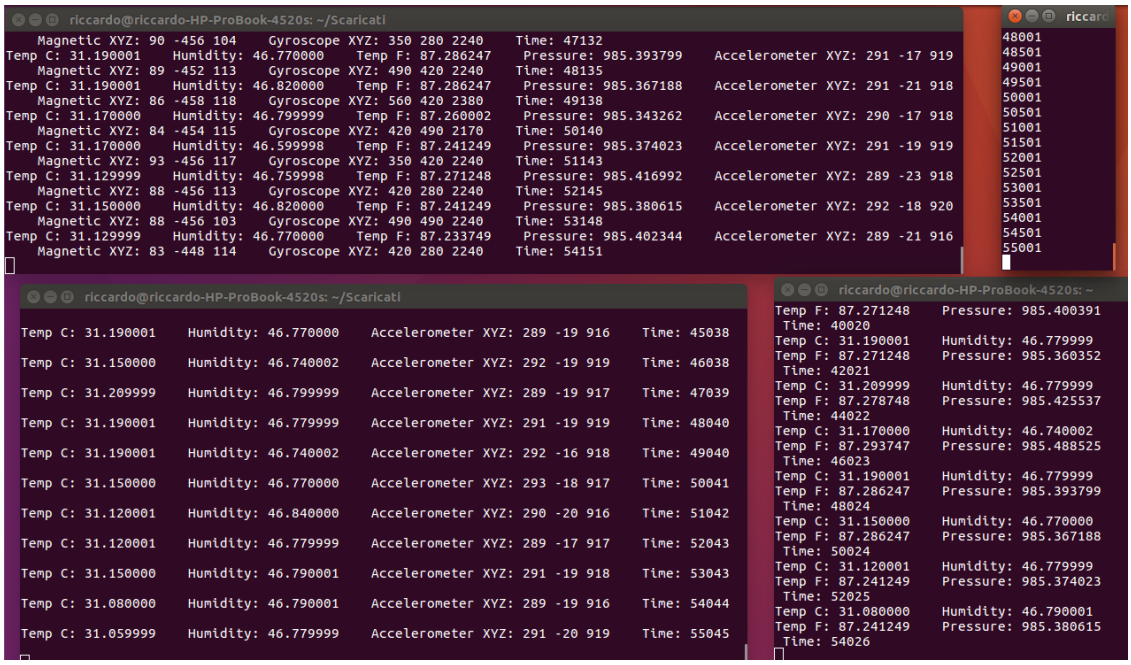


Figure 6.2: Messages received by the MQTT client subscriber

6.3 Results and Analysis

In order to test the efficiency of the framework, the computational cost of the code of the MQTT and I2C blocks have been compared with programs written ad hoc without the framework utilization, both uploaded on the NUCLEO-F767ZI board. On both cases, the times were comparable, showing how the structure of the framework slightly increase the overhead of the code execution. The only delay was due to the moving of the data among blocks, which is on the order of 10 us. To test the different use case and the impact of the MQTT block on the system, the tests were run changing the QoS levels and analyzing the time taken to complete the dispatch operation. The analyses were done on the block with the bigger payload to send (the one with all the measurement from the external I2C board) and the average time spent is shown in the table 6.1

QoS level	Completion Time
QoS 0	631 μs
QoS 1	918 μs
QoS 2	1702 μs

Table 6.1: Timing differences from QoS levels

In the table 6.2 are, instead, shown the time required to obtain the different measures from the X-NUCLEO-IKS01A1 shield.

Parameter	Time Required
Thermometer ($°C$) & Hygrometer	1826 μs
Barometer & Thermometer($°F$)	1096 μs
Magnetometer	1906 μs
Accelerometer	1908 μs
Gyroscope	1907 μs

Table 6.2: Timing differences from QoS levels

These measures show the max sampling frequency settable on the X-NUCLEO-IKS01A1 block

At last, the influence of the context switching of the middle and output blocks was measured. As presented in the previous chapter, these blocks have a default null time set as wait in the waiting state. Therefore, if output blocks which takes 3 ms to complete their threads, wait only 0.2 ms to return in the ready state, then the RTOS could swap the thread execution before the end of the previous one since they have the same priority. This means the CPU has to sustain a high number of context switches which slow down the overall system. The graph 6.3 shows the impact of a growing number of MQTT output blocks to the system where the wait time to each of those blocks is set to 0.2 ms.

