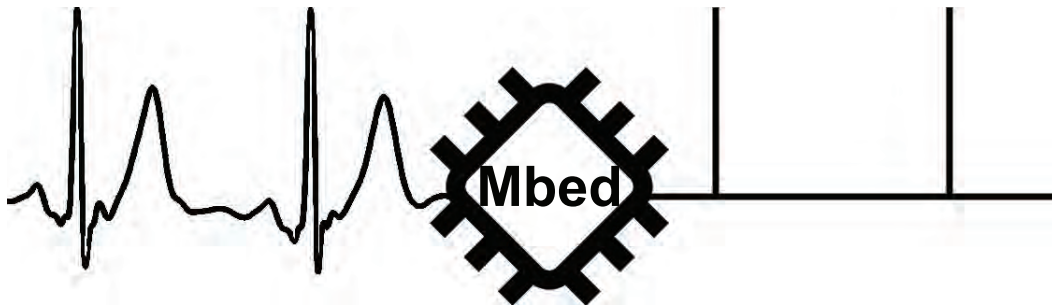


Technische Universität Kaiserslautern

Distance Study Program
Software Engineering for Embedded Systems

Master of Engineering (M. Eng.)

Connecting Simulink to ARM-Cortex Based Processors via Mbed to Support Medical IoT Applications



Provided by

[Sébastien Dupertuis](#)

Date: *September 20, 2021*
First supervisor : *Prof. Dr.Ing. habil. Peter Liggesmeyer*
Second supervisor: *Donald Barkowski*



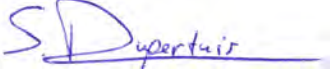
Declaration of originality

Ich versichere, dass ich diese Masterarbeit selbstständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt und die den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

I certify that I independently prepared the following Master thesis called

“Connecting Simulink to ARM-Cortex based processors
via Mbed to support Medical IoT applications”

using only the sources and aids stated, and that I have indicated as such the parts of the sources used literally or in substance.



Bern, on September 20th, 2021

Sébastien Dupertuis

Abstract

Connecting Simulink to ARM-Cortex based processors via Mbed to support medical IoT applications

This Master thesis evaluates abstraction techniques at different levels of a product development workflow for an engineer to be more effective and productive to develop embedded real-time, connected and safety related applications. The context of this work is oriented towards the medical devices industry by evaluating the deployment, onto an Internet of things device, of a medical application developed in a model-based environment using two different strategies to interact with the hardware.

A common problem to address in the medical devices industry is the analysis of an electrocardiogram signal and the extraction of some of its properties, like the heart rate. A model-based design approach has been taken to design and implement a signal processing application for the analysis of such signals. After the design phase of the algorithm, automatic code generation tools are used to generate C/C++ code automatically out of the designed model for its deployment onto an embedded system. At the end of the development process, the system is composed of the three following parts: the sensors (placed on the patient) that acquire the electrocardiogram signal in real-time, the deployed algorithm onto an ARM based device and the data exchange to a remote location following the Internet of things principle.

This work has been done following two hardware deployment approaches. The first one, called bare metal, consists of connecting the algorithm directly to the peripherals at the lowest possible level in code using firmware libraries provided by the hardware vendor. The second one is using the Mbed hardware abstraction layer to make Internet of things application development easier and faster. These approaches have been evaluated against each other with regards to measurable system properties and characteristics like: ease of use, development time, tasks execution time, processor utilization and resources consumption.

After the delivery of this thesis, a next step for MathWorks is to have a direct support, through Mbed, of many ARM based microprocessors from Simulink. This would support the integration through automatic C/C++ code generation and the use of the Mbed hardware abstraction layer by extending Simulink with a hardware support package to work at a higher level of abstraction. For developers, the support of Mbed for ARM based targets will provide them with common functionality of the processors so that development and deployment can be done quickly and easily. A Mbed hardware support package will be created with the programming ability of more than 150 processors and boards through Mbed following a model-based design approach with all the abstraction methods used and explained in this thesis.

I Table of Contents

Abstract	II
I Table of contents	III
II List of Figures	V
III List of Tables	VII
1 Introduction	1
1.1 Motivation	1
1.2 Objectives	1
1.3 Context	2
2 State of the Art	3
2.1 Reference workflow	3
2.2 IEC 62304 standard	4
2.2.1 IEC 62304 metrics	4
2.3 Medical industry user story	5
3 Mbed Ecosystem	6
3.1 Operating System	7
3.2 Hardware Abstraction Layer	8
3.2.1 Mbed microcontroller library	9
3.3 Hardware	12
3.3.1 Standard pin names	12
4 Hardware System	13
4.1 Evaluation board	15
4.1.1 Main characteristics of the microprocessor	16
4.2 Extension board	16
4.2.1 Schematic of the analog path	17
4.3 Sensors	18
5 Model-Based Design	19
5.1 MBD workflow	19
5.2 Mapping to the V-model	20
5.3 Software tools used in this project	21
5.4 Competitor tools and solutions	22
5.4.1 Mbed Online Compiler	23
5.4.2 Mbed Studio and CLI	24
5.4.3 Scilab and Xcos with Mbed	24
5.4.4 Observations on competition	25

6	Application	26
6.1	Electrocardiogram basic theory	26
6.2	Electrocardiogram problem formulation	27
6.2.1	Signal processing chain	28
6.2.2	Sampling frequency	29
6.3	System parameters	30
6.4	First investigations in MATLAB	31
6.4.1	Import and convert measured data	31
6.4.2	Pre-processing stage	33
6.4.3	Post-processing stage	35
6.5	Simulation model in Simulink	37
6.5.1	Band-pass Butterworth filter	38
6.5.2	Low-pass Butterworth filter	40
6.5.3	Peaks extraction	40
6.5.4	Peaks time	41
6.5.5	Delta time	42
6.5.6	BPM final value	43
6.6	Deployment models in Simulink	45
6.6.1	Processor-In-the Loop verification	45
6.6.2	External mode	47
6.6.3	Standalone deployment	49
7	Drivers	50
7.1	Techniques to integrate C code in Simulink	51
7.2	Digital inputs and outputs	53
7.2.1	Digital read	54
7.2.2	Digital write	62
7.3	Analog-to-digital converter	65
7.3.1	ADC modes of operation	66
7.4	Communication's protocols	68
7.4.1	UDP and TCP/IP	68
7.4.2	IoT channel	70
8	Profiling	79
8.1	Static code metrics analysis	79
8.2	Dynamic execution metrics analysis	81
8.3	Discussion on profiling results	83
9	Results Analysis	84
9.1	Comparison of numerical results	84
9.2	ThingSpeak data compression	88
10	Conclusion	91
11	Annex	93
12	Bibliography	94

II List of Figures

2.1	ISO 26262 reference workflow for software development & verification processes [1] . . .	3
2.2	Highly complex MIMO system ventilator requiring total robustness and reliability [3] . . .	5
3.1	Mbed layered architecture [10]	6
3.2	AUTOSAR classic simplified layered architecture [11]	7
3.3	Overview of the Mbed software layers architecture [12]	8
3.4	Trend in programming embedded systems in C or C++ [13]	9
3.5	Mbed API for digital I/O's	10
3.6	Mbed HAL and common API for digital I/O's	11
3.7	Arduino Uno pin mapping to functionality [14]	13
4.1	Hardware system for the electrocardiogram (ECG) application	14
4.2	STM32 Nucleo-F767ZI development board [15]	15
4.3	Olimex EKG-EMG Shield [18]	16
4.4	Olimex EKG-EMG Shield signal acquisition, stage 1 [18]	17
4.5	Olimex EKG-EMG Shield variable gain, stage 2 [18]	17
4.6	Olimex EKG-EMG Shield pre-emphasis, stage 3 [18]	18
4.7	Olimex SHIELD-EKG-EMG-PRO Sensors	18
5.1	MBD Workflow [7]	19
5.2	V-model development process	20
5.3	Mapping of the MBD workflow to the V-model development process	21
5.4	Mbed Online IDE	24
5.5	Mbed support example from Scilab/Xcos for the Arduino platform [8]	24
5.6	HSP for STM32 targets from Scilab/Xcos [9]	25
6.1	Normal sinus rhythm impulse [19]	26
6.2	Einthoven's triangle [20]	27
6.3	Example of a captured ECG signal with an optical heart sensor from a smart watch . . .	27
6.4	Export of data into a CSV file	28
6.5	Signal processing chain to extract the heart rate of an ECG signal	29
6.6	Wavelet spectrogram of one of the ten recorded ECG signal	30
6.7	Bringing external CSV data into MATLAB as a table	32
6.8	Real raw ECG signal #6	33
6.9	Bessel Bode diagram of amplitude and pre-emphasized signal	33
6.10	Band-pass Butterworth Bode diagram of amplitude and rectified signal	34
6.11	Low-pass Butterworth Bode diagram of amplitude and envelope of the signal	34
6.12	Extraction of maxima without and with threshold	35
6.13	BPM results of the ECG signal #6	36
6.14	Simulation model of the signal processing algorithm in Simulink	37
6.15	Mask to set the band-pass filter's parameters	39
6.16	Amplitude response of the Butterworth band-pass filter	39
6.17	Interface to parametrize and visualize the designed low-pass filter	40
6.18	Peaks extraction subsystem based on the findpeaks() function	40
6.19	Peaks time subsystem providing the time occurrences of peaks	41

6.20	Delta time subsystem computing the Δ distance	42
6.21	Top level of the subsystem computing the BPM final value	43
6.22	Subsystem computing the BPM value	43
6.23	Subsystem computing the standard deviation	44
6.24	Library of common IP components for re-use	44
6.25	External mode model for PIL equivalence testing	45
6.26	SOS diagram with coefficients for an IIR filter of fourth order	46
6.27	Host-target communication with external mode XCP	47
6.28	Configuration parameters of the external mode using serial XCP	48
6.29	Standalone mode model for final deployment onto hardware	49
7.1	Variants management	51
7.2	C code integration techniques	52
7.3	Variant subsystems for the digital read (left-hand side) and write (right-hand side)	53
7.4	Logic controlling the red, green, and blue LEDs	54
7.5a	Digital read system object masks for the Mbed implementation	55
7.5b	Digital read system object masks for the Stm32 implementation	55
7.6a	Pin mapping of the digital inputs and outputs of the connector 7	55
7.6b	Pin mapping of the digital inputs and outputs of the connector 10	56
7.7a	Mbed implementation of the digital read system object (properties)	56
7.7b	Mbed implementation of the digital read system object (methods)	57
7.8a	Stm32 implementation of the digital read system object (properties)	59
7.8b	Stm32 implementation of the digital read system object (methods part 1)	60
7.8c	Stm32 implementation of the digital read system object (methods part 2)	61
7.9a	Digital write system object masks for the Mbed implementation	62
7.9b	Digital write system object masks for the Stm32 implementation	62
7.10a	Mbed implementation of the digital write system object (properties)	63
7.10b	Mbed implementation of the digital write system object (methods)	64
7.11	Stm32 implementation of the digital write system object (stepImpl() method)	65
7.12	ADC system object mask for the Stm32 implementation	67
7.13	ADC system object mask for the Mbed implementation	67
7.14	UDP send system object's mask	68
7.15	TCP/IP send system object's mask	69
7.16	Ethernet configuration of the embedded target within a private local network	69
7.17	ThingSpeak IoT systems [21]	70
7.18	Structure of a ThingSpeak channel	71
7.19	Mask of the BytePack subsystem packing data as one uint8 vector	72
7.20	Algorithm of the IoT subsystem sending data to ThingSpeak	72
7.21	Mask of the IoT subsystem sending data to ThingSpeak	74
7.22	Writing mechanism to the ThingSpeak's data fields in the system object	75
7.23	Ethernet configuration of the embedded target for IoT connectivity	76
7.24	ThingSpeak web interface to manage collected data	77
7.25	Raw ECG signal #6 displayed by the ThingSpeak web interface	78
8.1	Multirates within the IoT subsystem	83
9.1	Simulation vs deployment modes results for the ECG signals #1 to #5	86
9.2	Simulation vs deployment modes results for the ECG signals #6 to #10	87

9.3	Simulink model to gather Ethernet data coming from the embedded system	88
9.4	Comparison between the original and the ThingSpeak received ECG signal #6	89
9.5	Comparison between the original and the ThingSpeak ECG pulse at 11 seconds	90
9.6	BPM results of the ThingSpeak received ECG signal #6	90

III List of Tables

2.1	Metrics of the IEC 62304 standard [2]	4
4.1	Costs of the needed hardware items	14
5.1	Assessment of competitors to MATLAB and Simulink	22
6.1	Filters parameters for the signal processing chain	31
6.2	Table of measured raw ECG data in MATLAB	32
6.3	Filters coefficients of the Butterworth band-pass and low-pass IIR filters	46
6.4	STM32F76xxx advanced Arm-based 32-bit MCUs programmed baud rates [17]	49
7.1	ADC independent operation's modes	66
8.1	Comparison of static code metrics with or without the Mbed HAL	79
8.2	Comparison of dynamic execution metrics with or without the Mbed HAL	81
9.1	Comparison of numerical results among all implementations	84

1 Introduction

1.1 Motivation

Being able to quickly go from an idea down to its implementation onto a real system to have it up and running before the competition is what is driving the fast evolution of the technology nowadays. It has always been challenging to develop complex systems including both innovative products and new features. However, today's situation can change the game. Indeed, it has never been so easy to get hardware and software solutions that are available to everyone and at an affordable price, like low cost hardware such as Arduino, Raspberry Pi, Beaglebone, you name it.

Engineers developing domain specific applications, that are linked to a particular industry, are not necessary specialist in the required programming languages to develop them. Moreover, there is a need to always be more productive and deliver more complex products as soon as possible to shorten the time to market. The complexity of today's systems is at such a level that they may contain several dozens or even hundred processors that are programmed with several million lines of code. All this is driven by the requirements of customers, the safety standards, the environmental regulations and the market competition. To support the aforementioned needs, one solution is to use the *Model-Based Design* approach.

Model-based design (MBD), also called Model-driven engineering (MDE), is a modeling method that allows to handle the level of complexity of embedded systems as most of them are way too complex to be directly coded in C code for example. MBD is a model-centric approach to develop any type of dynamic systems. Out of the base product's requirements, a model is derived and serves as an executable specification that is used throughout all phases of the development process that can be sum up as: requirements, architecture, design, implementation and integration.

1.2 Objectives

The main goal of this project is to make it easier for engineers working on complex products, like in the medical devices industry, to create innovative *Internet of things* (IoT) applications using a development platform, including software and hardware, that abstracts the coding part of the algorithm and its interaction with the used hardware. This is done by using *MATLAB and Simulink* as MBD tool on the software side and off-the-shelf evaluation or demo boards on the hardware side, as well as an IoT *hardware abstraction layer* (HAL) called *Mbed* supported on both sides.

Mbed is an online and open source collaborative platform managed by ARM for engineers interested in rapid IoT device development. Mbed provides multiple resources to speed up the development of applications such as an offline and an online *integrated development environment* (IDE), a *real-time operating system* (RTOS), a HAL, and many development boards from various vendors like NXP, Renesas, STMicroelectronics and many more.

The Mbed model architecture makes its ecosystem extremely flexible and evolutionary for embedded solutions developers. Based on the aforementioned points, this Master thesis focuses on the following three main topics:

- **The application** which includes two main tasks. The first one consists of the real-time processing of an electrocardiogram (ECG) signal to extract its properties like the heart rate in beats per minute. The second one is to transmit in real-time the obtained results over the Ethernet network either locally or on the Internet for further analysis
- **The driver layer** which is the implementation of the low-level drivers having direct access to the board peripherals and also accessing them through the Mbed HAL. An objective comparison is done between both approaches based on technical and productivity related criteria
- **The system profiling** that involves the comparison of both driver approaches by analyzing specific system properties and characteristics, as well as the comparison and discussion over the numerical results provided by the application at the different stages of development

1.3 Context

To be able to efficiently analyze and discuss the pros and cons of using this Mbed HAL for IoT applications in the medical devices industry using an MBD approach, it is needed to properly define the context boundaries of a practical problem that allows its assessment.

A common problem to solve, that encompasses all the aforementioned points, is the processing of ECG signals. It requires real-time processing of data acquired by hardware sensors and the extraction of relevant signal characteristics to provide output information like the heart rate in beats per minute and its standard deviation. Therefore, this ECG problem statement serves as the basis for the implementation and evaluation of the Mbed HAL.

The context boundaries are not only defined by the software part of the application, but also by the hardware that executes it. In this project, an evaluation board from the vendor STMicroelectronics is used, as it is made of an ARM microprocessor and it supports the Mbed HAL off-the-shelf. An extension board is connected to the evaluation board to allow the acquisition of ECG data via physical sensors placed on a patient.

2 State of the Art

The main goal of this Master thesis is to make it easier and quicker for engineers working in the medical devices industry to create innovative Internet of things (IoT) applications. Therefore, it is important to understand the current state-of-the-art concepts for a model-based design (MBD) approach, and how it connects to the IEC 62304 reference standard which is very important for the medical devices industry.

2.1 Reference workflow

The reference workflow (2.1) is defined by the ISO 26262 standard, part 6 [1] from the automotive industry for software development and verification processes. It is a derivative of the IEC 61508 generic functional safety standard and is reused in other industries for functional safety. It starts, on its left-hand side, with the requirements for which it is required to trace them down to their code implementation and testing; that is called *forward traceability*. Moreover, especially for safety related applications, it is also important to trace the requirements *backwards*; this helps to reduce unintended behaviors in the modeling and implementation phases. This full round-tripping among the requirements is called *bi-directional traceability*.

The executable specification is made of models representing parts of the system or the complete system itself. At this point, defining application-specific guidelines and rules helps to cope with the complexity and the quality of the design. Indeed, software engineering principles such as *divide and conquer* to reduce the system complexity, *low-coupling* to avoid effects/impacts between components and *high cohesion* within each component must be applied to models, because they are not only nice drawings representing the system or parts of it, but are artifacts that can already be simulated, tested and validated against functional requirements.

Once the model is configured for production code generation, code can be generated automatically out of the design for each components. Finally, the verification of the generated code must be done to confirm that the way it runs onto the embedded processor provides the same results and functionality than the verified model and does not contain any unintended or unexpected behavior.

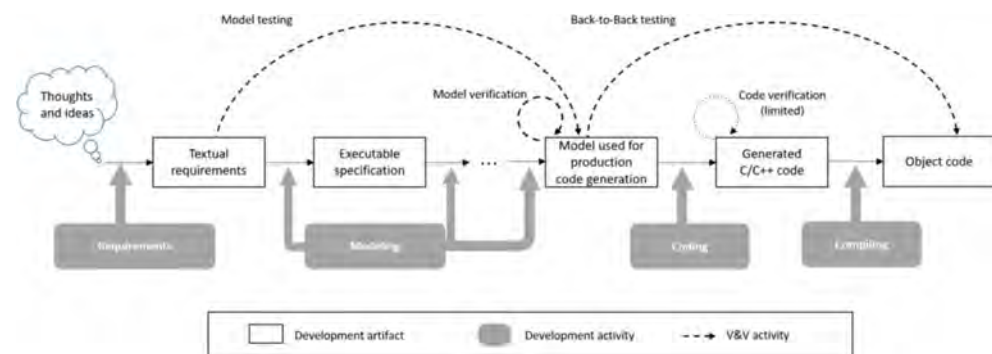


Figure 2.1: ISO 26262 reference workflow for software development & verification processes [1]

2.2 IEC 62304 standard

The international standard IEC 62304 [2] defines a process to produce medical software and especially software within medical devices. This standard is approved by both the European Union (EU) and the United States (US) regulation authorities and can therefore be used as the basis when medical devices products need to be certified for these two markets. The IEC 62304:2015 standard for software systems defines three software safety classes called A, B and C. This classification represents the risk level of harm resulting from a hazardous situation in which the software system contributes. A software system is specified as software safety class:

- (A) if the software system cannot be part of a hazardous situation; or if it can, it must not result in an unacceptable risk after consideration of risk control measures ¹
- (B) if a hazardous situation can arise from a failure of software which results in unacceptable risk after consideration of risk control measures and the resulting possible harm is non-serious injury
- (C) as for (B) except that the resulting harm is a serious injury or even death

2.2.1 IEC 62304 metrics

The metrics table (2.1) shows the MBD phases that are mapped to the processes required by IEC 62304. Entries highlighted in green represent artifacts that can be generated automatically out of models, like production code, test vectors and reports by MBD tools during the design, implementation, verification and validation phases.

Software Development Process	Modeling and Simulation	Metrics
5.2 Requirements Analysis	Requirement analysis via modeling and simulation	Traceability of simulation test cases to functional requirements
5.3 Architectural Design	Develop structural software models	Allocation of requirements to model components
5.4 Detailed Design	Elaborate architectural model with behavior and software specifications	<ul style="list-style-type: none"> • Model coverage during simulation testing • Demonstrated compliance to modeling standards
5.5 Unit Implementation and Verification	<ul style="list-style-type: none"> • Code generator • Unit models and Software-in-the-loop (SIL) and Hardware-in-the-loop (HIL) testing 	<ul style="list-style-type: none"> • Code ⇔ model ⇔ requirements traceability • Unit model vs code behavior (back to back testing) • Structural code coverage • Coding standards compliance • Absence of run-time errors (static analysis) within units
5.6 Integration and Integration Testing	Subsystem models and SIL testing	<ul style="list-style-type: none"> • Subsystem simulation vs code behavior • Absence of run-time errors at code interfaces
5.7 System Testing	System model and SIL testing	<ul style="list-style-type: none"> • System simulation vs code behavior • Absence of run-time errors at OS and driver-level boundaries
5.8 Release	Model configuration management	<ul style="list-style-type: none"> • Issue tracking/resolution • Change management process conformance

Table 2.1: Metrics of the IEC 62304 standard [2]

¹Risk control measures: actions that are taken in response to a risk factor that has the potential to cause accident or harm

Remark: in this project, not all green metrics from the table (2.1) are covered as the final objective is not to certify the developed application/product. Indeed, model and code coverage, compliance to modeling and coding standards and bi-directional traceability with requirements are not covered. However, some of them, such as bi-directional traceability between model and code, back to back testing, system simulation versus code behavior and absence of run-time errors are done and discussed when required in the corresponding sections.

2.3 Medical industry user story


To highlight how MBD is used in the medical devices industry, the user story (2.2) shows an example of a ventilator system that had to be certified against the IEC 62304 standard introduced in §2.2.

This product has been developed by the medical consulting company IMT based in Switzerland. It is a very complex “multiple inputs and multiple outputs” (MIMO) system for which many parameters and properties must be handled to realize the algorithm. Knowing that it had to be delivered as soon as possible on the market with limited development costs, the MBD approach was used to make its development easier, quicker and cheaper with regards to a conservative software development approach.