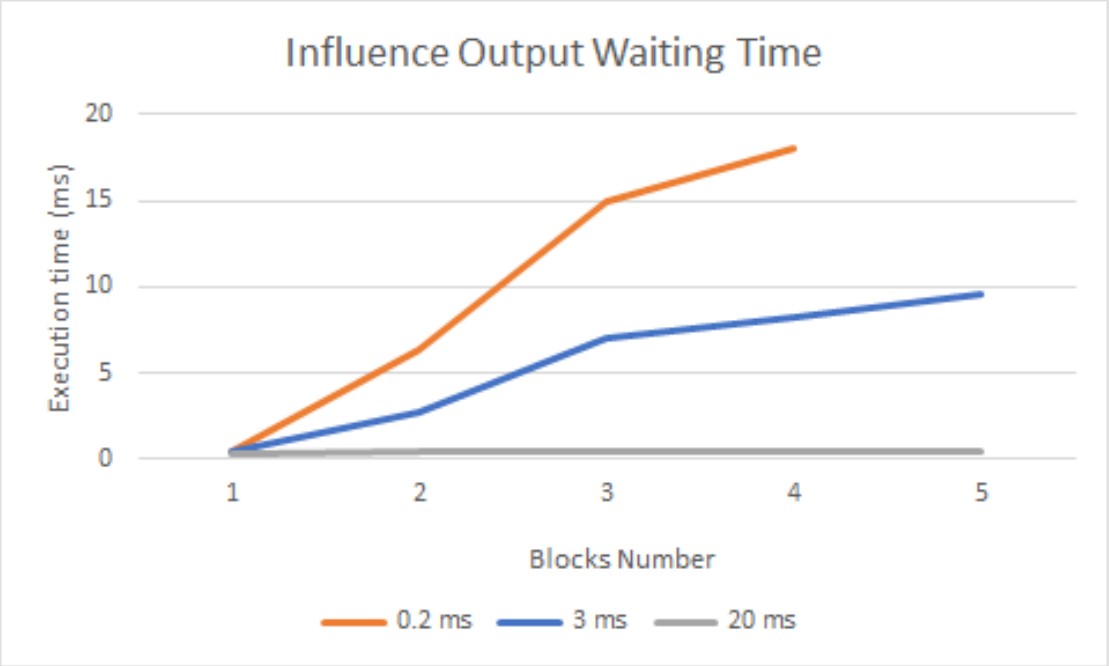


Figure 6.3: Waiting time influence

As shown in the graph, with lower waiting time between the execution, the blocks slow each other, leading, in some cases, to a system malfunctioning. In fact, setting the waiting time to 0.2 ms works only with the instantiation of a maximum of four blocks. With a higher number of blocks, their execution could slow down bringing some block to complete its operation in over 300 ms. With the increasing of the delay between the execution, the blocks work without interruption and their execution is optimized. However, set a too high waiting time would lead to a starvation on the retrieving data from the input queue, causing data loss.

6.4 Code Sample

Here it is reported the code of the orchestrator corresponding to the described structure on the environment measures implementation.

As shown, the code is simple and linear, and makes evident the whole structure of the system just looking on the implementation of the blocks and their connections. The functions required to configure the basic blocks and the threads are