As a result of this MIMO system project, the following advantages of using an MBD approach stood out:

- 1) Reduce of prototypes development costs through model simulation
- 2) More than 156'000 lines of code were automatically generated in 15 minutes out the top-level Simulink model
- 3) No bug or error in the generated code throughout the full project's duration
- 4) Only two days were needed to verify, validate and test the system against the required performance criteria as opposed to two weeks in the past for projects of equivalent complexity

IMT Developed a highly complex ventilator called Bellavista, partially unknown, MIMO system requiring total robustness and reliability



Challenge
Develop and deliver such a complex and reliable product without spending too much time and money

Solution
Use Model-Based Design to model, implement, test, and deploy the MIMO system onto an embedded hardware

Results

- Improved time to market and used fewer prototypes
- 156k+ lines of source code generated in 15 minutes
- No bugs in the generated code since the project began
- Validation time for ventilation performance reduced by 80%, 2 weeks down to 2 days

“Model-Based Design not only helps to deal with highly complex MIMO systems, but also enables us to identify problems early in the development process. That reduces costs and improves time to market.”

Matthias van der Staay
IMT

matlab EXPO 2016

Figure 2.2: Highly complex MIMO system ventilator requiring total robustness and reliability [3]

3 Mbed Ecosystem

Mbed is an online collaborative platform managed by ARM for engineers interested in rapid Internet of Things (IoT) device development [4]. Mbed provides a full development ecosystem made of the following elements:

- Online and offline software integrated development Environments (IDE)
- Open source code
- Real-time operating system (RTOS)
- Hardware abstraction layer (HAL)
- Hardware evaluation boards that are *Mbed Enabled*

The Mbed model architecture makes its ecosystem extremely flexible and evolutionary for embedded solutions developers. In the Mbed layered architecture (3.1), there are four different layers of abstraction that are called: **Application**, **Mbed OS**, **Mbed HAL**, and **Hardware**. In such architecture, a component in one layer may obtain services only from a layer below itself. For each layer, services are provided through a well-defined interface that is accessed by components located in the layer directly above it. As opposed to the AUTOSAR layered software architecture (3.2) (that is well-known and used in the automotive industry), the Mbed one has a very strict hierarchy that makes the dependency of one layer restricted to the one below it only. In the case of AUTOSAR, there is always the possibility to access a backdoor called “Complex Drivers”. It is located on the right-hand side of the AUTOSAR layered software architecture (3.2) and it allows to bypass some layers for custom hardware access or backward compatible implementations. However, this makes the application hardware dependent which is exactly what Mbed wants to always avoid. Indeed, the goal of Mbed is to be able to reuse the developed software applications, including their hardware access through the HAL, onto any board that is compatible with Mbed, or as they are called *Mbed Enabled*.

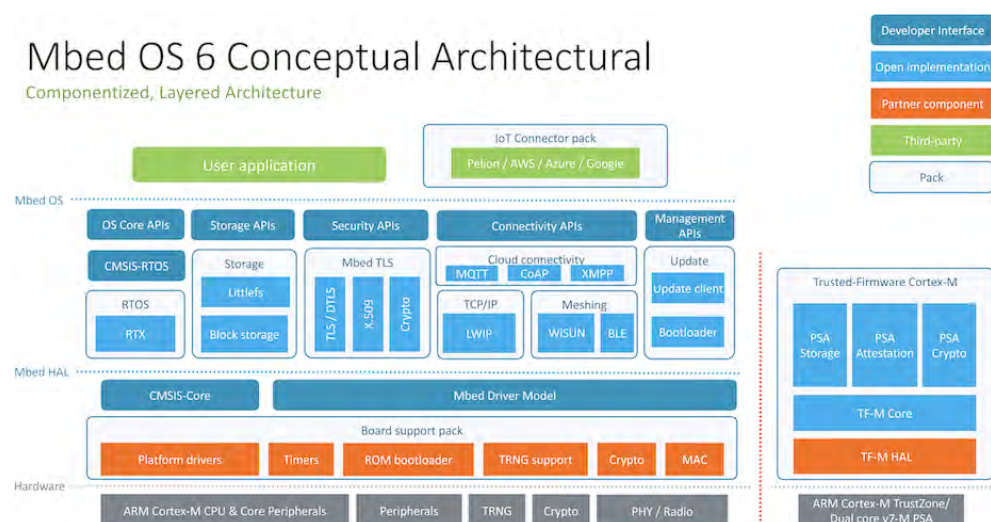


Figure 3.1: Mbed layered architecture [10]

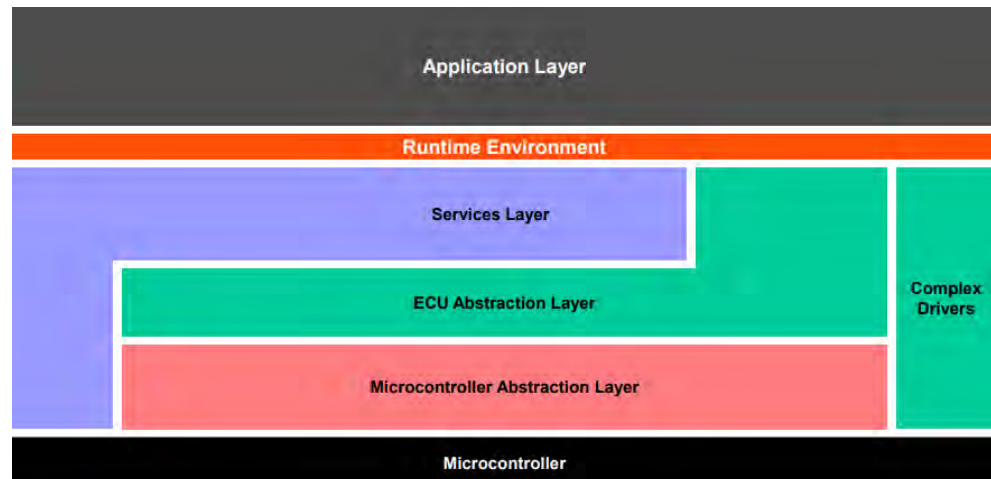


Figure 3.2: AUTOSAR classic simplified layered architecture [11]

Remark: in this project, the emphasis is put on the application, Mbed HAL and hardware layers. The Mbed OS layer is out of scope as it would add too much time and effort to implement it and have it supported in MATLAB and Simulink. However, the Mbed OS is still mentioned in §3.1 to get the big picture on Mbed and its ecosystem.

3.1 Operating System

The Mbed OS has been especially designed for IoT applications. As for the operating system FreeRTOS, Mbed OS is also an open-source OS for embedded systems. It has been designed and developed to mainly run on ARM Cortex-M microcontrollers. This allows developers to quickly develop IoT applications using low-cost hardware.

Here are the most important Mbed OS features:

- As Mbed is designed for real-time systems, performed functions must not only give correct results, but they must provide them at the right time. That is why the software executes in real-time and supports predictable execution, multi-tasking, tasks prioritization, shared resources management via Semaphore or Mutex, queues, timers and Interrupt Service Routines (ISR)
- Fully available open source code that can be used for private, academic or commercial projects
- Modular components as represented in the Mbed layered architecture (3.1) allowing to only include what is required for the designed application
- Drivers to support common available peripherals such as digital and analog inputs and outputs, timers, various communication protocols and so on
- Data security and encryption, as processed data travel onto networks

Remark: developing the integration and the support of the Mbed OS for MATLAB and Simulink could be a topic for another Master thesis to continue and augment the work done in this project. As the focus is on the Mbed HAL, the deployment of the signal processing algorithm is done bare metal without the Mbed OS.

3.2 Hardware Abstraction Layer

One of the main idea of Mbed is to have the application software that is hardware agnostics. This way, it is easy to port an application to another Mbed hardware board without modifying it. It is only needed to compile and link it against the new target. To accomplish this, Mbed provides an *Mbed library* that contains several abstraction layers and *application programming interfaces* (APIs) with respect to the *microcontroller unit* (MCU).

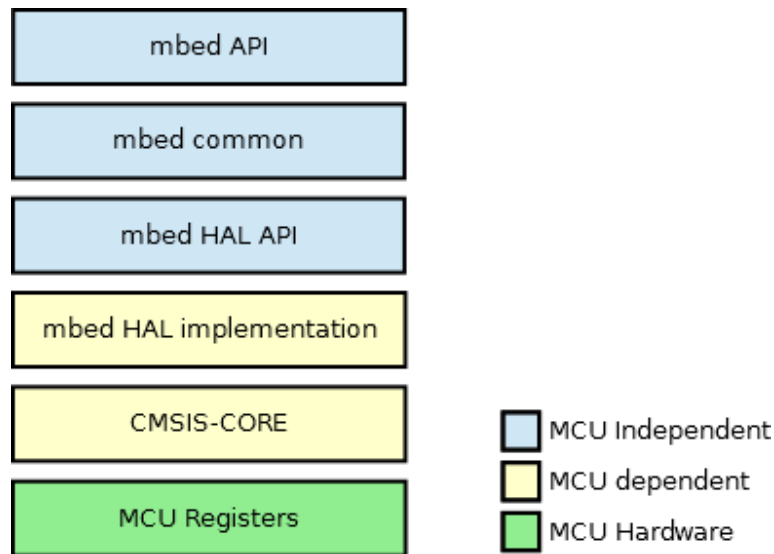


Figure 3.3: Overview of the Mbed software layers architecture [12]

In the Mbed software layers architecture(3.3), there are three distinct sets of layers: the blue one which is independent from the MCU, the yellow one that depends on the MCU, and the green one representing the MCU hardware registers.

The green and yellow groups are out-of-scope in this project as they are hardware related. On the other hand, a very important topic that is being discussed in this project is the comparison between connecting the application to hardware peripherals by using the Mbed HAL and directly connecting it to the low-level hardware drivers. In the first implementation, that uses Mbed, the *mbed API* as well as the *mbed HAL API* are used as mediators to connect to the hardware, abstracting to the user most of hardware specifics. The *mbed common* layer implements functionality, such as data structures and functions that are linked to common peripherals and communication protocols.

The mbed API layer provides peripheral specific C++ classes that contain their related methods and properties. It nicely encapsulates and groups by peripherals low-level C functions that are defined in the mbed HAL API layer. At the end, the user has the choice between accessing the API by using C++ or C code. Having this flexibility of implementation is key for embedded systems as both programming languages are used a lot for embedded applications. As shown in the graph (3.4), it is interesting to notice that C code is still used at least twice more than C++ code for embedded systems in 2020.

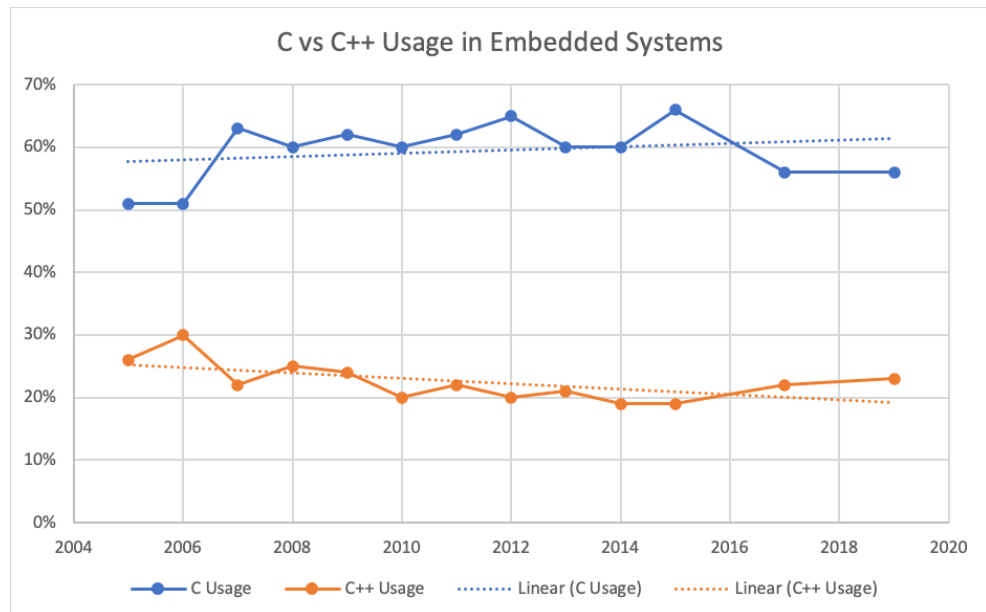


Figure 3.4: Trend in programming embedded systems in C or C++ [13]

3.2.1 Mbed microcontroller library

The Mbed library has been built in a way that each group or type of peripherals, like digital I/O's, analog I/O's, communication protocols, interrupts and so on are defined as separated modules (made of one header and one source files each). As the coding pattern is the same for each module, a detailed description is given in this section for the digital I/O's module, but not for the others.

In the mbed API C++ code (3.5) it is possible to look at the definition of the digital I/O's class including its methods and properties. The class named *DigitalInOut* is made of two constructors; the first one only having the name of the digital pin as input argument, and the second one also has the pin direction (in or out), the pin mode (pull-up, pull-down or high-impedance), and its default value (0 or 1) as additional input arguments. Then, two methods are defined to write data to and read data from a specific pin. Two methods are also available to change the direction of a GPIO pin and one method can change its mode. Finally, one method is available to check if the pin is physically connected to the MCU or if it is not connected (1 or 0).

There is one protected property defined which is a structure called *gpio_t* that contains all needed information about a digital I/O pin to map it to its corresponding low-level hardware description.

In the mbed HAL API and mbed common C code (3.6) there are, in the first half of the code, the function prototypes that correspond to the functions called within the methods defined in the mbed API C++ code (3.5). In the second half of the code, there are the function prototypes belonging to the common API that are called behind the scene and not directly used by the application developer.

Remark: some comments within the mbed MCU independent library files have been simplified in the C/C++ code shown in (3.5) and (3.6) so that the full API modules can be read on one single page each.

```

/* mbed Microcontroller Library */

#include "platform/platform.h"
#include "hal/gpio_api.h"

namespace mbed {
    /* A digital input/output, used for setting or reading a bi-directional pin */
    class DigitalInOut {

    public:
        /* Create a DigitalInOut connected to the specified pin */
        DigitalInOut(PinName pin): gpio() {
            gpio_init_in(&gpio, pin);
        }
        /* Create a DigitalInOut connected to the specified pin */
        DigitalInOut(PinName pin, PinDirection direction, PinMode mode, int value): gpio() {
            gpio_init_inout(&gpio, pin, direction, mode, value);
        }
        /* Set the output, specified as 0 or 1 (int) */
        void write(int value) {
            gpio_write(&gpio, value);
        }
        /* Return the output setting, represented as 0 or 1 (int) */
        int read() {
            return gpio_read(&gpio);
        }
        /* Set as an output */
        void output() {
            gpio_dir(&gpio, PIN_OUTPUT);
        }
        /* Set as an input */
        void input() {
            gpio_dir(&gpio, PIN_INPUT);
        }
        /* Set the input pin mode */
        void mode(PinMode pull) {
            gpio_mode(&gpio, pull);
        }
        /* Return the output setting, represented as 0 or 1 (int) */
        int is_connected() {
            return gpio_is_connected(&gpio);
        }
        /* A shorthand for write() */
        DigitalInOut &operator= (int value) {
            write(value);
            return *this;
        }
        /* A shorthand for write() using the assignment operator which copies the
         * state from the DigitalInOut argument */
        DigitalInOut &operator= (DigitalInOut &rhs) {
            write(rhs.read());
            return *this;
        }
        /* A shorthand for read() */
        operator int() {
            return read();
        }

    protected:
        gpio_t gpio;
    };
} // namespace mbed

```

Figure 3.5: Mbed API for digital I/O's

```

/* mbed Microcontroller Library Hal */

#include <stdint.h>
#include "device.h"
#include "pinmap.h"

/* hal_gpio GPIO HAL function */

/* Set the given pin as GPIO */
uint32_t gpio_set(PinName pin);

/* Checks if gpio object is connected (pin was not initialized with NC) */
int gpio_is_connected(const gpio_t *obj);

/* Initialize the GPIO pin */
void gpio_init(gpio_t *obj, PinName pin);

/* Set the input pin mode */
void gpio_mode(gpio_t *obj, PinMode mode);

/* Set the pin direction */
void gpio_dir(gpio_t *obj, PinDirection direction);

/* Set the output value */
void gpio_write(gpio_t *obj, int value);

/* Read the input value */
int gpio_read(gpio_t *obj);

// The following functions are generic and implemented in the common gpio.c file

/* Init the input pin and set mode to PullDefault */
void gpio_init_in(gpio_t *gpio, PinName pin);

/* Init the input pin and set the mode */
void gpio_init_in_ex(gpio_t *gpio, PinName pin, PinMode mode);

/* Init the output pin as an output, with predefined output value 0 */
void gpio_init_out(gpio_t *gpio, PinName pin);

/* Init the pin as an output and set the output value */
void gpio_init_out_ex(gpio_t *gpio, PinName pin, int value);

/* Init the pin to be in/out */
void gpio_init_inout(gpio_t *gpio, PinName pin, PinDirection direction, PinMode mode,
                    int value);

/** Get the pins that support all GPIO tests
 *
 * Return a PinMap array of pins that support GPIO. The
 * array is terminated with {NC, NC, 0}.
 *
 * Targets should override the weak implementation of this
 * function to provide the actual pinmap for GPIO testing.
 */
const PinMap *gpio_pinmap(void);
}

```

Figure 3.6: Mbed HAL and common API for digital I/O's

In the C++ code (3.5) and C code (3.6), the structure and relationship between layers within the Mbed library is clearly represented. C++ methods from the mbed API access either the C functions from the mbed common layer or from the mbed HAL API. For example, the C++ method `read()` in (3.5) calls the C function `gpio_read()`.

3.3 Hardware

The hardware part of the Mbed ecosystem is very important as, at the end, the developed software applications have to run somewhere. A lot of development boards have been realized by various vendors like for example: NXP, Renesas, STMicroelectronics and many more as shown here: <https://os.mbed.com/platforms>.

What tells the application developer if a board supports Mbed is when it has the **Mbed Enabled** tag on it. This means that the vendor of an Mbed Enabled board has created the MCU dependent layers that are represented in the software layers architecture (3.3), and especially the Mbed HAL implementation one. This is indeed at this level that an application can be ported from one target to another.

3.3.1 Standard pin names

To be able to reuse a developed application onto different hardware targets, it is mandatory that the peripheral identifiers are the same between those. For example, if data are coming in via an external sensor, this one is connected to an analog pin and then forwarded to an *Analog-to-Digital Converter* (ADC) that will discretized the analog signal. Therefore, it is of paramount importance that the name or identifier of the analog pin is the same between all Mbed Enabled hardware systems.

In Mbed, two groups of standard pin names have been defined; the **Generic Pin Names** and **Arduino Uno Pin Names**.

The Generic Pin Names group defines names for the following default peripherals:

- **LED pins:** defined as LEDx (e.g. LED0, LED1)
- **Button pins:** defined as BUTTONx (e.g. BUTTON0, BUTTON1)
- **UART ¹ pins:** defined as CONSOLE_TX and CONSOLE_RX

Every Mbed board includes a serial interface to the host PC, with CONSOLE_TX to send data to the host and CONSOLE_RX to receive data from the host.

The Arduino Uno Pin Names group, for which the connector is physically compatible with the Arduino Uno one, defines names for the peripherals connected through the Arduino Uno connector:

- **Digital pins:** defined as Dxx (from D0 to D15)
- **Analog pins:** defined as Ax (from A0 to A5)
- **UART pins:** defined as TX, RX and respectively mapped to D0, D1
- **SPI ² pins:** defined as CS, MOSI, MISO, SCK and respectively mapped to D10, D11, D12, D13
- **I²C ³ pins:** defined as SDA, SCL and respectively mapped to D14, D15

There are other digital pins (Dxx) that may provide particular functionality, like PWM and timers. However, the application should not assume the same behavior for these pins on all Mbed Enabled targets, as these are not part of the standard pins.

¹UART = Universal Asynchronous Receiver-Transmitter

²SPI = Serial Peripheral Interface

³I²C = Inter-Integrated Circuit

A detailed view of the relationship between the Arduino Uno pins and their functionality is provided in the diagram (3.7).

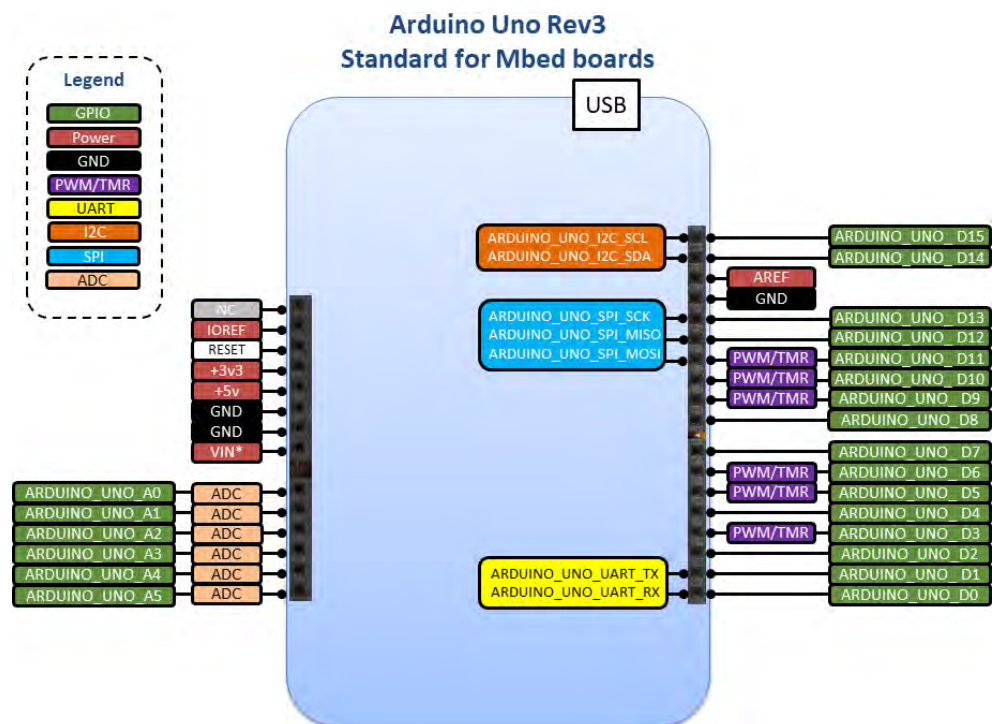


Figure 3.7: Arduino Uno pin mapping to functionality [14]

4 Hardware System

For this Master thesis, a set of hardware components had to be selected in order to fulfill the mandatory requirements of the application that are:

- use of a **low-cost** hardware platform
- use of an **Mbed Enabled** development board
- connect to **analog sensors**
- connect to the **Internet of things** (IoT)
- use **off-the-shelf available hardware** with minimum customization
- use a board that is already supported by a **hardware support package** (HSP) in MATLAB and Simulink. This means that the compiled and linked code can be downloaded to the board via a micro-USB cable directly from Simulink and also that data can be exchanged between the host computer and the board while it is executing the code. This is discussed in details at the §6.6.2

In this project, the goal is to focus more on the software part and its integration with hardware, rather than on the hardware part itself. Therefore, the custom work related to the hardware system has been constrained to its minimum. Basically, it has consisted in stacking two electronic boards together and adding a push button to it as shown in the photo of the hardware system (4.1).

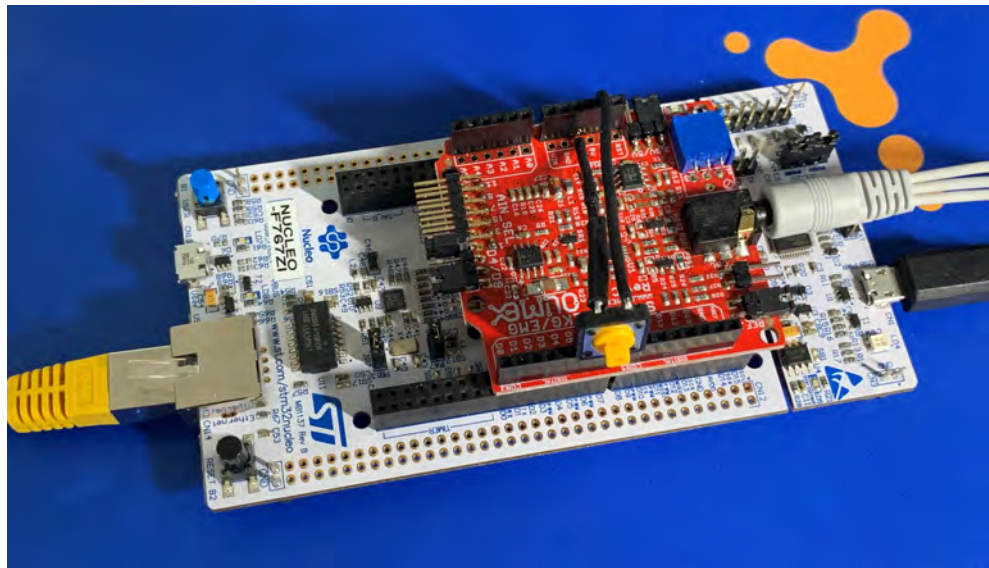


Figure 4.1: Hardware system for the electrocardiogram (ECG) application

The first mandatory requirement mentioned at the beginning of the §4 is about using a low-cost hardware platform. In the table (4.1) the cost of the needed hardware components is summarized. The values are in euros without the value added tax (VAT) as it varies between countries.

Hardware component	Price in € (no VAT)	Shipping cost in € (no VAT)
Evaluation board	30	8
Extension board	20	6
Sensors	10	6
30 electrodes	30	6
Push button	1	6

Table 4.1: Costs of the needed hardware items

Based on the values provided by the table (4.1), an overall cost estimation can be calculated. Knowing that the evaluation board comes from one vendor, and the four other items from another one, the shipping costs must not be counted for each entry, but for only once per vendor. The names of the parts vendors are not mentioned as there are many of them providing these items and they may differ based on the country the end user is located. The total cost is calculated in the expression (4.1).

$$TotalCost = (30 + 8) + (20 + 10 + 30 + 1 + 6) = 105 \text{ € (no VAT)} \quad (4.1)$$

For example, if the purchase is done from Germany, the *TotalCost* is then multiplied by a VAT factor of 19% in 2021. The *FinalCost* is given by the expression (4.2).

$$FinalCost = 105 \cdot 1.19 = 124.95 \cong 125 \text{ €} \quad (4.2)$$

At the end, the final cost of hardware for the end user is of 125 €, and he or she only has to solder one push button (one pin to the ground, one pin to the 3.3 Vcc, and one pin to the D6 I/O on the Arduino Uno connector) as shown in the figure (4.1).

4.1 Evaluation board

The main development board that is being used for this project is the *STM32 Nucleo-F767ZI* from STMicroelectronics [15]. It has been selected, because it fulfills all the requirements defined in the §4:

- it is Mbed Enabled
- it has the standard Arduino Uno Revision 3 connector that provides analog signals connectivity
- it has an Ethernet connector to send data over IT networks
- the board is fully ready for the application and does not need customization
- this board is already part of an STM32 HSP in MATLAB and Simulink and can be directly connected to the integrated development environment (IDE)

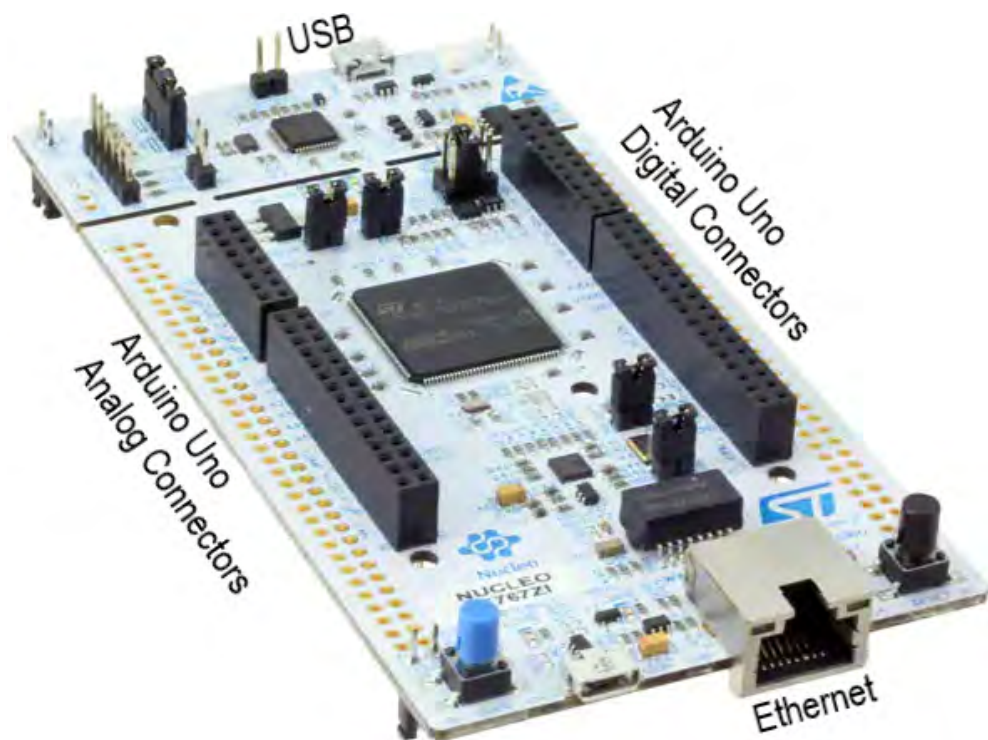


Figure 4.2: STM32 Nucleo-F767ZI development board [15]

The STM32 Nucleo-F767ZI development board (4.2) can be connected to a host computer and programmed using MATLAB and Simulink via the micro-USB port that is located on top of the board. The Arduino Uno connectors, located on both sides, provide access to digital and analog I/O's. For the IoT part, the main board does not have any wireless module. Such modules exist, as extension board, and can be plugged-in the main board. However, there is an Ethernet port, at the bottom right, that is available and is used to exchange data over IT networks. The software application uses several packet communication protocols such as UDP and TCP/IP for a local private network and a Channel-Based Communication protocol over Internet for the Cloud that can be routed through this Ethernet port.

4.1.1 Main characteristics of the microprocessor

There are three family of ARM Cortex processors:

- **A** for Application processor cores optimized for high performance systems
- **R** for Real-time application cores optimized for hard real-time systems
- **M** for Microcontroller cores optimized for low-power embedded applications

The STM32 Nucleo-F767ZI development board is made of an ARM Cortex-M7 32bit microprocessor (MCU) having the following main characteristics [16]:

- Core: CPU frequency up to 216 MHz with an L1-cache of 16 Kbytes
- Memory: 2 Mbytes of Flash and 512 Kbytes of RAM
- Energy: optimized for low energy integrated circuits
- Power supply: dedicated USB power
- Peripherals: 168 configurable I/O's, 3x12bit Analog-to-Digital Converter (ADC), 2x12bit Digital-to-Analog Converter (DAC), 16-stream Direct Memory Access (DMA), 18 timers
- Connectivity and communication interfaces: JTAG, I²C, UART, and many more

4.2 Extension board

The extension board that is being used for this project is the *Shield-EKG-EMG* from Olimex [18]. This board has been especially designed, with specific amplifiers and filters, to measure what are called biosignals such as electrocardiogram (ECG) and electromyogram (EMG), which is exactly what is needed for this project as ECG signals are measured.

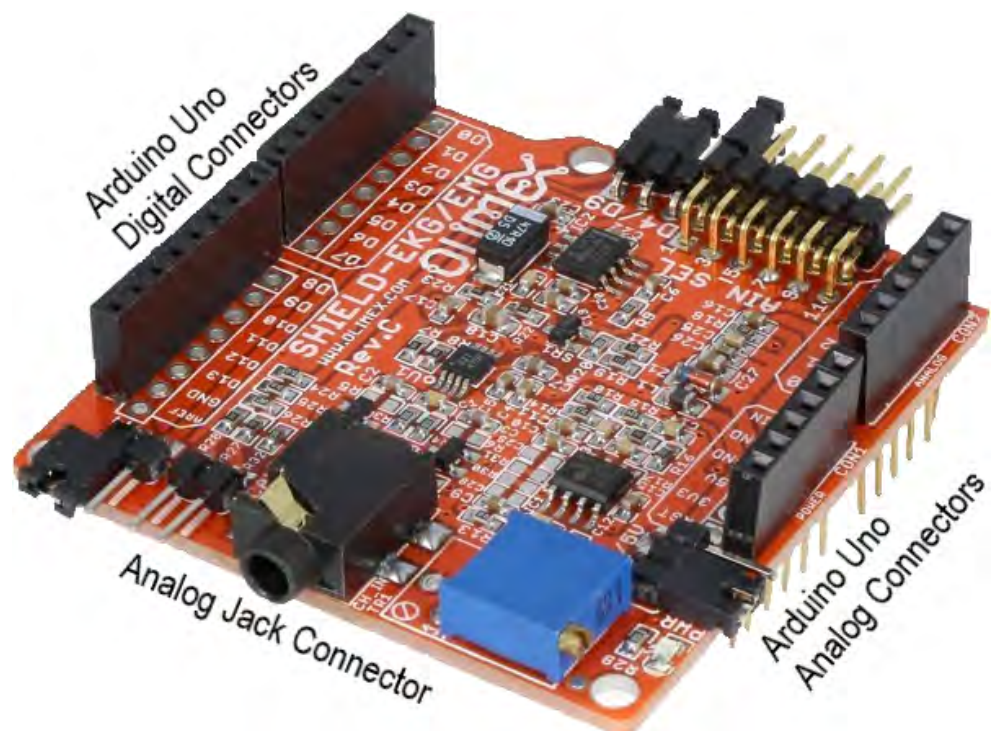


Figure 4.3: Olimex EKG-EMG Shield [18]

The Olimex extension board can be connected to a main board like the one shown in (4.2) via the Arduino Uno connectors. Analog signals are acquired by the Jack connector and recomposed before being routed to the main board.

4.2.1 Schematic of the analog path

The stages of the analog path are shown by the schematics (4.4), (4.5), and (4.6). For clarity, they have been split into the following three stages: signal acquisition, variable gain, and pre-emphasis.

The first stage (4.4) is made of the jack connector, on the left-hand side, that gets two analog channels (**Left** and **Right**) as inputs and the ground (DRL). It then combines the signals to provide the **Lead I ECG signal** and amplifies it by a factor of 10.

The second stage (4.5) allows to amplify the signal even more by using a variable regulated gain. In this project, it has been set to its maximal value which gives an amplification factor of 15. If this value is changed, it has a direct impact on the peaks detection threshold within the post-processing stage of the ECG software application described at §6.2.1.

The third stage (4.6) is made of a pre-emphasis filter of third order that increases the amplitude of low frequencies, that are below 40 Hz, by a factor of 3.56.

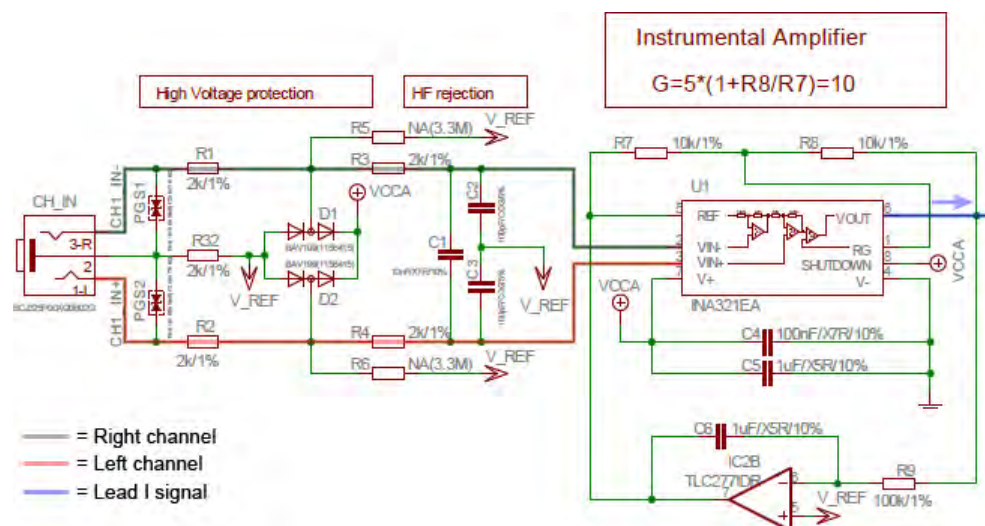


Figure 4.4: Olimex EKG-EMG Shield signal acquisition, stage 1 [18]

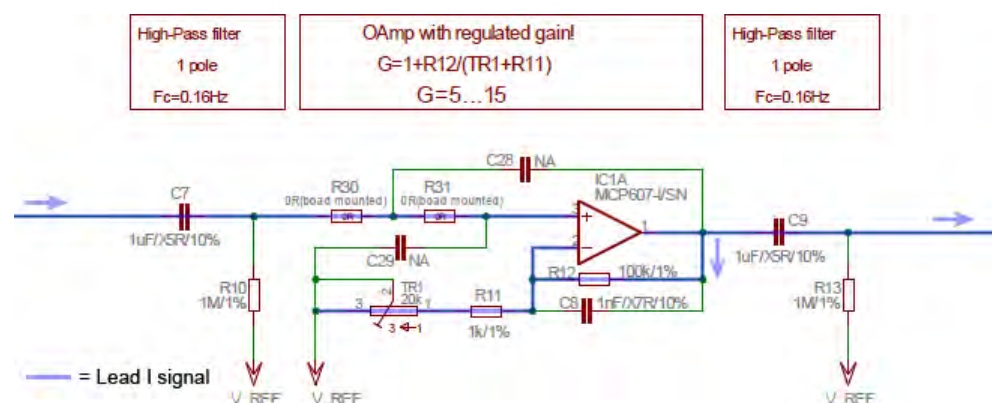


Figure 4.5: Olimex EKG-EMG Shield variable gain, stage 2 [18]

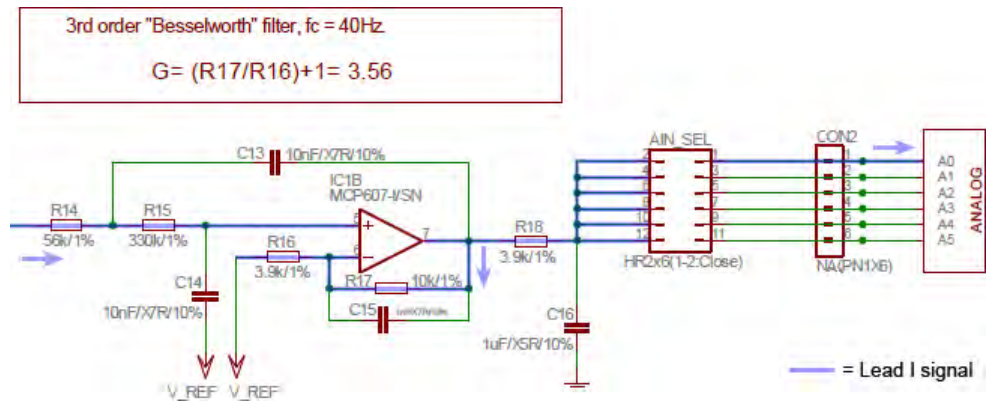


Figure 4.6: Olimex EKG-EMG Shield pre-emphasis, stage 3 [18]

At the end of the third stage (4.6), the **Lead I ECG signal** is routed towards the standard Arduino Uno pin A0 as seen in the §3.3.1. An ADC is connected to this pin transitioning from the continuous time domain to the discrete time domain and allowing to do digital signal processing within the MCU.

4.3 Sensors

The sensors are three passive electrodes that can be connected to the extension board (4.3) via a jack connector.



Figure 4.7: Olimex SHIELD-EKG-EMG-PRO Sensors

The electrodes shown in (4.7) are marked with *L* for the Left arm, *R* for the Right arm, and *D* for the Driven Right Leg (DRL) ground.

Remark: to measure a correct input signal, the electrodes must be placed on the patient as indicated in the Einthoven's triangle representation (6.2).

5 Model-Based Design

Nowadays, very complex embedded systems are made of several dozens or even hundred processors that are programmed with several million lines of code making them very difficult to program by hand. For example, a magnetic resonance imaging (MRI) machine contains around ten million lines of code [6]. The Model-Based Design approach (MBD) is a modelling method that allows to handle such level of complexity. In the literature, it is also called Model-Driven Engineering (MDE).

5.1 MBD workflow

The MBD workflow (5.1) starts with the project's research and requirements on top. At this stage, there is a need to gather, elicit, negotiate over the requirements before going to the design phase. With an MBD approach, it is easier to iterate between the design and the requirements to improve their correctness and refine the specifications for each components.

At the design level, also called the executable specification, the full system can be implemented in components with its external stimuli for testing. This way, the system can already be tested earlier in the development process which is a considerable advantage, as in practice, real system prototypes often are not easily available and are usually incomplete and expensive to build; thus preventing rapid iteration or system level testing early on.

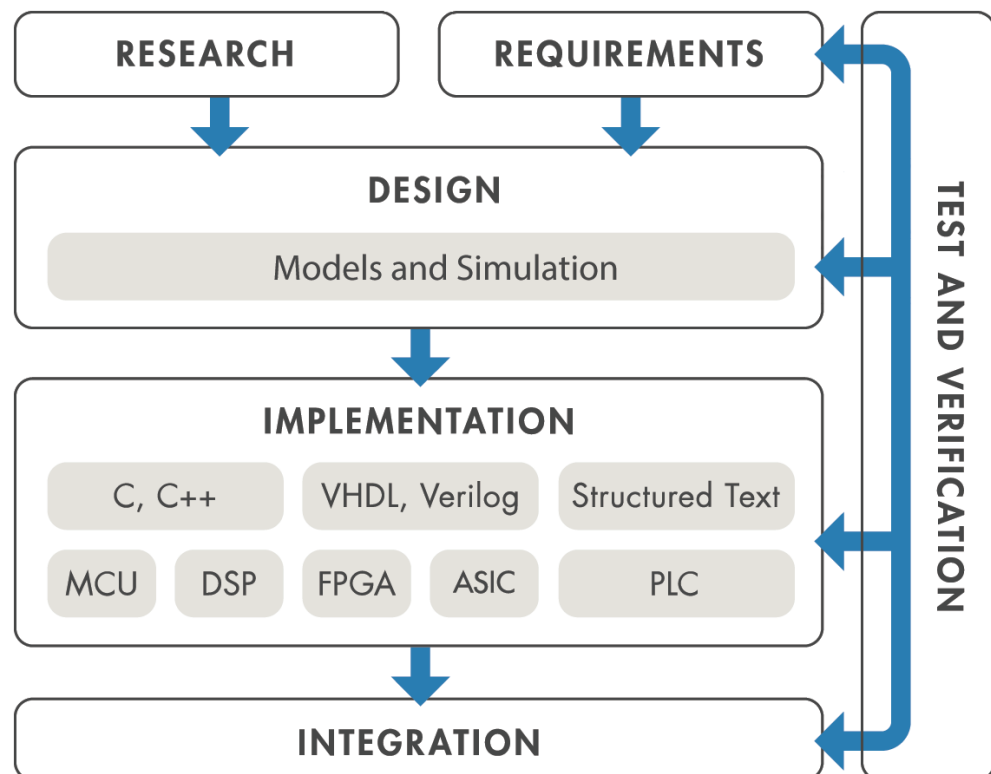


Figure 5.1: MBD Workflow [7]

In the implementation phase, code can be generated automatically out of the design for each component and reused onto embedded systems or real-time machines. The full system can already be tested prior to its final integration without using costly hardware test benches. This allows to detect the majority of the errors prior to the final integration and test phases for which fixing errors is expensive.

With an MBD approach for the medical devices industry (as followed in [5]), the testing, verification and validation phases are present all along the workflow to help detecting errors very early in the process, and therefore allowing to quickly design safer products with well understood and tested performance.

5.2 Mapping to the V-model

In the medical devices industry, the V-model is the reference development process for products development. It can be used following a waterfall or agile (with iterative sprints) development methodology.

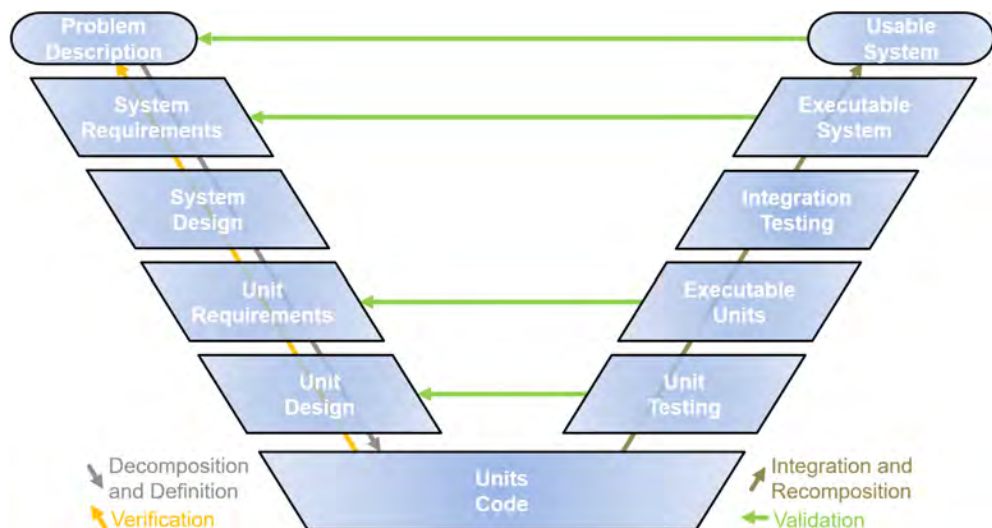


Figure 5.2: V-model development process

The V-model process (5.2) starts with the development phases on the left-hand side. It is made of the gathering, elicitation and negotiation steps over the requirements on top and it goes down through the design of the system and its decomposition into design units and finally code units. On the right-hand side, it goes up with the testing and integration phases. It starts with the testing of each units and their validation. Then it goes through the integration of these units and their testing until the validation of the system is correct.

Being able to map the MBD workflow (5.1) to the V-model (5.2) is key to see how the MBD approach can fit such a product development process. The mapping of the two, done in the figure (5.3), highlights how the different phases of both representations match or align among each other. Basically, the test and verification phase that is all along the MBD workflow maps to the right-hand side of the V-model. The other phases of the MBD workflow can be mapped to their corresponding phases on the left-hand side of the V-model at both, the system and unit levels.