separated from the constructor, even if it is possible to implement them inside it. This would allow a future graphic implementation where will be simpler to associate parameters with the members of the classes.

```
Thread *block1; //thread to block the orchestrator execution
/*-----Configuration network----------*/
EthernetConnection* NetworkConfig=new EthernetConnection();
NetworkConfig->ConfigNetwork("192.168.1.200","255.255.255.0", "
    192.168.1.1");

/*------ Blocks instances ----------*/
InputBlock<int>* intMqttInp= new InputBlock<int>(10);
InputBlock<Nucleo_Ikso1a1Data>* I2cAll= new InputBlock<
    Nucleo_Ikso1a1Data>(15);
InputBlock<Nucleo_Ikso1a1Data>* I2cAll2= new InputBlock<
    Nucleo_Ikso1a1Data>(15);
InputBlock<Nucleo_Ikso1a1Data>* I2cTHP= new InputBlock<
    Nucleo_Ikso1a1Data>(15);
MiddleBlock<int>* MiddleFilter= new MiddleBlock<int>();
OutputBlock<Nucleo_Ikso1a1Data>* OutMqttI2cTHP= new OutputBlock<
    Nucleo_Ikso1a1Data>();
OutputBlock<Nucleo_Ikso1a1Data>* OutMqttI2c2= new OutputBlock<
    Nucleo_Ikso1a1Data>();
OutputBlock<Nucleo_Ikso1a1Data>* OutMqttI2c= new OutputBlock<
    Nucleo_Ikso1a1Data>();
OutputBlock<int>* MqttIntout= new OutputBlock<int>();

/*------ Threads instances and configurations ----------*/
TestInput2<int>* Inp_timer= new TestInput2<int>();
Inp_timer->setSamplingTime(500);

XNucleo_Ikso1a1<Nucleo_Ikso1a1Data>* I2cNucleo= new XNucleo_Ikso1a1<
    Nucleo_Ikso1a1Data>(nullptr,D14, D15);
I2cNucleo->setSamplingTime(1000);
I2cNucleo->ActiveSensor(1,1,1,1,1,1);

XNucleo_Ikso1a1<Nucleo_Ikso1a1Data>* I2cNucleo2= new XNucleo_Ikso1a1<
    Nucleo_Ikso1a1Data>(nullptr,D14, D15);
```

```
I2cNucleo2->setSamplingTime(1000);
I2cNucleo2->ActiveSensor(1,1,1,1,1,1);

XNucleo_Ikso1a1<Nucleo_Ikso1a1Data>* I2cNucleoTHP= new XNucleo_Ikso1a1
    <Nucleo_Ikso1a1Data>(nullptr,D14, D15);
I2cNucleoTHP->setSamplingTime(2000);
I2cNucleoTHP->ActiveSensor(1,1,0,0,0,1);

Filter_T_H_A<Nucleo_Ikso1a1Data>* Filter= new Filter_T_H_A<
    Nucleo_Ikso1a1Data>();

MqttBlock<int>* MqttintThread= new MqttBlock<int>();
MqttintThread->setEth(*NetworkConfig);
MqttintThread->setClientId("F767ZI");
MqttintThread->setHostname("192.168.1.1");
MqttintThread->setTopic("time");

MqttBlock<Nucleo_Ikso1a1Data>* MqttThreadI2c= new MqttBlock<
    Nucleo_Ikso1a1Data>();
MqttThreadI2c->setEth(*NetworkConfig);
MqttThreadI2c->setClientId("F767ZI");
MqttThreadI2c->setHostname("192.168.1.1");
MqttThreadI2c->setMqttMessageLength(500);
MqttThreadI2c->setTopic("i2c_all");

MqttBlock<Nucleo_Ikso1a1Data>* MqttThreadI2c2= new MqttBlock<
    Nucleo_Ikso1a1Data>();
MqttThreadI2c2->setEth(*NetworkConfig);
MqttThreadI2c2->setClientId("F767ZI");
MqttThreadI2c2->setHostname("192.168.1.1");
MqttThreadI2c2->setTopic("filter");

MqttBlock<Nucleo_Ikso1a1Data>* MqttThreadI2cTHP= new MqttBlock<
    Nucleo_Ikso1a1Data>();

MqttThreadI2cTHP->setEth(*NetworkConfig);
MqttThreadI2cTHP->setClientId("F767ZI");
MqttThreadI2cTHP->setHostname("192.168.1.1");
MqttThreadI2cTHP->setMqttMessageLength(300);
```

```
MqttThreadI2cTHP->setTopic("i2cTHP");

/*------Connections----------*/
MqttIntout->SetSource(*intMqttInp);
OutMqttI2c->SetSource(*I2cAll);
OutMqttI2c2->SetSource(*MiddleFilter);
OutMqttI2cTHP->SetSource(*I2cTHP);
MiddleFilter->SetSource(*I2cAll2);

/*------Start blocks--------*/
block1=intMqttInp->StartBlock(Inp_timer);
MqttIntout->StartBlock(MqttintThread);

I2cAll->StartBlock(I2cNucleo);
OutMqttI2c->StartBlock(MqttThreadI2c);

MiddleFilter->StartBlock(Filter);

I2cAll2->StartBlock(I2cNucleo2);
  OutMqttI2c2->StartBlock(MqttThreadI2c2);

I2cTHP->StartBlock(I2cNucleoTHP);
 OutMqttI2cTHP->StartBlock(MqttThreadI2cTHP);

/**------Stop the main process-----*/
block1->join();
```