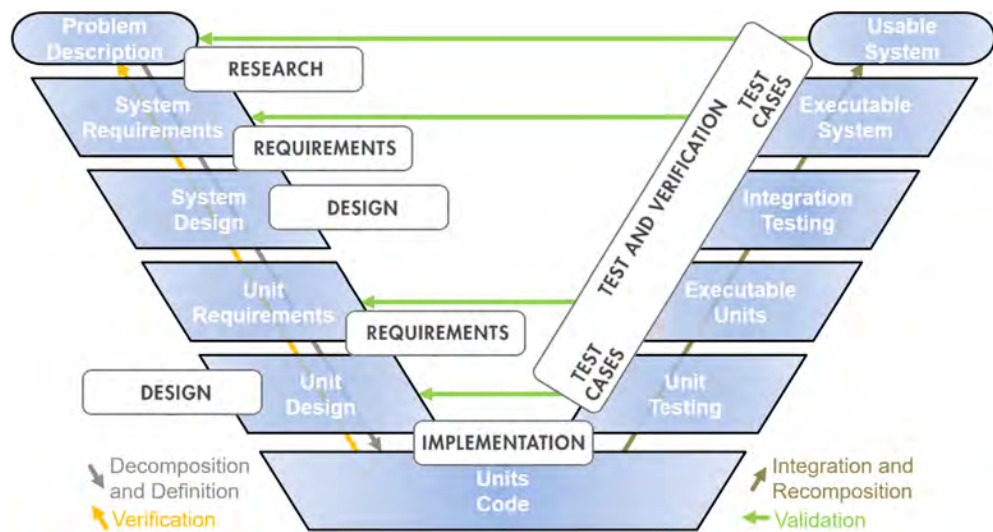


Figure 5.3: Mapping of the MBD workflow to the V-model development process

Remark: in the medical IEC 62304 standard [2], it is requested to decompose the system into separate units having a clear definitions of their interfaces. As in the ECG signal processing chain detailed in the §6.2.1 there are not that many complex components with formally defined specifications and also no clear interface contracts for them, this aspect requested by the standard is not applied. Indeed, this would only add a new abstraction layer to the problem to solve which is not meaningful in this project and this is also not the goal of this Master thesis.

5.3 Software tools used in this project

As mentioned in §1, one of the main goal of this project is to allow engineers and developers to be able to design, test, implement and deploy onto an Mbed Enabled hardware their algorithms directly from Simulink. This way, the MBD approach can be used abstracting the development complexity at many different levels like programming language, hardware drivers and connectivity, as well as the testing, verification and validation of the application. To do this, the following MathWorks tools have been used:

1) Integrated Development Environment

- *MATLAB* as the *language of technical computing* (LTC) and main platform
- *Simulink* as the *Design Automation* (DA) and dynamic system modelling platform

2) Domain Specific Libraries

- *Signal Processing and DSP System Toolboxes*
- *Wavelet Toolbox* (optional)

3) Ethernet Communication Layer

- *Instrument Control Toolbox* to manage UDP and TCP/IP communication protocols for data exchange on a local network

4) Automatic Code Generation

- *MATLAB Coder* to generate C/C++ code out of MATLAB code
- *Simulink Coder* to generate C/C++ code out of Simulink models
- *Embedded Coder* to optimize the generated code for embedded systems

5) Hardware Integration

- *Hardware Support Package (HSP)* to directly connect to the C/C++ code generation tools providing capabilities such as board connectivity, *Code Replacement Libraries (CRL)* and low-level drivers implementation to control hardware peripherals

5.4 Competitor tools and solutions

The analysis of the competitors is based upon three main abstraction criteria:

- Model-Based Design vs handwritten code
- Mbed support or not
- Code generation or not

Based on these criteria, the competitors table (5.1) has been setup and populated with the main competitors (open source or commercial) present on the market. It uses the following ratings for each criterion:

- “++” = good support
- “+” = partial support
- “±” = partial support with a different criterion's approach
- “o” = no support

Main competitors	Model-Based	Mbed support	Code generation
Mbed Online Compiler, and Mbed Studio and CLI	o	++	+
LabVIEW from NI	±	±	+
Scilab and Xcos	+	+	+
ANSYS and Scade	++	o	++
MATLAB and Simulink	++	++	++

Table 5.1: Assessment of competitors to MATLAB and Simulink

As Mbed is C/C++ based, there already exists some solutions that are available on the market to develop Internet of things (IoT) applications using this hardware abstraction layer (HAL). Many of them are based on handwritten code, but are also using frameworks to generate part of the C/C++ code automatically to facilitate and speed-up the development of IoT applications. On the [Arm Mbed official website](#) the following three Integrated Development Environments (IDEs) are available to author and reuse C/C++ source code: [Mbed Online Compiler](#), [Mbed Studio](#), and [Mbed CLI](#) (command Line tool for Mbed). All of them can be used on Windows, Mac and Linux platforms. More information are provided in §5.4.1 and §5.4.2 about these IDEs.

These are not the only solutions when end users work at the code level. The following other IDEs are also available and do support Mbed as well: [Keil uVision \(ARM official\)](#), [IAR Embedded Workbench](#), [Atollic TrueSTUDIO \(Eclipse based\)](#). The concepts of these IDEs are similar to the Arm official ones except that they are commercial ones.

Regarding competitors following an MBD approach, there are third party blocksets that connect to Simulink and the Embedded Coder. These are developed by providers such as [Aimagin Waijung](#) or [ST32MAT/TARGET](#). Both of them rely on MATLAB, Simulink and the Embedded Coder products. They are simply third party toolboxes that can be used like any other MathWorks toolboxes for simulation and/or C/C++ code generation.

There is also [LabVIEW](#) from [National Instruments \(NI\)](#) that can do Remote Procedure Call (RPC) from a computer running a LabVIEW model and communicating with an Mbed Enabled hardware via a serial or Ethernet connection called respectively *SerialRPC* or *HTTPrpc* to execute subroutines on it. To achieve this, LabVIEW Virtual Instruments (VI) have been created to mirror the Mbed API. A VI is a basic block in LabVIEW similar to what subsystems are in Simulink. However, the LabVIEW model is not deployed onto the embedded system, but it connects to the target and exchanges data via a communication channel. This workflow does not follow the MBD approach as it is defined in §5 and is therefore not further detailed.

One of the closest available competitor of MATLAB and Simulink following an MBD approach and partially supporting Mbed is the [Scilab](#) platform and its extension called [Xcos](#). The former provides the computation engine like MATLAB does and the latter is a dynamic systems modeler and simulator in discrete and continuous time domains similar to Simulink. Scilab and Xcos are parts of an open source solution. More information on their usage with regards to Mbed are provided in the §5.4.3.

Finally, there is the [Ansys SCADE Suite](#) which is a model-based development environment for reliable embedded software. It can generate C and Ada code from SCADE models that can be used for certification against some industry safety related standards. However, there is no Mbed support at all from SCADE and that is why it is not further detailed.

5.4.1 Mbed Online Compiler

The online IDE for Mbed (5.4) can be accessed for free as soon as an Mbed Enabled board is purchased, like the STM32 Nucleo-F767ZI (4.2). With such a board, the required license key is directly embedded onto the hardware to make its activation easier for the end user who just needs to create an online account on the [login page](#) and can then use the Mbed Online IDE to develop, compile and link its application.

With the Mbed Online IDE, once the binary file is generated for the desired target, it is automatically downloaded into the default download location locally on the end user computer. When the board is connected to the computer over USB, it appears on the computer as a removable storage. The user just has to drag and drop the binary file from the download folder to the root folder of the removable drive and this programs the board automatically which is very easy and convenient to use.

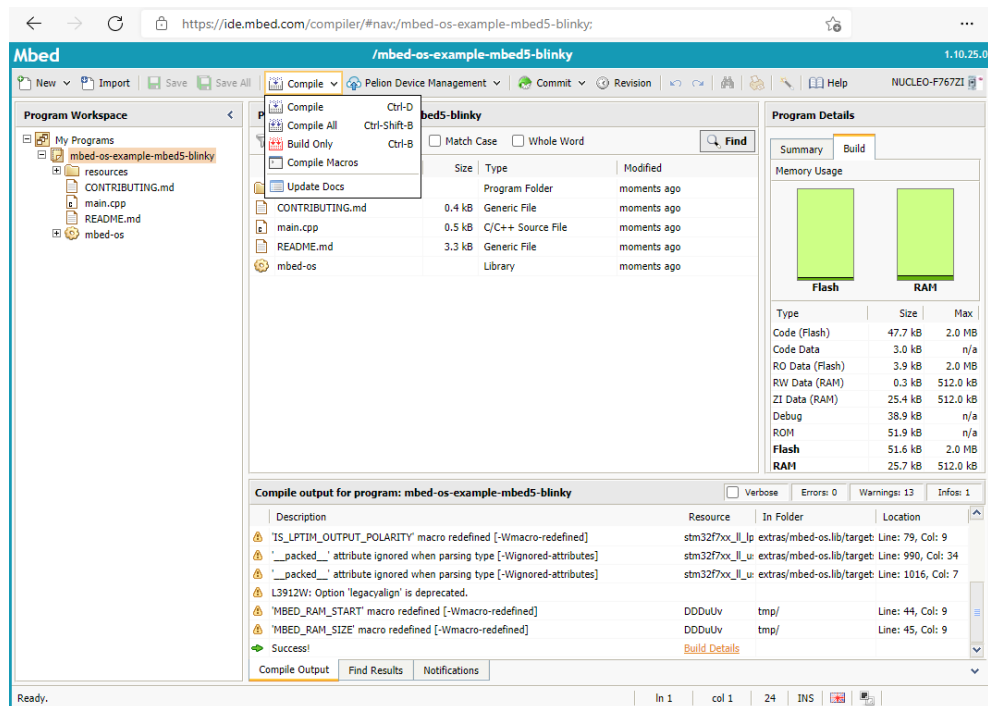


Figure 5.4: Mbed Online IDE

5.4.2 Mbed Studio and CLI

The Mbed Studio and CLI IDE for Mbed can also be accessed for free in the same condition as described at the beginning of §5.4.1. There is a need to be connected to the Internet for the first login to go through the activation phase. After this, both of them can be used offline, except for additional functionality, like version control or IoT data management for example. The combination of these two tools provides similar services and experience to what very well-known C/C++ IDEs like *Visual Studio* or *Eclipse* do provide.

5.4.3 Scilab and Xcos with Mbed

There are lite hardware support packages (HSP) that are available for Xcos. They follow the same concepts as the ones available with the Embedded Coder toolbox for MATLAB and Simulink, but they are much more limited.

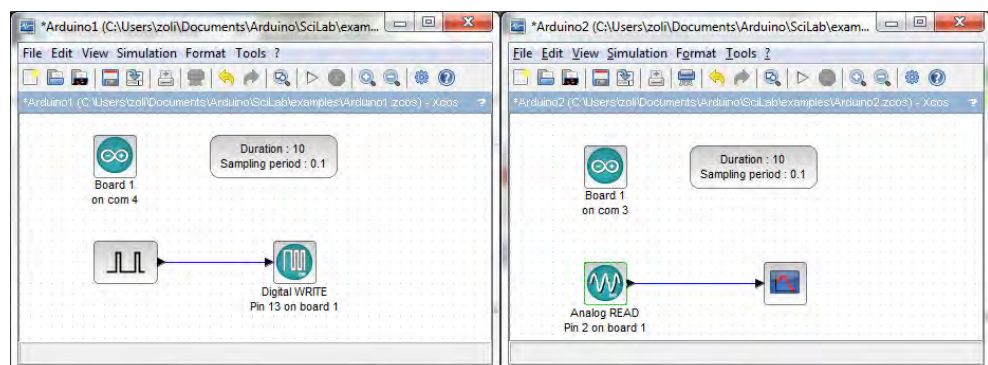


Figure 5.5: Mbed support example from Scilab/Xcos for the Arduino platform [8]

The Mbed support for digital and analog read and write functionality is only available out of the Arduino HSP and not for other platforms. In the example (5.5) models have been designed to do a digital write and an analog read using Mbed. Such hardware support is being provided by an individual contributor and can be found here [8].

There is also an HSP for some STM32 targets, but without the Mbed support, and the MBD workflow is not fully automated as it is depicted in the diagram (5.5). In this case, the blocks configuration must be done at the low-level hardware using the [STM32CubeIDE](#) development environment, requiring the end user to have deep technical knowledge about the used embedded target. More info can be found here [9].

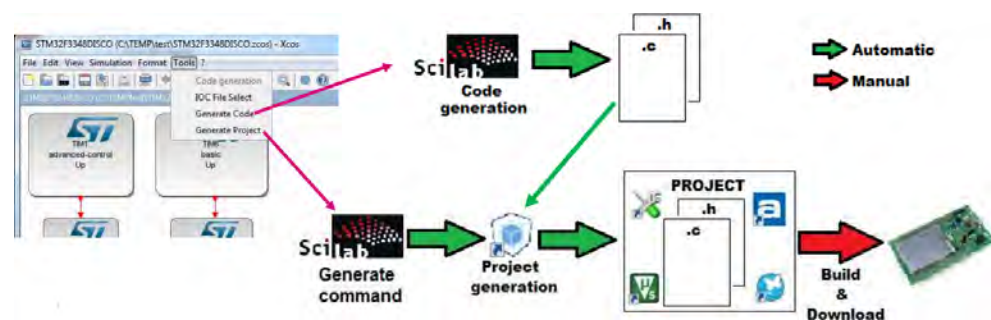


Figure 5.6: HSP for STM32 targets from Scilab/Xcos [9]

5.4.4 Observations on competition

With the aforementioned IDEs, they definitely provide an added value when working at the source code level. It is very efficient to quickly deploy a short algorithm onto the target. However, this is true when the end user is a software developer or a software engineer who is familiar and has experience with C/C++ programming language and embedded systems specifics. For an engineer working at the domain and algorithmic levels, this is not the most efficient way to do rapid prototyping and to test on the hardware what has been designed. Also, when it comes to tuning and maintaining the code, this can become very complicated and time consuming. This is where MBD can help a lot.

The only alternative to MATLAB and Simulink with regards to the MBD approach and the support of Mbed is provided by Scilab and Xcos. However, the Mbed support is very limited and only very basic applications could be done. The implementation of a complete algorithm to process electrocardiogram (ECG) signals and its deployment onto an embedded system seems to be extremely difficult to realize with such IDE.

As summarized in the competitors table (5.1) there are other good MBD solutions, like LabVIEW and ANSYS Scade, but the main problem is that they respectively lack deployment capabilities and Mbed support which is problematic to quickly develop an Mbed related IoT application.

Based on this analysis of competitor tools and solutions, the conclusion is that for an MBD approach, only MATLAB and Simulink can be used to develop, deploy and test an application in a professional way and without requesting the end user to be a hardware and/or software expert.

6 Application

In §1 and §4, the used application and its objectives has been briefly introduced. In this Master thesis, the application part is one of the most important because it provides the context boundaries and specifications of the project for which the main goal is to implement, compare and profile the Mbed Hardware Abstraction Layer (HAL) with respect to the low-level implementation of the drivers.

In §3, it is stressed that Mbed has been especially designed for rapid Internet of things (IoT) device development and to abstract developed applications from the used hardware and its complexity. Fast development, IoT, and abstraction are the keywords here. Due to the pandemic situation that happened in 2020 and 2021, the engineers working in the medical devices industry need to be more effective and productive to quickly develop embedded real-time, connected and safety related applications. This supports the biggest trend in this industry which is even called the *Internet of medical things* (IoMT).

Therefore, the focus of the application has been put on the development of an electrocardiogram (ECG) algorithm which is a common problem to solve in the medical devices industry and that can be used to support the aforementioned points.

6.1 Electrocardiogram basic theory

An ECG is a measure that checks how the human heart is behaving by looking at its electrical activity. An electrical impulse or wave travels through the heart with each heartbeat to produce a train of the following normal sinus rhythm one (6.1).

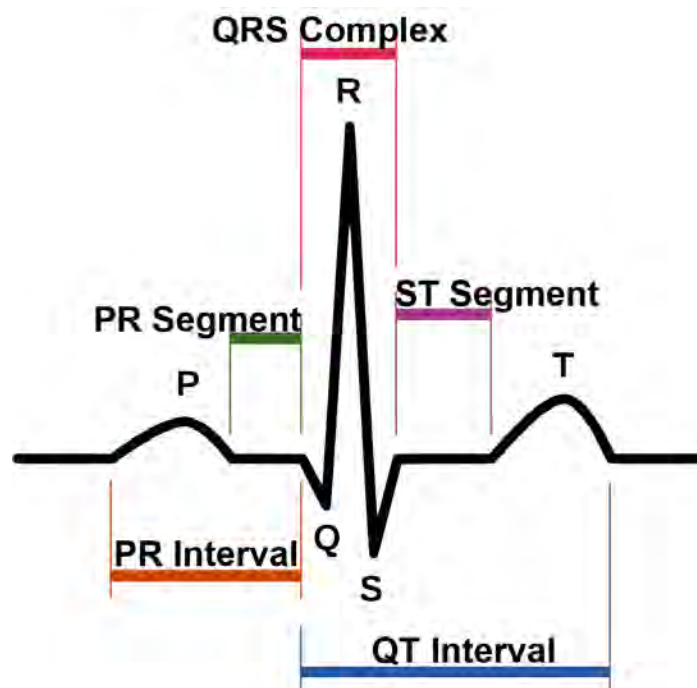


Figure 6.1: Normal sinus rhythm impulse [19]

When electrodes are used to measure a single-lead ECG, they must be placed at the right locations on the patient's body following the Einthoven's triangle (6.2). This triangle represents three differential measurements of electrical signals that are captured by the electrodes and sent to the data acquisition hardware described in §4 to be recombined into one single ECG signal.

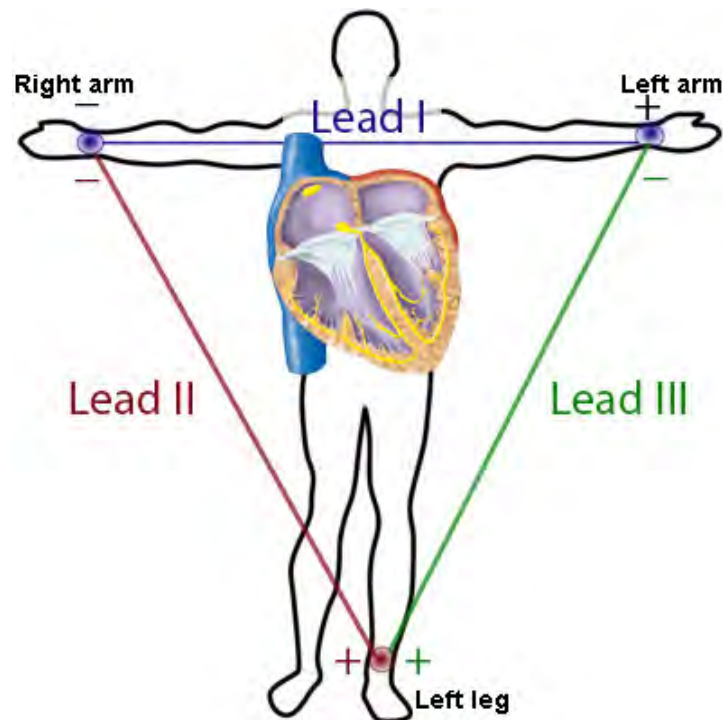


Figure 6.2: Einthoven's triangle [20]

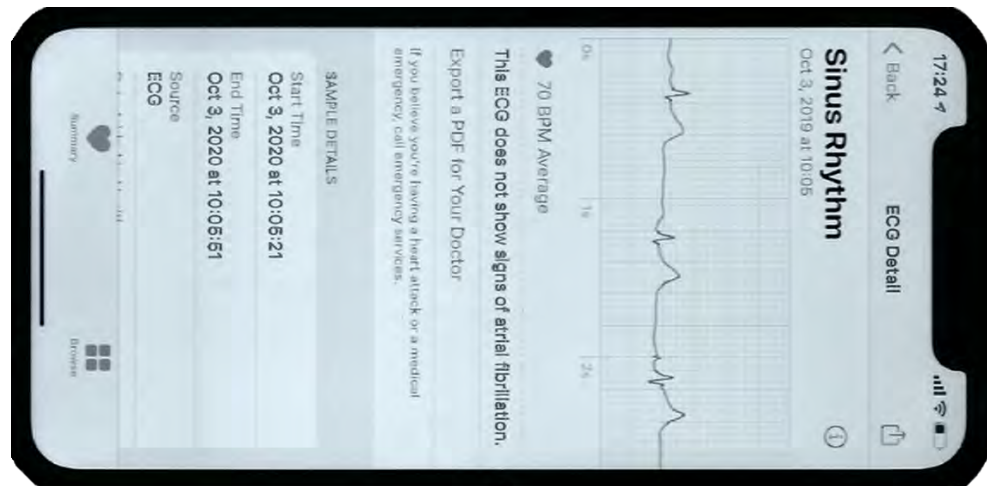
6.2 Electrocardiogram problem formulation

In this project, several reference ECG signals have been acquired by using a smart watch with an optical heart sensor that provides a signal like the one depicted in the example of a captured ECG signal (6.3). This is a typical measurement of an ECG signal taken between the right and left arms following the triangle's technique (6.2).



Figure 6.3: Example of a captured ECG signal with an optical heart sensor from a smart watch

Ten reference measurements have been taken and transmitted to a smartphone, before being forwarded to a computer as a comma-separated value (CSV) file (6.4). They serve as test signals for the algorithm implementation in §6.4, §6.5, and §6.6.



```

1 MeasurementId,MeasurementDate,MeasurementTime,NumberOfValues,ValuesChunk
2 1,2020-11-01,20:07:19,15328,0.00001578 0.00000320 -0.00000983 -0.0000233
3 2,2020-11-04,14:15:09,15328,0.00039330 0.00057526 0.00058876 0.00055363
4 3,2020-11-07,14:52:50,15328,0.00053717 0.00054353 0.00053597 0.00054600
5 4,2020-11-10,14:53:44,15328,0.00071225 0.00095084 0.00093720 0.00084823
6 5,2020-11-13,14:29:18,15328,-0.00225236 -0.00179177 -0.00166143 -0.00164
7 6,2020-11-16,16:41:20,15328,-0.00020077 -0.00019160 -0.00018206 -0.00017
8 7,2020-11-19,11:54:12,15328,0.00252507 0.00292403 0.00320922 0.00331579
9 8,2020-11-22,09:05:21,15328,0.00053583 0.00053895 0.00054163 0.00054384
10 9,2020-11-25,19:50:31,15328,-0.00087423 -0.00130669 -0.00156705 -0.00163
11 10,2020-11-28,16:44:02,15328,0.00475347 0.00441380 0.00446931 0.00438371

```

Figure 6.4: Export of data into a CSV file

By collecting and analyzing an ECG signal (6.3), it is possible to extract several information like: the heart rate (distance between two similar and consecutive peaks), the shape of the electrical activity to see if it is normal or irregular (arrhythmia), or hidden heart diseases for example.

6.2.1 Signal processing chain

From a signal processing point of view, an ECG wave is very interesting to study and process, because it is made of both: *low frequency* and *high frequency* components. In this project, the goal of the application is to extract the heart rate or in other words to determine the distance between two consecutive high frequency **R peaks** (cf. Normal sinus rhythm impulse (6.1)).