Chapter 7

Conclusions

In this thesis project, a framework system to simplify the creation of an IoT node firmware has been presented.

Since the implementation is most likely implemented without any dependencies from the RTOS in which the framework run, the first part of the thesis describes the most important feature to choose a suitable RTOS. Then, the second part, presents the implementation of the framework.

The primary goal was to implement a system able to speed up the creation of a proof of concept and a structure to be used as simulator for a test environment for Industry 4.0. At the same time the framework sets the stage for future development of any kind of Cyber Physical Structure.

The critical point for a designer is to write a firmware that is not only working properly but that can be easily modifiable and upgradeable since the different protocols (especially security and communication ones) and structures are in continuous evolution.

The key point was the implementation of a generic structure subdivided in independent modules easily instantiable. With this approach, at the cost of a minimal overhead on the computational execution, the following advantages are obtained:

- The blocks portfolio can easily grow with time just adding new functional modules.

- The independency among any function implemented in blocks consent to an easy modifiability of the system.
- High portability of the project in different hardware and systems.
- High code legibility (leading to a clear organization of the code).

Furthermore, the framework is implemented mostly using standard C++ libraries, making it easily interchangeable with any RTOS available on the market.

7.1 Future Developments

As mentioned in the previous chapters, the framework implemented is the base on which can be built a more complex and complete system. This improvement can be reached implementing several new features:

- **Implementation of new thread blocks**: as obvious, the number of thread implementable on this framework is illimitated. The structure is done in the way that each blocks can be derived from already existing library, thus simplifying and speeding up the implementation of new blocks just copying working firmware with minimum variation to adapt it to the blocks. In this way it is possible to implement the framework in system already on work without re-projecting all the code. The blocks implementable are various, from more simpler functions to make the device reachable and controllable from an external system (e.g. an input MQTT subscriber client), to complex one such middle blocks that elaborate the data through machine learning paradigms.
- **Optimization of the scheduling**: the framework is based on an RTOS with preemptive scheduling, which allows a correct behavior of the system giving a higher priority of execution to the critical blocks. However, when the blocks have the same priority (i.e. middle and output blocks), the scheduler could alternate the thread execution on the CPU slowing down the system. Greater efficiency would occur if these threads are executing without any context switching. Therefore, by analyzing the different blocks required time,

it is possible to set a suitable waiting time between sequent operation, thus optimizing the used resources.

- **Association with a graphic environment**: the final goal is to implement a graphical programming tool which allows to program different devices. This would allow anyone to be able to create an IoT node or a simple non-connected embedded system. In the figure 7.1 is shown a screen from Node-RED, a visual tool to wire together different hardware devices, APIs and online services at an higher-level language, as an example of how the system could result.