To get the number of beats per minutes (BPM) out of an ECG captured on a patient, a one minute measurement can be done and the high frequency peaks can be manually counted. To speed it up and still have an accurate result, a 30 seconds measurement can be realized. The sum of the counted peaks is then multiplied by two to get the final BPM value.

The goal of the application software is to automate the process measuring the BPM. It first filters the ECG signal to separate and extract the high frequency peaks and then counts the number of peaks and compute the time difference between every pair of consecutive peaks. Based on this, it is then possible to come up with the signal processing chain represented in the block diagram (6.5).

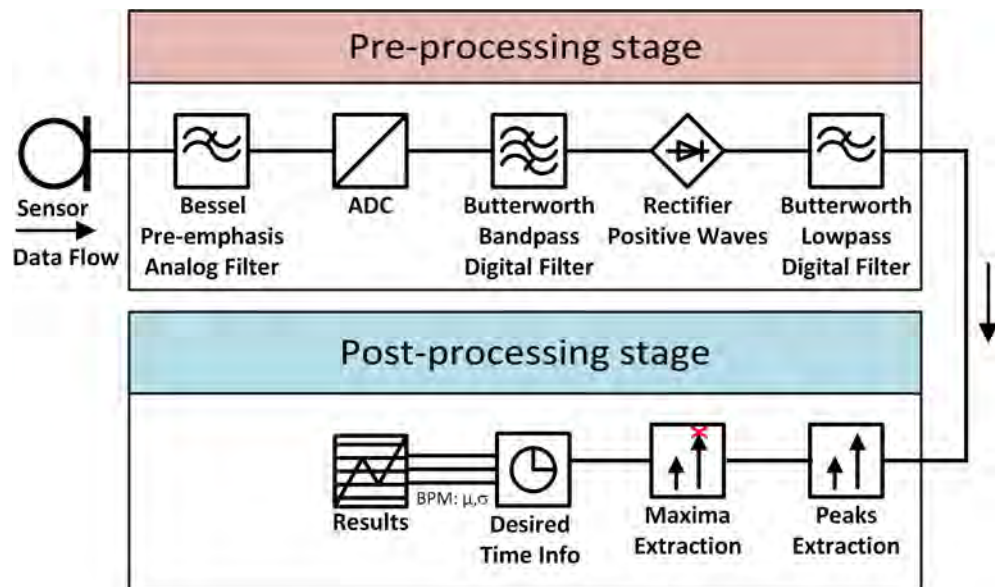


Figure 6.5: Signal processing chain to extract the heart rate of an ECG signal

Out of the diagram (6.5), it is possible to identify two distinct sections; the first one that goes from the input sensors to the output of the “Lowpass Digital Filter” block, which represents the *pre-processing* part, and the second section that goes from the input of the “Peaks Extraction” block to the output of the processing chain which represents the *post-processing* part. The first stage focuses more on filtering the ECG signal and the second one more on computing the needed statistics.

The aim of the pre-processing stage is to almost vanish the low frequency components to only keep the high frequency peaks. That is why a “Bandpass Digital Filter” subsystem is used. The coefficients of this filter must be tuned precisely for this first stage to work correctly. The “Rectifier” subsystem is there to get the absolute value of the signal and the “Lowpass Digital Filter” subsystem integrates over the processed signal. The “Pre-emphasis Analog Filter” located on the extension board (4.3) amplifies the frequency components of interest in the acquired ECG signal to reduce the effect of potential higher ones.

The post-processing stage must then distinguish between global maxima representing the high frequency peaks and local maxima that are some remains of the filtering process. This is done by the “Maxima Extraction” subsystem that has a threshold parameter that must be tuned precisely so that the global maxima are identified correctly. Once these maxima have been extracted, the “Desired Time Info” subsystem computes the time difference in seconds between each pair of consecutive peaks and provides results in the desired BPM format.

6.2.2 Sampling frequency

The heart rate values depend on the age and gender of human beings. In general, when there is a normal sinus rhythm, the maximal possible value for an athlete is of 220 BPM. In theory, the maximum possible heart rate for a human being is of 300 BPM. A value that would be over this theoretical one would mean that the heart is not following a normal sinus rhythm and that the patient life is in severe danger.

Based on these considerations, it has been decided for this project, to have the system being able to measure a heart rate within the range of 0 to 255 BPM. An advantage of this choice is that BPM values can be encoded as unsigned integer of 8bits. Moreover, if the maximal value of 255 BPM is reached, this highlights the fact that it is abnormal and the patient needs urgent assistance.

Following the *Nyquist frequency* principle for a digital system, the *sampling frequency* (FS) of the digital part of the signal processing chain is twice this value as defined in the expression (6.1).

$$FS = 2 \cdot 255 = 510 \text{ Hz, for a measurement range of } [0; 255] \text{ Hz} \quad (6.1)$$

6.3 System parameters

The first stage of the signal processing chain diagram (6.5) is composed of three filters; an analog one and two digital ones. In order to find out for each of them their right order and coefficients, it is necessary to analyze the frequency components of one raw ECG signal. A *spectrogram analysis* of such a signal could be done, but it would have the disadvantage that the localization of events would be either accurate in the *time domain* or in the *frequency domain*, but not in both domains at the same time. A spectrogram does not offer good time-localization of frequency components. To overcome this drawback, for real world signals like an ECG one, the wavelet transform can be used. It offers superior time-localization of frequency components and allows an easier extraction of particular features in the signal. By using the *MATLAB Wavelet Toolbox* it is straight forward to compute a continuous wavelet transform onto an unidimensional sampled signal and get the three dimensional representation of the wavelet spectrogram (6.6).

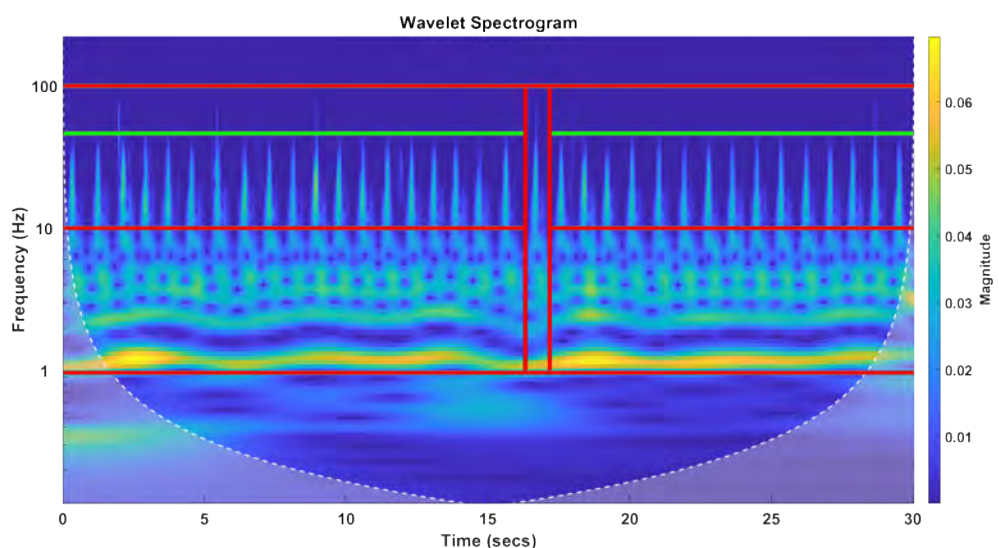


Figure 6.6: Wavelet spectrogram of one of the ten recorded ECG signal

The resolution of the wavelet transform (6.6) allows to precisely locate events in time and frequency domains like the one encompassed by the red vertical rectangle. Simply by counting the events in the wavelet spectrogram (35 in this case) and by

multiplying it by 2 as the measurement's duration is of 30 seconds, it can be found that, for this particular signal, there is a heart rate of 70 BPM.

The green line that has been added to the wavelet spectrogram shows the average maximal frequency of events which are at 40 Hz. This is the cut-off frequency that can be used for the pre-emphasis analog filter to increase the amplitude of the frequency components up-to 40 Hz. Moreover, red separator lines have also been drawn to separate the low-level frequency components of the ECG that are between 1 and 10 Hz from the high-level frequency ones that go from 10 up-to 100 Hz.

As mentioned in §6.2.1, the goal is to extract the high frequency peaks and then determine the distance between all consecutive ones. Therefore, these values of 10 and 100 Hz can be used as the two cut-off frequencies for the digital band-pass filter. The digital low-pass filter represented in the signal processing chain (6.5) behaves as an integrator over the pre-processed signal; it extracts the envelope of its input signal. By looking at the wavelet spectrogram, it can be seen that the **R peaks** events starting frequency is around 8 Hz. This indicates that the cut-off frequency of the digital low-pass filter can be set at 10 Hz.

The table 6.1 summarizes the filters parameters to use in the signal processing chain.

Filter type	Cut-off frequency 1	Cut-off frequency 2	Filter order
Pre-emphasis analog	40 Hz	-	3
Band-pass digital	10 Hz	100 Hz	4
Low-pass digital	10 Hz	-	4

Table 6.1: Filters parameters for the signal processing chain

The order of the filters, and the given data sampling rate FS , determine the required processing power. For such ECG signals, filters of fourth order are a very good compromise between quality and performance. The order of the pre-emphasis analog filter is of 3, because the data acquisition board mentioned in §4.2.1 has been designed this way.

For the second stage of the signal processing chain represented in the diagram (6.5) there is basically one parameter to look at that needs to be tuned correctly. This is the threshold that will separate global from local maxima. At this point, it is difficult to guess what is its correct value, but this is discussed in §6.4.

6.4 First investigations in MATLAB

In this section, the goal is to quickly check the feasibility of the sketched algorithm (6.5) by using MATLAB. The reference measurements acquired from the smart watch and transferred to a CSV file as described in the figure (6.4) are brought into MATLAB, converted into usable values/numbers, passed through the full signal processing chain and the provided results are analyzed.

6.4.1 Import and convert measured data

The data are organized as a matrix of five columns and N lines. The goal is to create a table with column headers and data out of it as shown in the figure (6.7).

Moreover, numerical data have to be converted into floating point double precision data type so that mathematical operations can be applied on them.

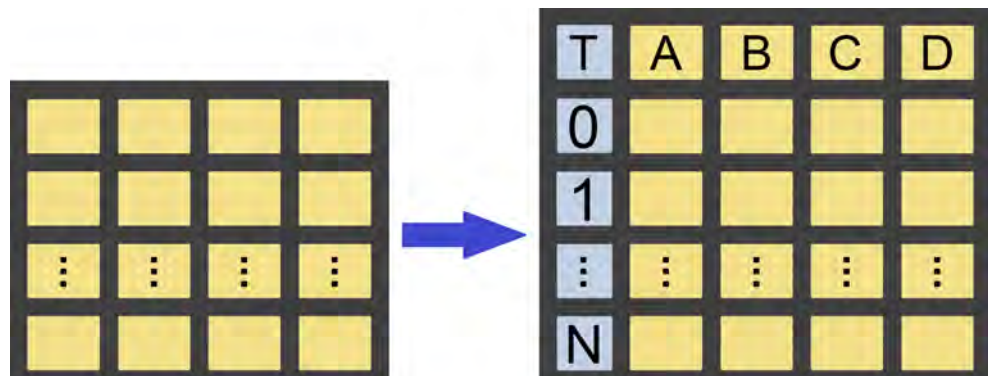


Figure 6.7: Bringing external CSV data into MATLAB as a table

The easiest way to get the data in the format described by the figure (6.7) is to use the MATLAB function named **readtable(filename,opts)**. It has two input arguments; the first one which is the name of the file to import data from, like 'Measurements.csv' in this case, and the second one for which data import options can be specified, like which columns to select or the data type of each column's values for example. Once the import process is complete, it results in the table (6.2). At this point, users can select which row, corresponding to a specific ECG signal, they would like to process.

<code>signalId</code>	<code>signalDate</code>	<code>signalValues</code>
1	2020-11-01	[1×15328 double]
2	2020-11-04	[1×15328 double]
3	2020-11-07	[1×15328 double]
4	2020-11-10	[1×15328 double]
5	2020-11-13	[1×15328 double]
6	2020-11-16	[1×15328 double]
7	2020-11-19	[1×15328 double]
8	2020-11-22	[1×15328 double]
9	2020-11-25	[1×15328 double]
10	2020-11-28	[1×15328 double]

Table 6.2: Table of measured raw ECG data in MATLAB

By looking at the size of a signal in the table (6.2) there are 15'328 data samples. The number of data samples can be divided by the sampling frequency of the digital signal processing chain which is of 510 Hz and it gives a measurement duration of 30 seconds. This confirms that, from the sampling point of view, these ten ECG signals can be used as reference ones to test the designed signal processing algorithm (6.5).

To illustrate this, the real raw ECG signal #6 (6.8) is used as the input signal. Its evolution after each subsystem is shown from here onwards. As in the theoretical ECG representation (6.1), the ECG wave is made of low and high frequency components that can be clearly identified.

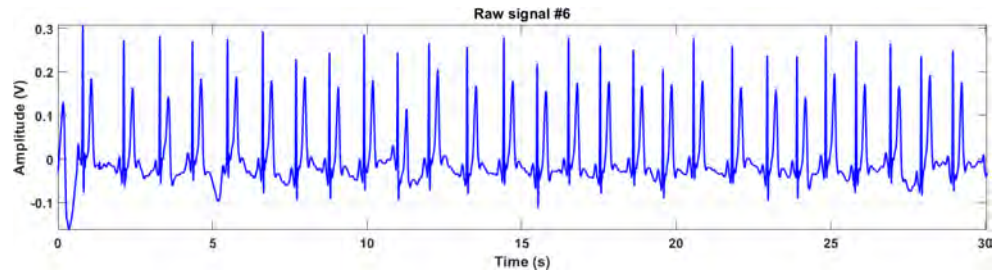


Figure 6.8: Real raw ECG signal #6

6.4.2 Pre-processing stage

At this point, the acquired data can flow through the first stage of the signal processing chain represented in the diagram (6.5). It consists of successively applying three filters on the provided ECG signal.

1) Pre-emphasis analog Bessel filter

This filter acts as a low-pass one increasing the amplitude of desired low frequency components to reduce the effect of potential unwanted higher ones. As represented in the Bessel Bode diagram of amplitude (6.9), low frequencies are amplified (when the frequency response of the filter has a dB gain higher than 0 dB) from 0 to 54 Hz. In §4.2.1, it is mentioned that the cutoff frequency of the Bessel filter is of 40 Hz, but such a filter is less selective than a Butterworth's one. That is why there is still some amplification up to 54 Hz. However, such a Bessel filter has the advantage of having a constant group delay within its bandwidth. In other words, all frequencies in its bandwidth go through the filter at the same time. By comparing the signal being output from this filter with the raw ECG input signal (6.8), it is clearly visible that it has been amplified by at least a factor of 3. As an example, the low frequency peak located around 13.5 seconds has an amplitude of 0.6 V in the pre-emphasized signal (6.9). On the other end, the similar peak in the raw signal (6.8) has an amplitude of 0.17 V. The computed ratio between both amplitude is $0.60/0.17=3.53$ which corresponds to the gain value of this filter, as mentioned in §4.2.1, which is of 3.56.

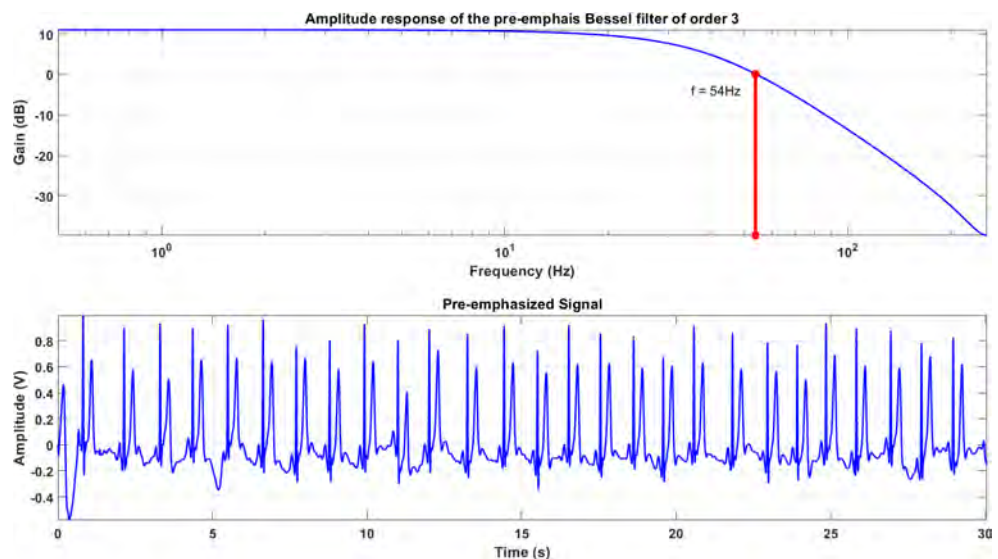


Figure 6.9: Bessel Bode diagram of amplitude and pre-emphasized signal

2) Band-pass digital Butterworth filter

This second filter keeps frequency components between 10 and 100 Hz, and attenuates all the others. At the same time, right after it, a rectifier block is there to compute the absolute value of the filtered signal to only have positive or equal to zero values as shown in the Bandpass Butterworth Bode diagram of amplitude and rectified signal graph (6.10).

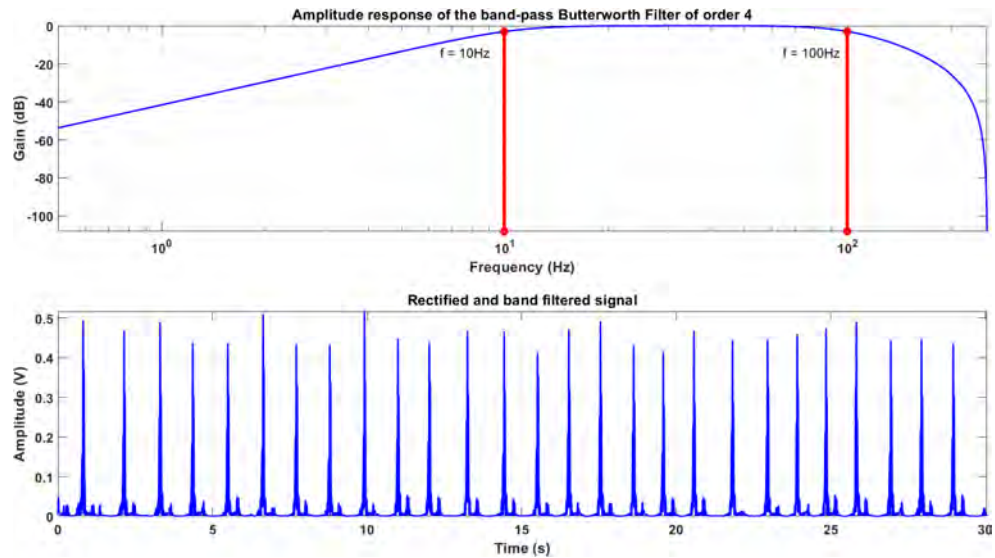


Figure 6.10: Band-pass Butterworth Bode diagram of amplitude and rectified signal

3) Low-pass digital Butterworth filter

This filter integrates over the remaining signal to nicely extract its envelope. The final result of this filtering process is shown in the figure (6.11). The comparison between this envelope and the original raw signal (6.8) shows that the overall amplitude remained the same, but the peaks are now much easier to identify in order to extract the needed global maxima.

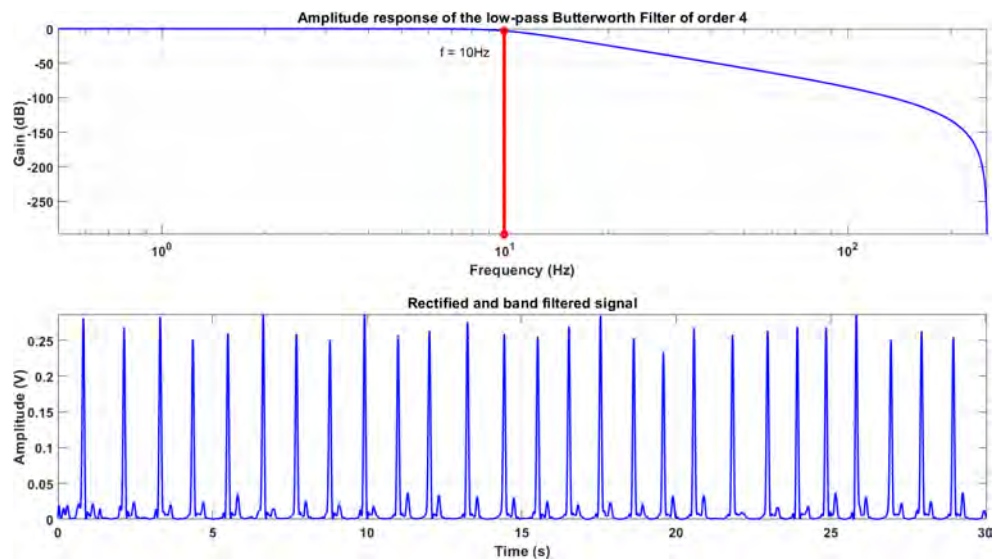


Figure 6.11: Low-pass Butterworth Bode diagram of amplitude and envelope of the signal

6.4.3 Post-processing stage

Once the ECG signal has been processed and transformed into a clean envelope, the goal is to identify all its maxima. Basically, a peak is defined as a data sample that is either larger than its two neighboring samples or is equal to infinite. An easy way to extract peaks from a signal or data vector is to use the function from the *Signal Processing Toolbox* in MATLAB named **findpeaks()**. It has a variable number of input arguments. Its easier implementation is when it is called with only one input argument; a vector of data. In this case, it will find all the local maxima as shown on top of the figure (6.12). The optimal solution is to call the function with a second input argument called 'MinPeakHeight' which allows to set a threshold value to only extract values that are above it. This way, all global maxima stand out of this analysis as illustrated at the bottom of the figure (6.12). For this ECG signal processing application, a threshold value of 0.15 V, represented by the red line in the bottom graph of (6.12), is the one providing the best results.

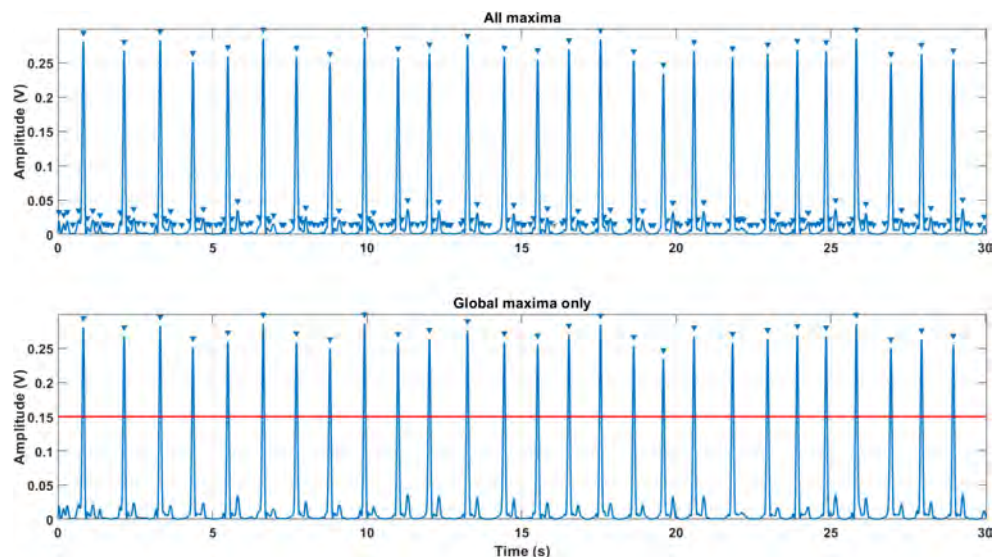


Figure 6.12: Extraction of maxima without and with threshold

At that point, the final step is to calculate the difference in seconds between each pair of consecutive global maxima and to either extract the median value or calculate the mean value among all distances. In MATLAB, the difference between pairs of consecutive values within a vector can be easily found out by using the function **diff()** over the vector containing the time values of the global maxima occurrences. It will output a vector with one value less than the input one containing the time differences between each pair of consecutive peaks. The pulse's rate or distance is obtained by inverting each difference and calculating their average value to express the heart frequency in beats per second (BPS). Multiplying this value by 60 provides a heart rate in beats per minute (BPM). The variation or uncertainty among the final heart rate can be calculated by using the least mean square (LMS) approach. To illustrate the computational operations to realize to be able to calculate the final BPM value and its uncertainty, a short mathematical development follows.

For a vector x of N elements representing the extracted global maxima, the distance Δ between each pair of consecutive maxima is given by the expression (6.2).

$$\Delta(n) = \frac{1}{[x(n+1) - x(n)]}, \forall n \in [1; N] \quad (6.2)$$

The mean heart rate value in BPS μ is given by the expression (6.3).

$$\mu = \frac{1}{N-1} \sum_{n=1}^{N-1} \Delta(n) \quad (6.3)$$

The variance on distance values σ^2 is given by the expression (6.4)

$$\sigma^2 = \frac{1}{N-2} \sum_{n=1}^{N-1} |\Delta(n) - \mu|^2 \quad (6.4)$$

Finally, the BPM value μ_{BPM} and its standard deviation σ_{BPM} , both rounded upwards to the next integer value ($\lceil v \rceil$ with v a real value) are given by their corresponding expressions in (6.5).

$$\begin{aligned} \mu_{BPM} &= 60 \cdot \lceil \mu \rceil \\ \sigma_{BPM} &= 60 \cdot \lceil \sqrt{\sigma^2} \rceil \end{aligned} \quad (6.5)$$

In the BPM results of the ECG signal #6 (6.13), a summary of the obtained results is given. Each ECG pulse events are highlighted in the spectrogram on top. The original and intermediate signals as well as the detected peaks are represented at the bottom. The end result for the ECG signal #6 is of 56 ± 5 BPM. By counting the events (in yellow) from the spectrogram, there are 28 of them for 30 seconds. For one minute, this gives $2 \cdot 28 = 56$ BPM, which confirms that the output result from the algorithm is correct. The signal processing chain represented in the diagram (6.5) has been tested with other ECG signals as well and has been proven to be very robust. A deeper discussion on the numerical results is provided in §9.1.

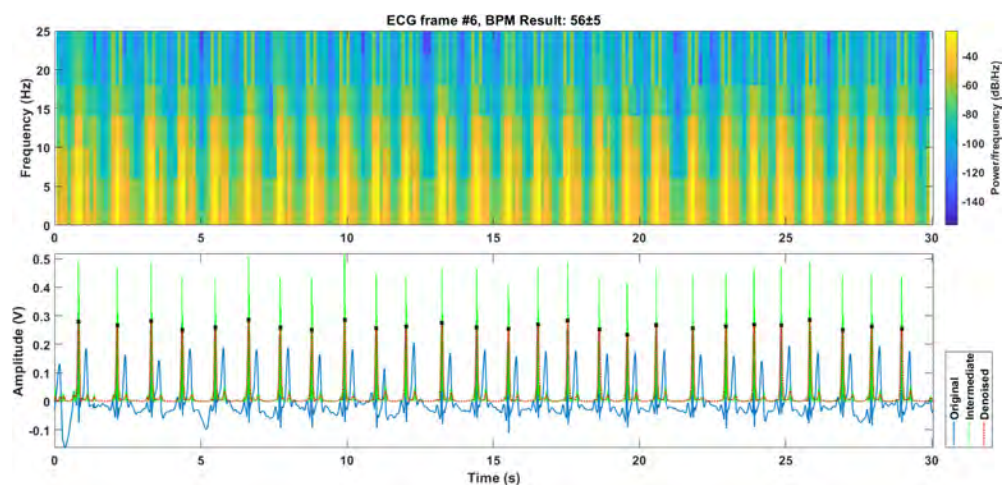


Figure 6.13: BPM results of the ECG signal #6

6.5 Simulation model in Simulink

A feasibility analysis of the sketched algorithm (6.5) has been verified in §6.4 by using MATLAB. The next step to realize is to create its Model-Based Design (MBD) implementation. This is the intermediate step before reusing the developed components to generate C code automatically out of them and deploying the application onto the real-time embedded system described in §4.

In the simulation model (6.14), the full algorithm is implemented. Only the access to the hardware peripherals like the analog input as well as the digital inputs and outputs is not. However, their behavior can be kind of emulated in the model which makes it very useful to debug it early in the development process as shown in §5.1. At this point, the focus is to model the system and its subsystems following the requirements and to test that all of them do work as expected. Indeed, modeling and simulation are key development steps requested by the IEC 62304 standard mentioned in §2.2.

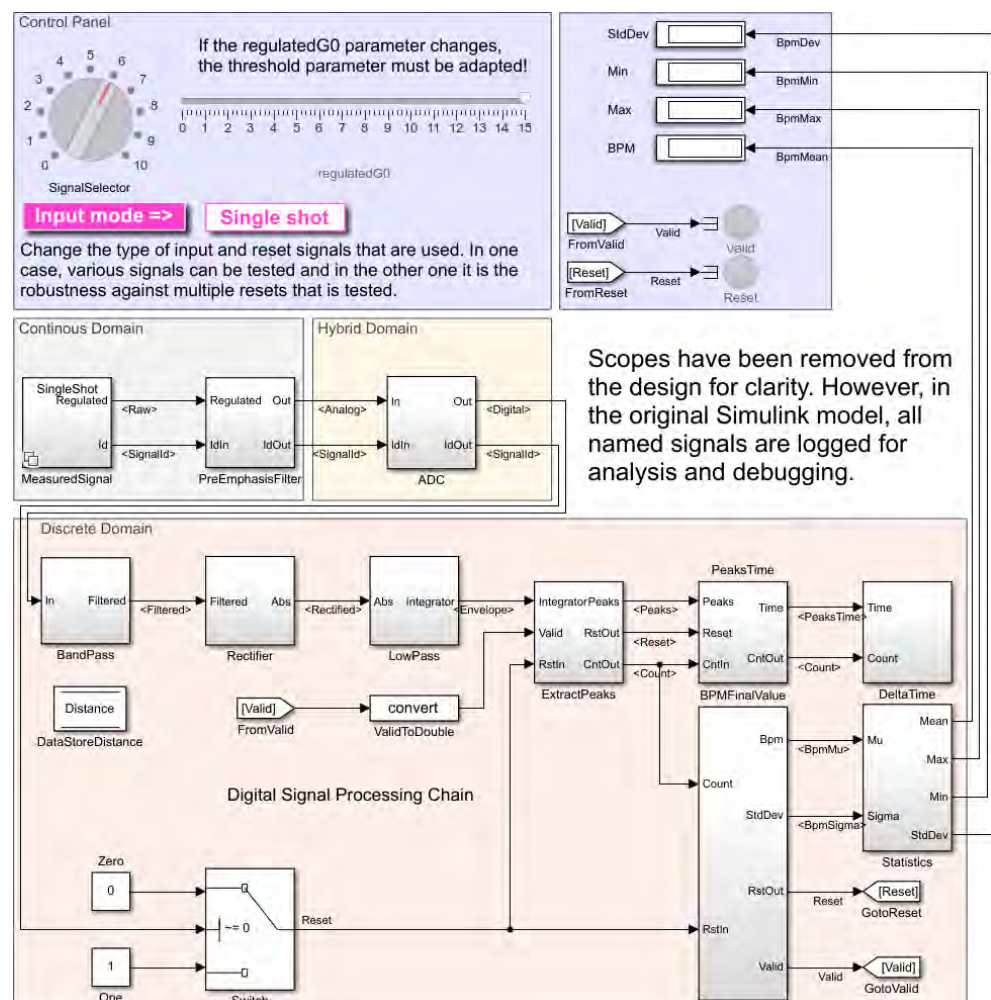


Figure 6.14: Simulation model of the signal processing algorithm in Simulink

Remark: the requirements used to realize the simulation model (6.14) comes from the §6.4. They have not been authored in a formal way, as this is not the focus of this Master thesis and it would have required extra effort to do it.

The Simulink model (6.14) is organized into three main parts:

- 1) The **violet area**, on top, represents the control panel allowing to select which mode of simulation to run. The first mode called “Single shot” provides one ECG signal only for 30 seconds as raw input. The signal number can be selected between 1 and 10 by using the “SignalSelector” knob. The 0 value means that a reset of the running algorithm is requested. The second mode of simulation called “Robustness” runs all ten input signals one after the other with a 15 seconds reset between each of them; running the simulation for a total of 460 seconds. This mode is especially useful to check that when a reset occurs, all subsystems having internal states, like delays for example are reset correctly so that the next measurement is not compromised. On the right-hand side of the top area, displays and LEDs are present to monitor on the fly needed values and flags
- 2) The area in the middle is split into two parts. On the left-hand side, the **gray area** represents the analog part with the data acquisition board containing the Bessel filter (4.6). In the **yellow area** there is the analog-to-digital converter (ADC) making the transition from the continuous to the discrete time domain. Here a key feature of Simulink is used; *multirate simulation*. Indeed, Simulink can handle multiple discrete sampling rates and continuous time in the same simulation, which is very useful for simulating multidomain physical systems
- 3) The **pink area**, at the bottom, corresponds to the part of the signal processing chain that is deployed onto the real-time embedded system; it is made of discrete time subsystems. By looking at the sketched diagram (6.5), it is straight forward to observe that there is a one-to-one mapping of the subsystems from one representation to the other. The pre-processing stage after the ADC is there with the band-pass filter, the rectifier and the low-pass filter. The post-processing stage is also present with the peaks extraction, the peaks time and delta time, and the computation of the needed results (with the subsystem called “BPMFinalValues”). A description of each subsystem’s contents from the discrete area is provided in this section, except for the “Statistics” one that only computes the BPM minimum and maximum values by respectively subtracting and adding the standard deviation to the BPM mean value

The blocks used in the Simulink model (6.14) to compose each subsystem come from the basic Simulink library for the standard mathematical operations, as well as from the *DSP System Toolbox* library for designing the filters and counters. Each one of these subsystems are designed using the same definitions and parameters as described in §6.4.2. All this allows to speedup the development of the application.

6.5.1 Band-pass Butterworth filter

A “Bandpass Filter” block is directly available allowing to design a Butterworth band-pass filter of fourth order. A band-pass filter’s mask (6.15) is provided to the user to enter the desired parameters. The rectifier subsystem located right after this band-pass filter is simply computing the mathematical function **abs()** to only have positive values at its output.

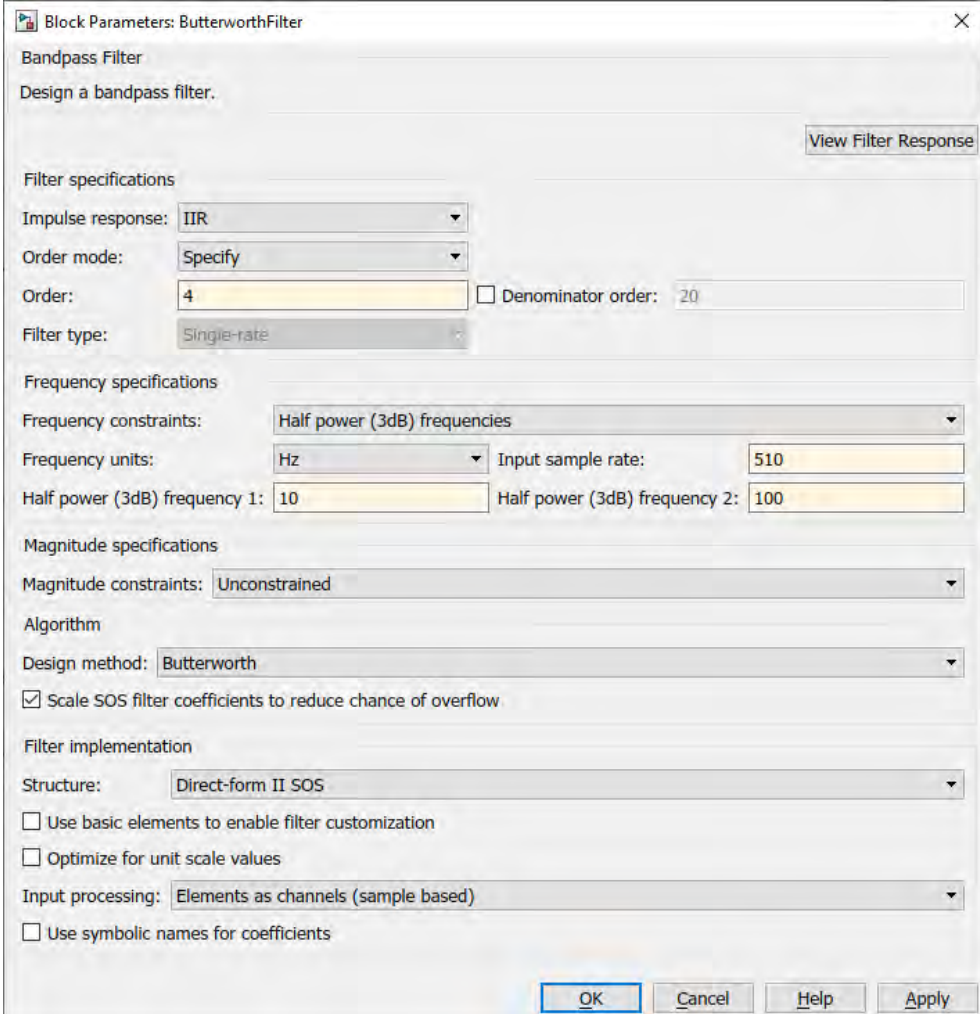


Figure 6.15: Mask to set the band-pass filter's parameters

The frequency response of the band-pass filter (6.16) is exactly the same as the one shown by the band-pass Butterworth Bode diagram of amplitude (6.10).

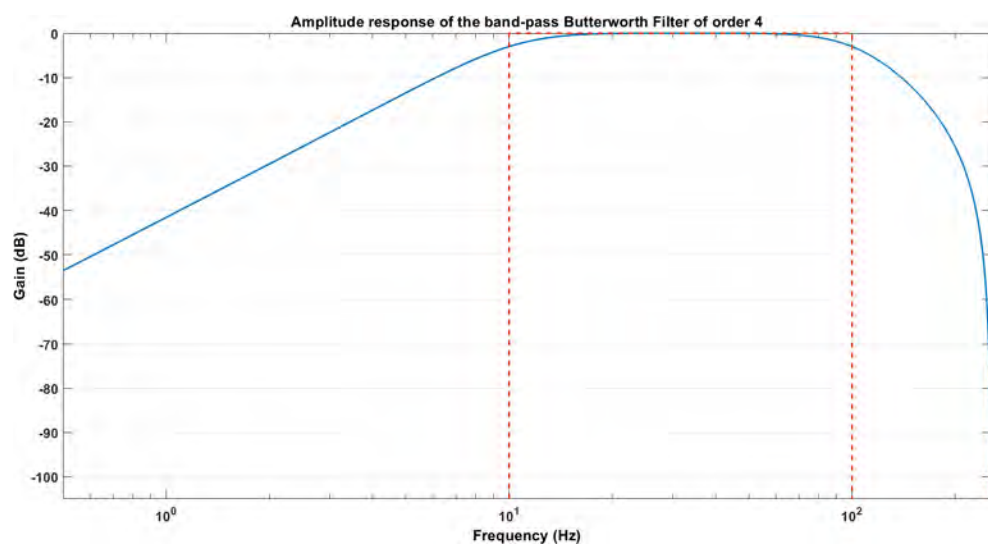


Figure 6.16: Amplitude response of the Butterworth band-pass filter

6.5.2 Low-pass Butterworth filter

Another way to design filters is to use the “Digital Filter Designer” block. The user-friendly interface (6.17) is provided to enter the desired parameters and directly visualize the results via various representations.

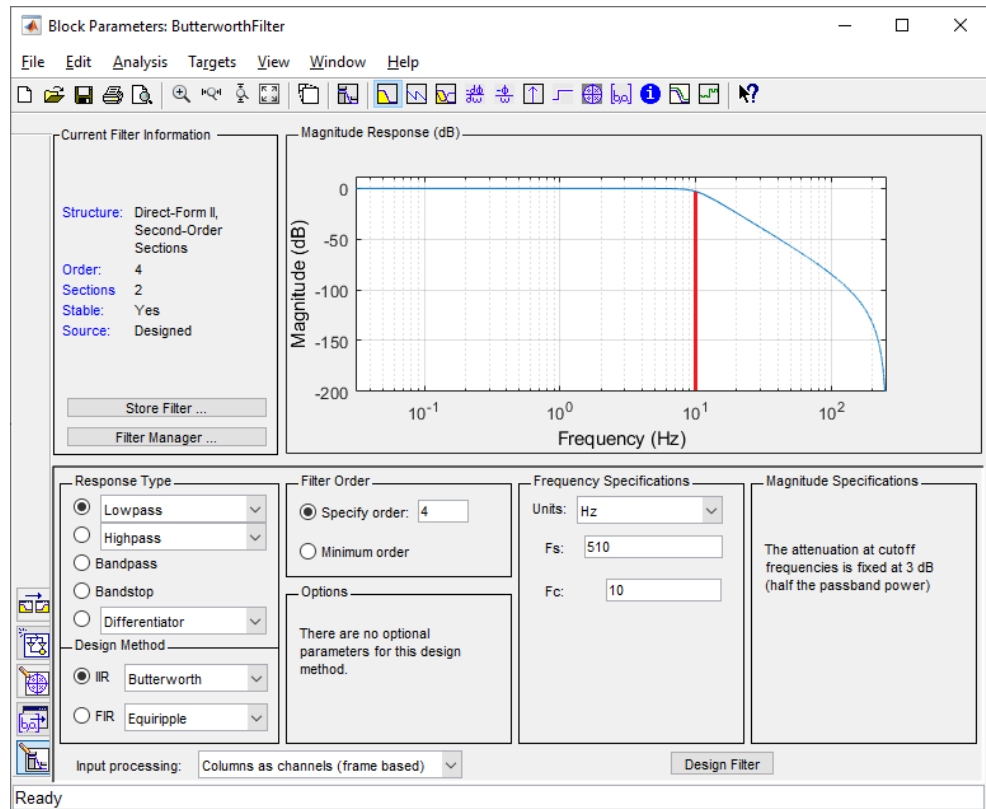


Figure 6.17: Interface to parametrize and visualize the designed low-pass filter

The frequency response of the low-pass filter in the interface (6.17) is exactly the same as the one shown by the low-pass Butterworth Bode diagram of amplitude (6.11).

6.5.3 Peaks extraction

The model of the peaks extraction subsystem (6.18) implements a similar functionality as the `findpeaks()` function described in §6.4.3.

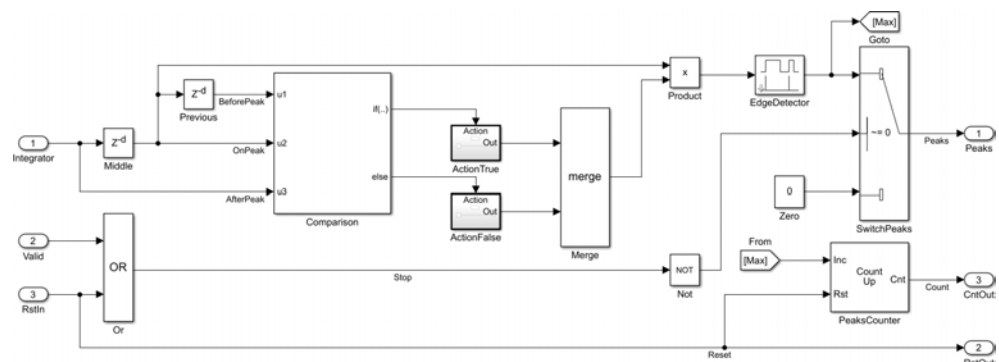


Figure 6.18: Peaks extraction subsystem based on the `findpeaks()` function

The peaks extraction subsystem (6.18) is made of three main sections:

1) Comparison

As mentioned in §6.4.3, a peak is defined as a data sample that is larger than its two neighboring samples. The “comparison” block has three inputs: $u1$ which contains the sample before the local maximum, $u2$ which represents the local maximum, and $u3$ which contains the sample after the local maximum. The two delay blocks Z^{-d} called “Previous” and “Middle” have both a delay of 60 samples. Indeed, the maximal number of peaks that can be measured by the application is of 255 for one minute. Therefore, the minimal possible number of samples between two consecutive peaks is given by the operation $(60/255) \cdot 510 = 120$. That is why the parameter $d = 60$ in each delay block Z^{-d} . The total delay of these two blocks gives then 120 samples. This construction allows to avoid the detection of peaks that would be too close to each other and impossible physically. The logical expression that evaluates the three inputs is: $(u3 < u2) \& (u2 > u1) \& (u2 > threshold)$, with $threshold$ having the value defined in §6.4.3 and allowing to only output identified global maxima

2) Edge detection

The “EdgeDetector” block outputs a Boolean pulse each time a global maxima is at its input. This allows to produce peaks having the same shape and focusing on the timing aspect only

3) Counting peaks

The “PeaksCounter” simply increments its value by one every time the “EdgeDetector” outputs a Boolean pulse. It is reset to zero when a user reset is requested. Its maximum possible value is of 255, that is why it is an 8bit counter. The output value provided by the counter is reused later to compute the BPM mean and standard deviation values over time

6.5.4 Peaks time

The “PeaksTime” subsystem (6.19) outputs a time tag every time a peak’s pulse comes in.