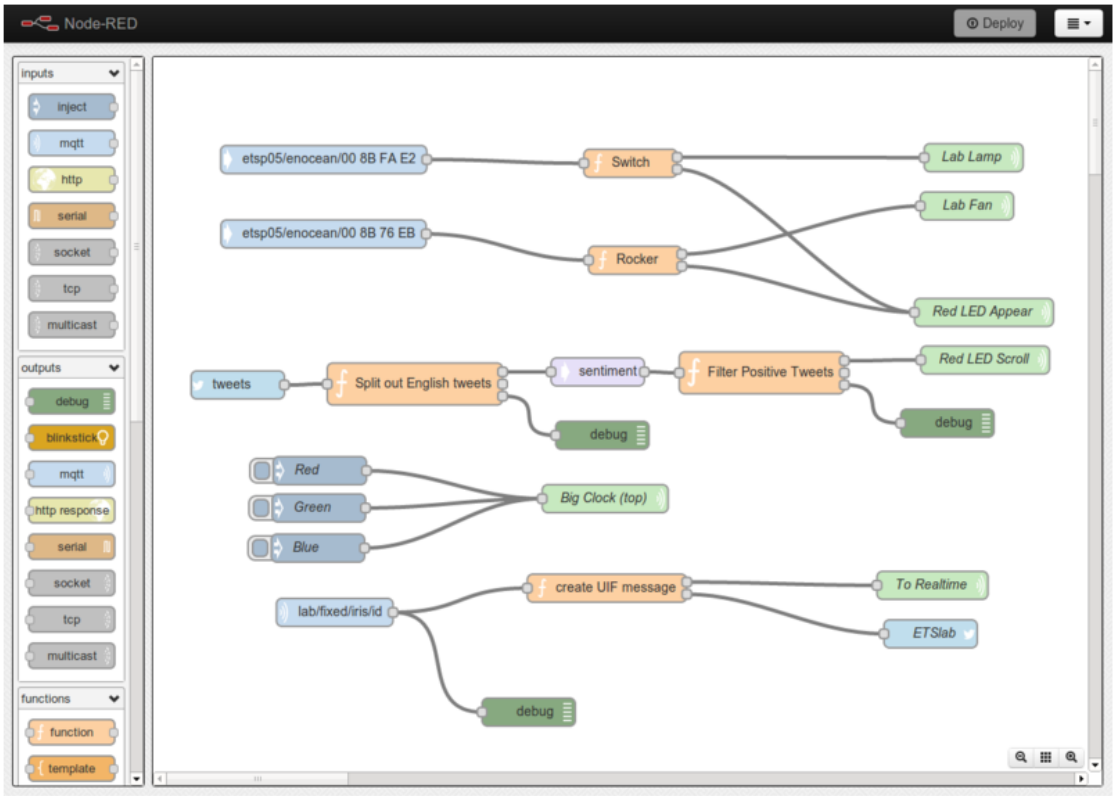


Figure 7.1: Screenshot of a Node-RED project

Bibliography

[1] Khaitan et al, *Design Techniques and Applications of Cyber Physical Systems: A Survey*, IEEE Systems Journal, 2014.

[2] Vermesan, Ovidiu; Friess, Peter, *Internet of Things: Converging Technologies for Smart Environments and Integrated Ecosystems*, Aalborg, Denmark: River Publishers,2013.

[3] Yanbing Li, M. Potkonjak, W. Wolf,*Real-time operating systems for embedded computing*, IEEE Systems Journal, 2002

[4] WIKIPEDIA Web Site,*https://en.wikibooks.org/wiki/Microprocessor_Design/Real-Time_Operating_System*

[5] ARM Mbed Web Site, *https://os.mbed.com/handbook/Homepage*

[6] OASIS Web Site, *http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html _Toc398718028*

[7] Scott Meyers,*Effective in an Embedded Environment*,2012

[8] Github Mbed Web site, *https://github.com/ARMmbed/mbed-cli*

[9] Andras K. Fekete,John Weiss, *Optimizing Performance of C++ Threading Libraries*, 2015, ResearchGate

[10] *I2C-bus specification and user manual*. Rev. 6. NXP. 2014-04-04. UM10204.

[11] *Enhanced Serial Peripheral Interface (eSPI) Interface Specification (for Client Platforms)*, Intel, May 2012, Document Number 327432-001EN. Retrieved 2017-02-05.

[12] *Enhanced Serial Peripheral Interface (eSPI) Interface Specification (for Client Platforms)*, Intel, May 2012, Document Number 327432-001EN. Retrieved 2017-02-05.