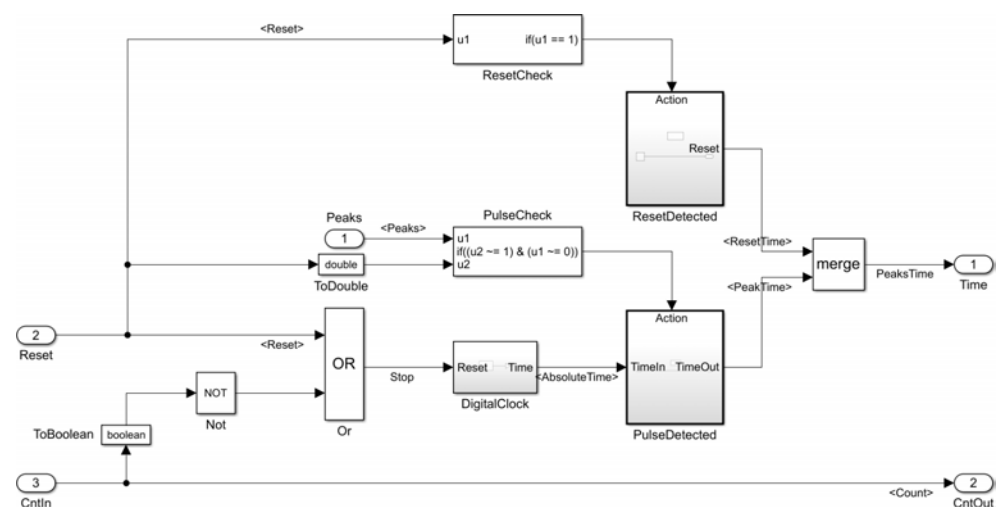


Figure 6.19: Peaks time subsystem providing the time occurrences of peaks

The “DigitalClock” subsystem contains a similar counter block than the one used in the peaks extraction subsystem (6.18). The only difference is that it counts at a resolution that is equal to the rate of the sampling frequency ($1/510$) up to a maximum duration of 30 seconds. That means its maximum value is of $30 \cdot 510 = 15'300$ which is smaller than $2^{14} = 16'384$ and is therefore a 14bit counter. Its output is then normalized by the sampling frequency to produce an absolute time in seconds.

6.5.5 Delta time

The “DeltaTime” subsystem (6.20) computes the difference between two consecutive peaks and save its invert value corresponding to the distance Δ as defined in the expression (6.2). This distance serves then as the basis to compute the BPM mean and standard deviation values.

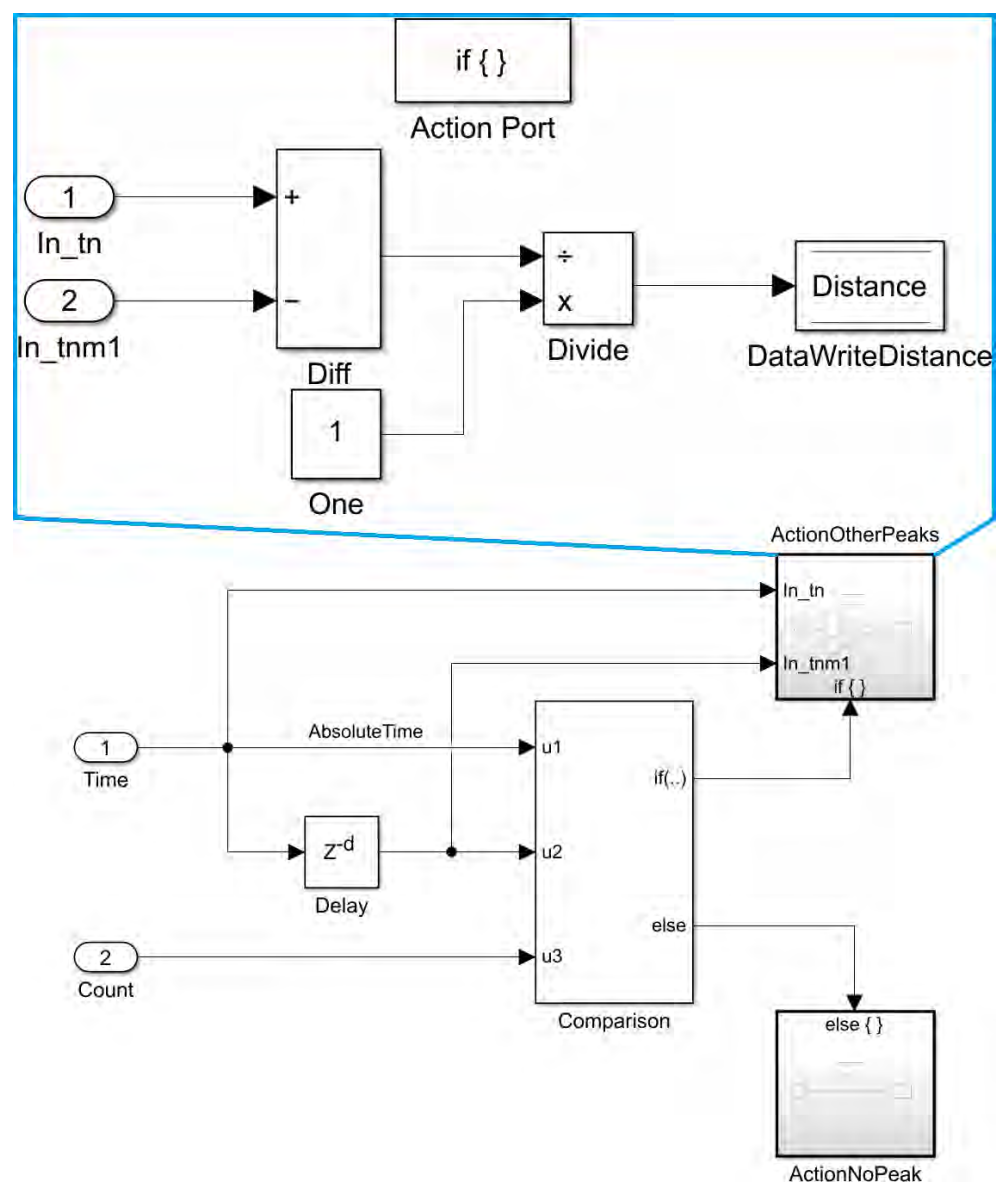


Figure 6.20: Delta time subsystem computing the Δ distance

6.5.6 BPM final value

The “BPMFinalValue” subsystem (6.21) computes the final BPM value and its standard deviation. Its top level decides if there is a need to compute the BPM and the standard deviation by using the logical expression $(u1 \neq 0) \& (u2 \neq 1)$, with $u1$ the peaks counter value and $u2$ the reset signal. Once this condition is fulfilled, the “IfPeakCompute” subsystem executes its BPM computing algorithm (6.22).

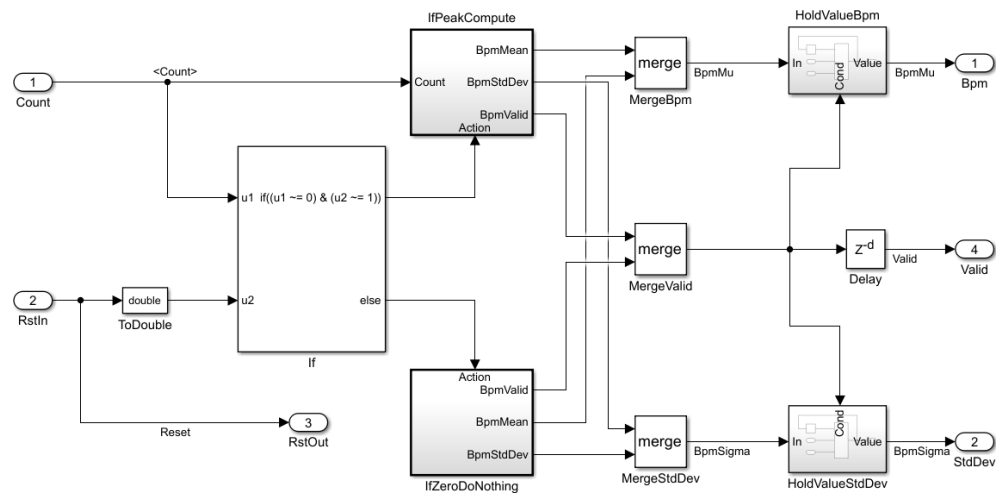


Figure 6.21: Top level of the subsystem computing the BPM final value

The “IfPeakCompute” subsystem (6.22) realizes the computational operation defined by the expressions (6.3). Indeed, the “Cummulator” block on top represents the sum over the time and the “Divide” block right after provides the running mean value μ . The obtained value is then converted and rounded (upwards to the next integer value) with respectively the “s2m” and “ceil” blocks. It finally outputs μ_{BPM} . The standard deviation is computed by the “LeastMeanSquare” subsystem (6.23).

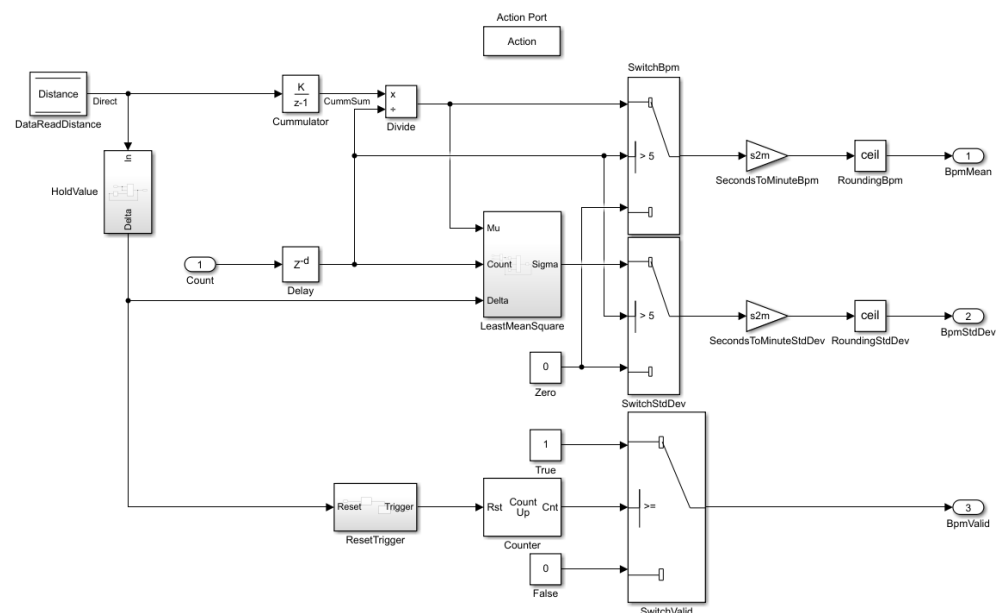


Figure 6.22: Subsystem computing the BPM value

The “LeastMeanSquare” subsystem (6.23) realizes the computational operation defined by the expressions (6.4) also including the square root operation at the end. Here again it is straight forward to follow the mathematical operations flow from the left to the right.

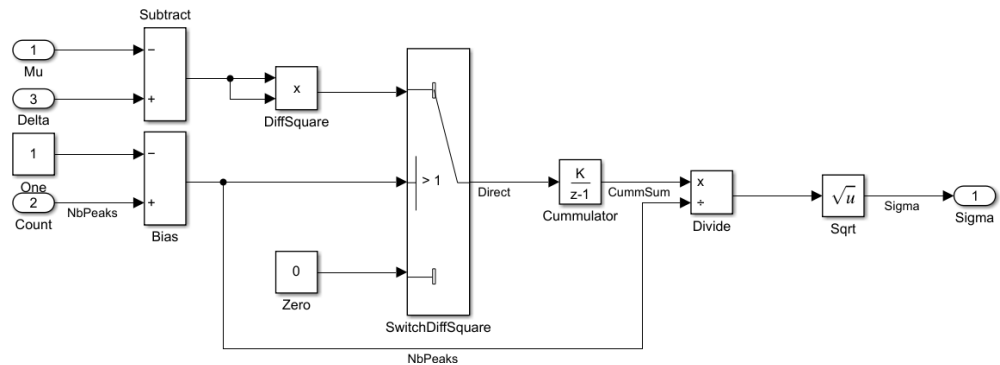


Figure 6.23: Subsystem computing the standard deviation

All the aforementioned algorithms for the post-processing stage, from §6.5.3 to §6.5.6 have been put together inside a library of common IPs (6.24). The goal is to re-use them as is for the deployment onto the embedded system at §6.6.

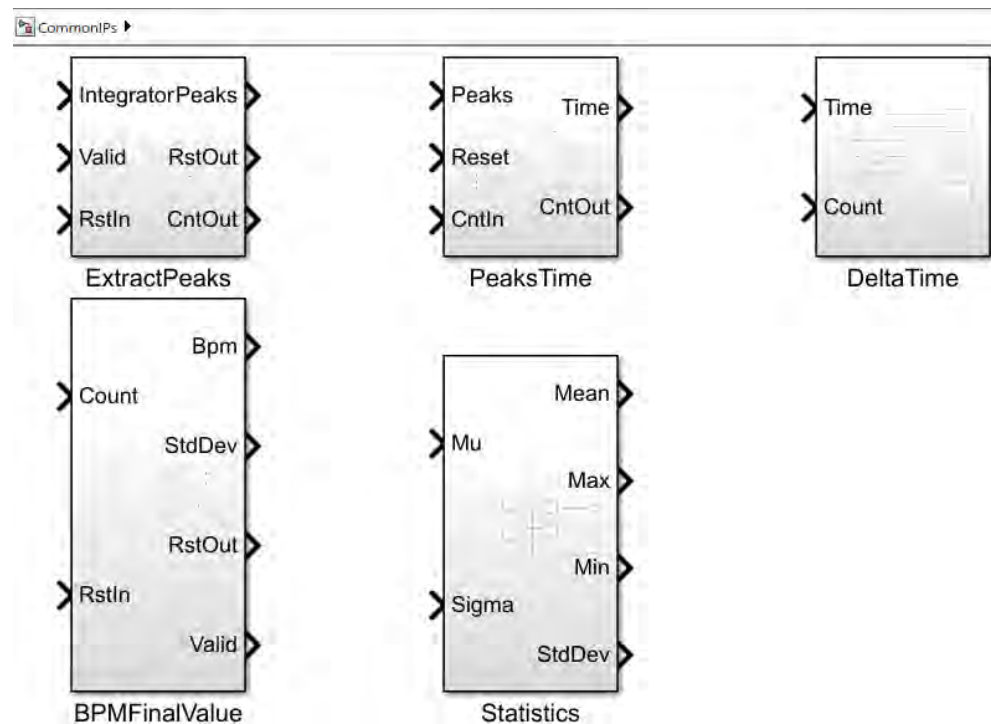


Figure 6.24: Library of common IP components for re-use

Remark: the two digital filters from the pre-processing part at §6.5.1 and §6.5.2 could have also been added to this library for re-use. However, in the *DSP System Toolbox* there are also filter blocks optimized for hardware implementation rather than for design. They can be directly parametrized with the a_k autoregressive and b_k moving-average coefficients and are oriented towards optimization for automatic C code generation. That is why these are used in the deployment part at §6.6.

6.6 Deployment models in Simulink

The MBD implementation of the signal processing chain (6.5) has been realized by using Simulink in §6.5. It has been verified by reusing the same ten test signals collected in MATLAB; a discussion on the obtained numerical results is done in §9.

At this point, the goal is to deploy the designed algorithm onto the embedded system and to connect it with the required hardware peripherals, that are: the ADC, the digital IO's, and the Ethernet port. Two types of deployment are done; the *External mode* or *Processor-In-the Loop (PIL)* and the *Standalone mode*.

6.6.1 Processor-In-the Loop verification

PIL is part of what are called *equivalence testing techniques* that encompass *SIL*, *PIL* and *HIL*. *SIL* and *HIL* stands respectively for *Software-In-the Loop* and *Hardware-In-the Loop*. In the IEC 62304 standard mentioned in §2.2, *SIL* and *HIL* are requested and are referred to as *back to back testing*. At the end, *SIL* and *PIL* are almost the same. In both cases, C code is automatically generated from the MBD model. The input stimuli and the output responses are produced and managed by the simulation environment. The only difference is that the algorithm to test runs onto the simulation computer in *SIL* mode whereas it runs directly onto the embedded target in *PIL* mode. This means that in the context of verification and validation according to the IEC 62304 standard, it is even “better” to do *PIL* rather than *SIL* verification only.

The Simulink model (6.25) is made of exactly the same discrete time subsystems for the post-processing stage than the ones used in §6.5; they are re-used from the common IP library (6.24). As the model runs in external mode, it is possible to add and connect signals to viewers, like the dashboard scope, the three LEDs, and the four displays at the bottom; this allows to monitor and visualize data coming in from the board in real-time. The reset signal is controlled by a virtual switch in the model.

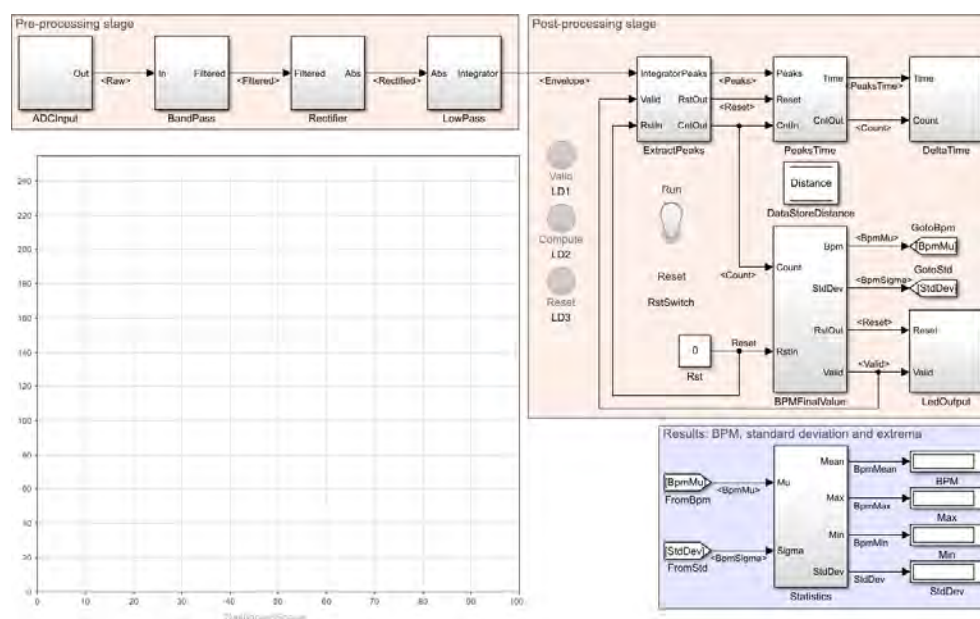


Figure 6.25: External mode model for PIL equivalence testing

The two filter subsystems from the pre-processing stage of the PIL model (6.25) have respectively the same frequency responses than in the Bode diagrams (6.16) and (6.17). Their implementations is simply focusing on their a_k and b_k coefficients. Indeed, these general *infinite impulse response* (IIR) filters are modeled using biquadratic structures, also called *second-order sections* (SOS). In general, biquadratic implementations of IIR filters are often preferred due to their desirable numeric properties, and it is straight forward to generate C code out of these.

Filter section	b_k	a_k	g_k
Band-pass digital IIR			
Section I	[1 0 -1]	[1 -1.8291848605 0.8454493997]	[0.4119575908
Section II	[1 0 -1]	[1 -0.5037226932 0.2658440798]	0.4119575908 1]
Low-pass digital IIR			
Section I	[1 2 1]	[1 -1.8956917312 0.9101698451]	[0.0036195285
Section II	[1 2 1]	[1 -1.7824701008 0.7960834983]	0.0034033494 1]

Table 6.3: Filters coefficients of the Butterworth band-pass and low-pass IIR filters

The filters coefficients table (6.3) provides the values of the a_k and b_k coefficients for the Butterworth band-pass and low-pass IIR filters. It also provides the g_k coefficients that are the scale gain values between each section and also at the beginning and the end of a filter as shown in the SOS diagram (6.26).

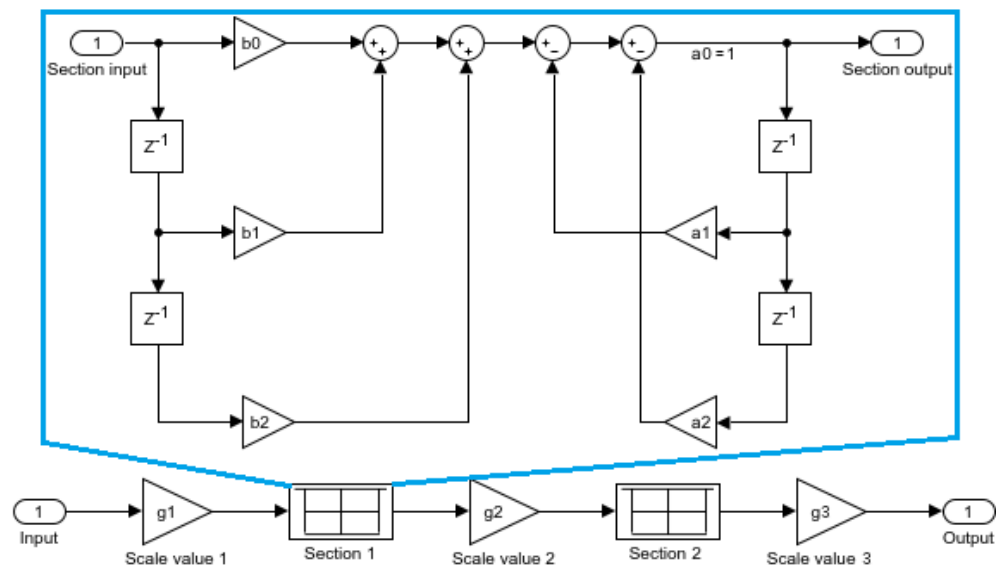


Figure 6.26: SOS diagram with coefficients for an IIR filter of fourth order

Two subsystems have been added to the Simulink model (6.25); the “ADCinput” at the beginning and the “LedOutput” at the end. A virtual “Reset” switch has also been added, but this one does not physically interact with the hardware yet; only with the processor. This is a signal controlled by the simulation environment in external mode via the serial connection. The “ADCinput” and “LedOutput” subsystems contain the drivers code for the interactions with the peripherals (using the Mbed hardware abstraction layer (HAL) or not) onto the hardware and are detailed in §7.

6.6.2 External mode

The external mode is used for rapid prototyping and to validate the generated algorithm code by enabling parameter tuning and signal monitoring. It establishes a communication channel between the simulation environment Simulink on the desktop computer, also called *host*, and the *target hardware* that runs the executable file created by the code generation and build process.

The communication diagram (6.27) shows that through the communication channel, it is possible to modify or tune desired block parameters in real-time. When some of them are changed on the host computer, Simulink downloads the new values to the executing target application via the *XCP¹ Master*. From the target running the application to the host it is possible to monitor and save signal data. At the bottom, the low-level transport layer of the channel manages the physical transmission of messages. The simulation environment and the generated C code running on the target are independent of this transport layer that formats, sends, and receives messages and data packets. In this project, the external mode is handled via a serial connection between the host and the target by using a micro-USB cable. It would also be possible to setup the XCP connection via Ethernet.

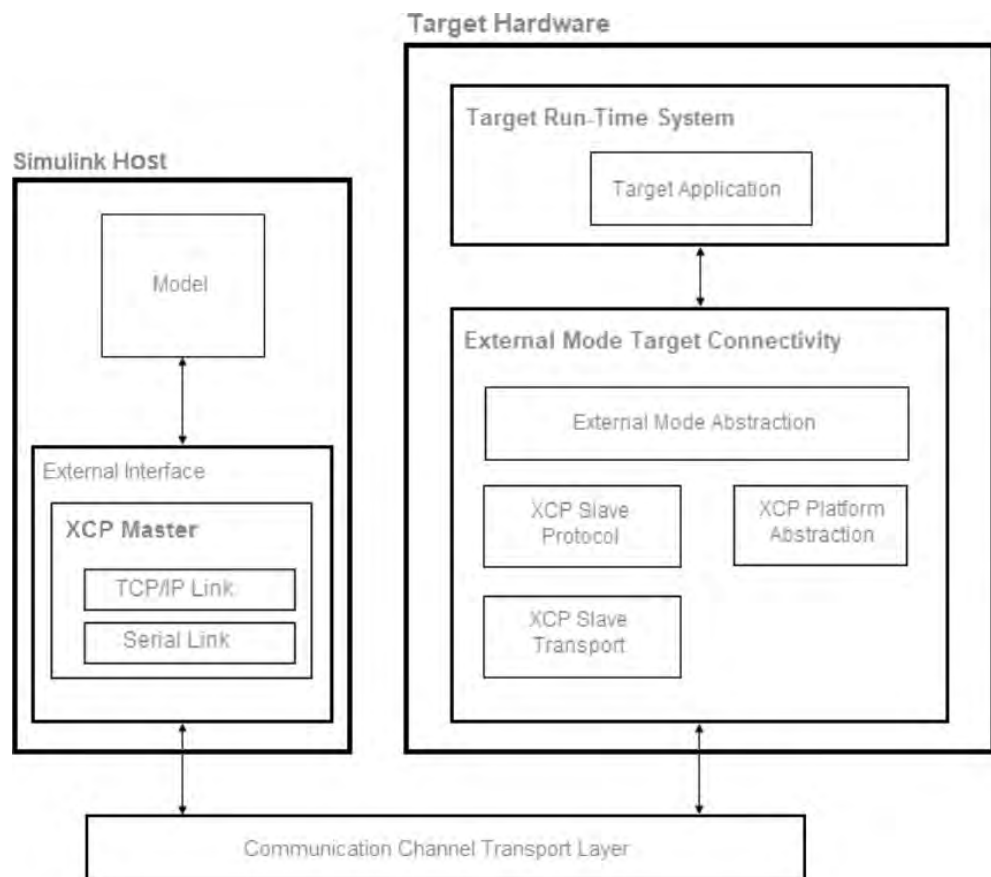


Figure 6.27: Host-target communication with external mode XCP

¹XCP = Universal Measurement and Calibration Protocol. It is a network protocol originating from [ASAM](#), in the automotive industry, for connecting calibration systems to electronic control units (ECUs)

The configuration of the external mode is shown in the hardware settings of the Simulink model (6.28). This capability is provided by the hardware support package (HSP) for the STM32 board introduced in §4.1. The protocol of the communication interface is set to “Serial” with the corresponding serial port onto which the board is connected, like “COM3” in this case.

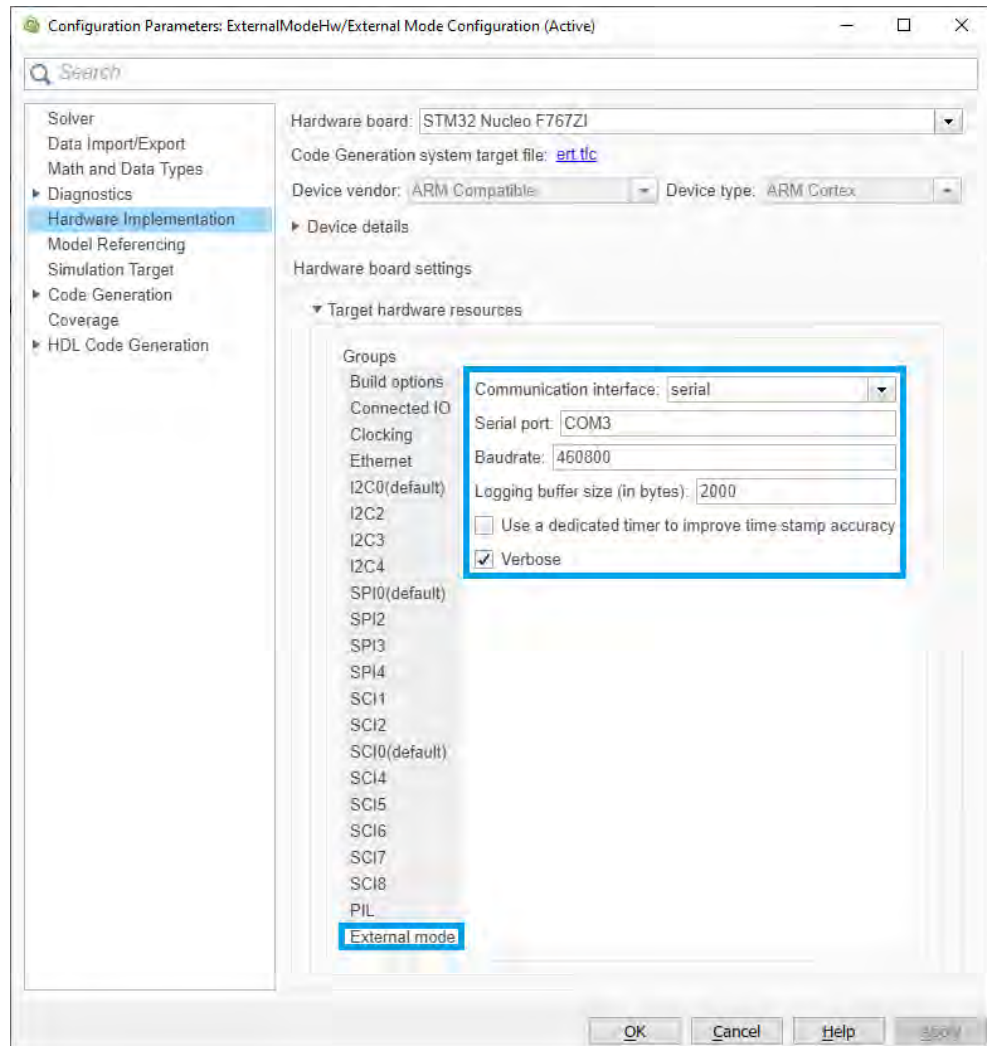


Figure 6.28: Configuration parameters of the external mode using serial XCP

Based on the number of data to send from the target to the host, the “Baudrate” and “Logging buffer size” can be adjusted. In the external mode model (6.25), ten double precision signals and four Boolean signals are monitored. As a Boolean is coded on one Byte, and a double on eight Bytes, the expression (6.6) gives the smallest baud rate to use to correctly send the needed signals data.

$$\begin{aligned}
 \text{BaudRate} &= (\text{NbDoubleSignals} \cdot 8 + \text{NbBooleanSignals} \cdot 1) \cdot \text{ByteSize} \cdot f_s \\
 &= (10 \cdot 8 + 4 \cdot 1) \cdot 8 \cdot 510 = 342'720 \text{ (bits/second)} \quad (6.6)
 \end{aligned}$$

The next pre-defined standard baud rate value from the STM table (6.4) is of 460'800 (bits/second) and is then used on both sides of the serial connection.

Desired baud rate (Bps)	Actual baud rate (Bps)	BRR	Error
9600	9600	AFC4	0.00000000
19200	19200	57E2	0.00000000
38400	38400	2BF1	0.00000000
57600	57600	1D46	0.00000000
115200	115200	EA3	0.00000000
230400	230400	751	0.00000000
460800	461538.461	3A4	0.160256293
921600	923076.923	1D2	0.001602564
13500000	13500000.000	20	0.00000000
27000000	27000000.000	10	0.00000000

Table 6.4: STM32F76xxx advanced Arm-based 32-bit MCUs programmed baud rates [17]

The value of “Logging buffer size” is by default at 2000 bytes which is fine for this application. Once these parameters are set, the user just needs to connect the board to the host PC and clicks on the “Monitor & Tune” button in the “Run on Hardware” section on top of the Simulink model.

6.6.3 Standalone deployment

At this point, the algorithm given by the signal processing chain (6.5) has been verified and validated through simulation in Simulink, but also onto the hardware itself. The last step is to deploy it onto the hardware so that it can run on its own without being connected to any external environment; that is standalone deployment.

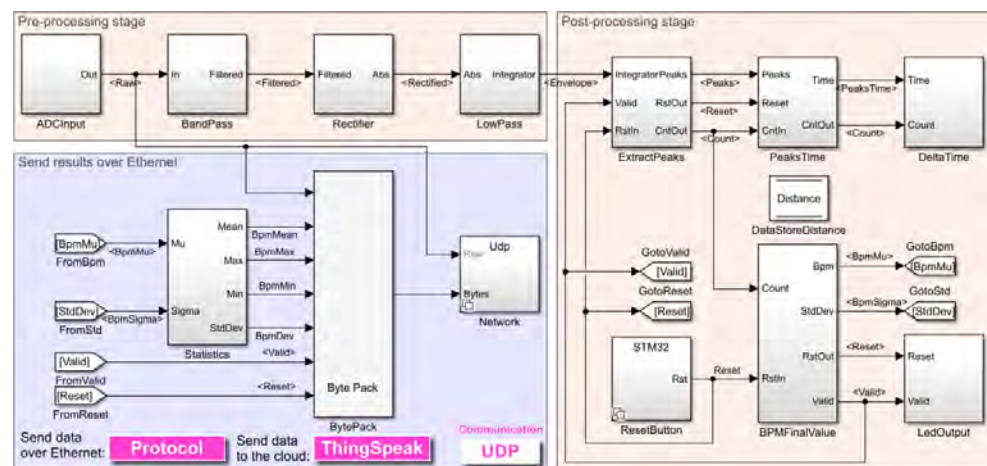


Figure 6.29: Standalone mode model for final deployment onto hardware

The standalone mode model (6.29) is made of exactly the same signal processing chain than in the external mode model (6.25). Only the dashboard components (LEDs, switch, and scope) have been removed and replaced with the real hardware peripherals drivers. Also, the physical data connection access has been added so that data can be sent over a local Ethernet network via either UDP or TCP/IP, or directly on the Internet up to the cloud for IoT communication.

7 Drivers

In this chapter, the focus is put on the subsystems that interact with the real hardware peripherals. These are:

- a push button connected to a digital input pin to allow the reset of the heart rate measurement
- three LEDs that provide the status of the current measurement
- an analog-to-digital converter (ADC) to convert the acquired electrocardiogram (ECG) signal
- the Ethernet port for UDP, TCP/IP, and IoT communication's protocols

For the three first types of peripherals, two drivers implementations are done. The first one, called "Stm32", consists of connecting the algorithm directly to the peripherals at the lowest possible level in the source code that is hardware dependent. The second one uses the hardware abstraction layer (HAL) called Mbed that has been introduced in §3.2.

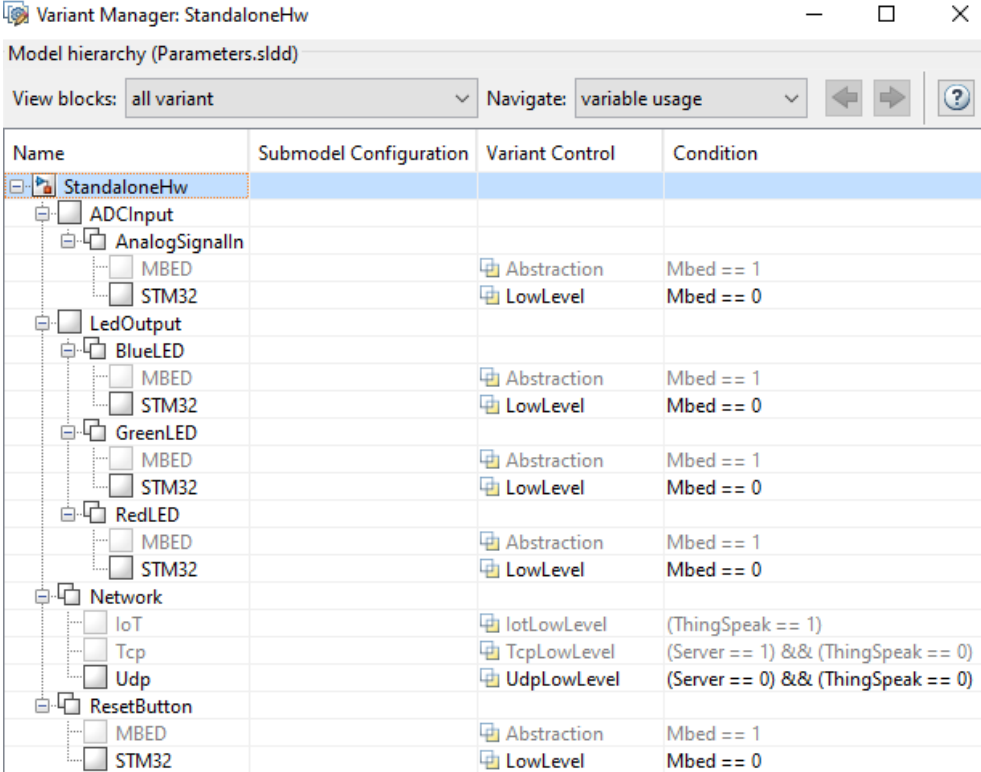
Regarding the Ethernet port, the three communication's protocols are implemented following the Stm32 approach. Their implementation using the Mbed HAL is not done, because they would need to interact with the Mbed operating system (OS), introduced in §3.1, that is not integrated with MATLAB and Simulink yet. Indeed, it would require too much time and effort; at least one man month, for someone used to such porting of OS to integrate it in the development environment. Nevertheless, this could be a topic for another Master thesis to continue and augment the work done in this one.

The handling of these two types of drivers implementations is done via *variants*. A variant subsystem contains several implementations of a type of algorithm. For example, the subsystem "Network" used in the Simulink model (6.29) is a variant subsystem implementing the UDP, TCP/IP, and IoT communication's protocols. During the automatic code generation process, C code is generated for all possible variants present in a variant subsystem. It is only at the compilation phase that the selected variant is compiled and linked before being deployed onto the embedded target. The offline selection of variants is convenient to do in the models by using an *annotation callback* that automatically adapts all variants for either the "Stm32" implementation (low-level bare metal) or the "Mbed" implementation using the HAL. This can be automated by using variants conditions as well.

In the variants management figure (7.1), the button callback "Hardware drivers" allows to select the variant implementation of the hardware peripherals. The *variant manager* view is used to show all variants present in the Simulink model (6.29). It also contains the variants conditions that are evaluated to see which *variant control* must be enabled in the model. In this case, the annotation callback is set to the value "STM32" which sets the parameter "Mbed" to '0' or 'false'.

The parameter “Mbed” is used in the “Condition” column, and based on its value, the condition expression for each variant is updated accordingly; for each subsystem, a line with **black text** is ‘true’ (selected), whereas one with **gray text** is ‘false’ (not selected).

Hardware drivers
STM32
Implementation



Name	Submodel Configuration	Variant Control	Condition
StandaloneHw			
ADCInput			
AnalogSignalIn			
MBED		Abstraction	Mbed == 1
STM32		LowLevel	Mbed == 0
LedOutput			
BlueLED			
MBED		Abstraction	Mbed == 1
STM32		LowLevel	Mbed == 0
GreenLED			
MBED		Abstraction	Mbed == 1
STM32		LowLevel	Mbed == 0
RedLED			
MBED		Abstraction	Mbed == 1
STM32		LowLevel	Mbed == 0
Network			
IoT		IoTLowLevel	(ThingSpeak == 1)
Tcp		TcpLowLevel	(Server == 1) && (ThingSpeak == 0)
Udp		UdpLowLevel	(Server == 0) && (ThingSpeak == 0)
ResetButton			
MBED		Abstraction	Mbed == 1
STM32		LowLevel	Mbed == 0

Figure 7.1: Variants management

In the variants management view (7.1), there are two other parameters called “Server” and “ThingSpeak” that are only used in the standalone mode model (6.29). They are used to select the type of communication’s protocol to use in the “Network” variant subsystem. The logical expressions within the “Condition” column for the “Network” variant subsystem indicates that if the parameter “ThingSpeak” is ‘true’ then the “IoT” subsystem is always used. If “ThingSpeak” is ‘false’ then it depends on the “Server” parameter; the “Tcp” subsystem is used if “Server” is ‘true’, otherwise it is the “Udp” subsystem.

7.1 Techniques to integrate C code in Simulink

When a model is deployed onto an embedded system, there is a need to interact with the hardware peripherals present on it. This is where already existing C code drivers are re-used and integrated in the project. It exists several “user-defined functions” that can be used in Simulink to bring in external code written either in MATLAB or C. In this project, the drivers are written in C and C++ for the low-level Stm32 implementation as well as for the Mbed HAL. The Simulink blocks allowing to integrate C/C++ code in Simulink have been represented and sorted by their level of complexity in the figure (7.2).

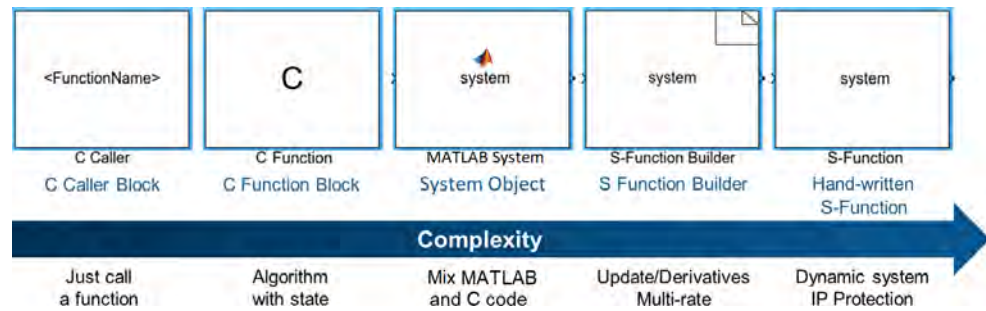


Figure 7.2: C code integration techniques

The C code integration techniques (7.2) represent the current state-of-the-art capabilities in MATLAB R2021a. This is important to mention, as such capabilities have been incrementally added to Simulink over time.

Here are the implementations details on the five C code integration techniques:

1) C Caller

On the left-hand side, the “C Caller” block has been first released in MATLAB R2018a. It allows to easily bring C functions declared in header files and defined in C files. The user just has to provide the path to the header files to include, the path to the C source files, as well as the name of the needed C files. Then, the mask of the block automatically provides a menu to select the C function to use within this C Caller block. Based on the selected function it automatically adapts the input and output ports. The limitation of this block is that it only works for what is called *procedural programming* paradigm. That means, it cannot handle states; it can only take inputs, apply mathematical operations with them and output results

2) C Function

The “C Function” block has been added to MATLAB R2021a. It allows to do procedural programming like the “C Caller” block, but not only. It can also encapsulate *object-oriented C++* code within C functions. Indeed, it also has the capability to define initialize and terminate functions. Moreover, it can handle states and internal parameters

3) System Object

The “System Object” technique is used to implement the hardware drivers in this project. It has been introduced in MATLAB R2013b. Indeed, all toolboxes having the word “system” in their name are using system objects, like the DSP System toolbox for example. System objects are coded following the *object-oriented programming* (OOP) paradigm and are by default using the “matlab.System” class which is the base class for System objects. It has the advantage that system objects can be both used in MATLAB and in Simulink. Basically, a system object is written in MATLAB OOP and contains a class with methods, attributes and states that can be private, public, protected, static and so on. It intrinsically has the three main methods for the initialization, the execution, and the termination of an object’s instance. These methods are respectively named `setupImpl()`, `stepImpl()`, and `resetImpl()` in the class. By using the *Coder* directives from the *MATLAB Coder* toolbox, it is possible to call and execute C/C++ code and files.

As system objects offer this capability of mixing MATLAB and C/C++ code, this makes them very convenient to use to implement hardware drivers. Moreover, system objects can run in multi-instance mode for re-entrant code and handle multiple rates as well. The implementation of such a system object is shown in details in §7.2

4) S-Function Builder

Creating an S-Function in Simulink is not straight forward for users who do not have experience in C programming. It requires to write a C wrapper around the C code to import it in Simulink. It is equivalent to bringing C code in MATLAB via a *MATLAB Executable* (MEX) file. The “S-Function Builder” (introduced before MATLAB R2006a) helps end users to create the requested wrapper by providing an interface that guides them through the required steps. It will then automatically generate required artifacts and already compile some of them as well. It provides the same kind of capabilities than the “C Function” block, but can also handle multi-rates and continuous derivatives (which is useful for dynamic systems). However, multi-instance mode is not directly supported; it can be manually added after the C and header files have been generated

5) S-Function

The “S-Function” block (introduced before MATLAB R2006a) is meant for experienced C programmers. The block only allows to enter the name of the S-Function, its parameters and the list of source files modules (C and headers). This technique is definitely the most complex as all the C code is written by hand. Some templates are available in the documentation to help on the structure of an S-Function

7.2 Digital inputs and outputs

In the standalone mode model (6.29), the “ResetButton” and the “LedOutput” variant subsystems are made of driver blocks that have access to the digital I/O's. The variant subsystems (7.3) shows the digital I/O's interfaces.

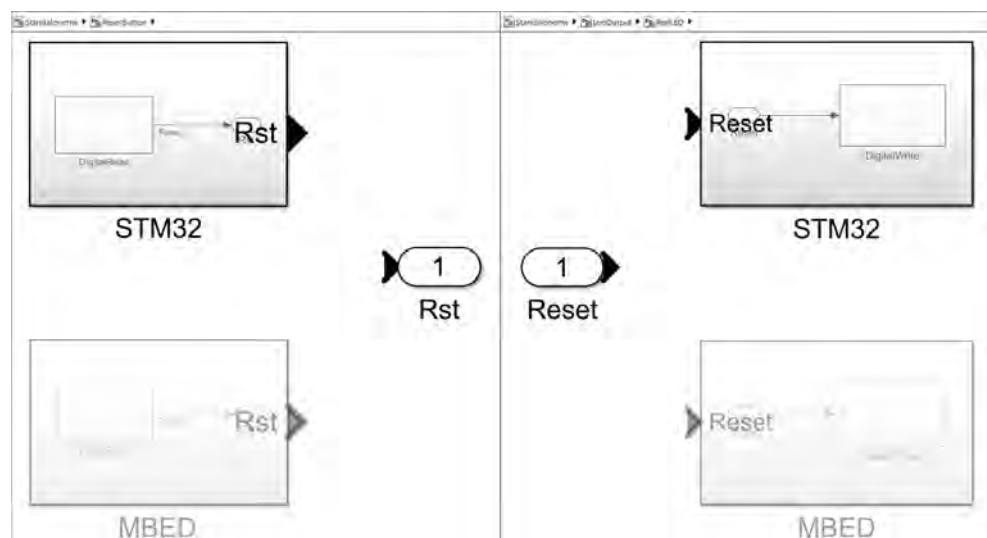


Figure 7.3: Variant subsystems for the digital read (left-hand side) and write (right-hand side)

To be complete on the description of each subsystem from the standalone mode model (6.29), a view on the contents of the “LedOutput” subsystem is given with its logic controlling each LED (7.4). Simple combinatorial logic operations are used to switch on and off the desired LEDs based on the current measurement’ status.

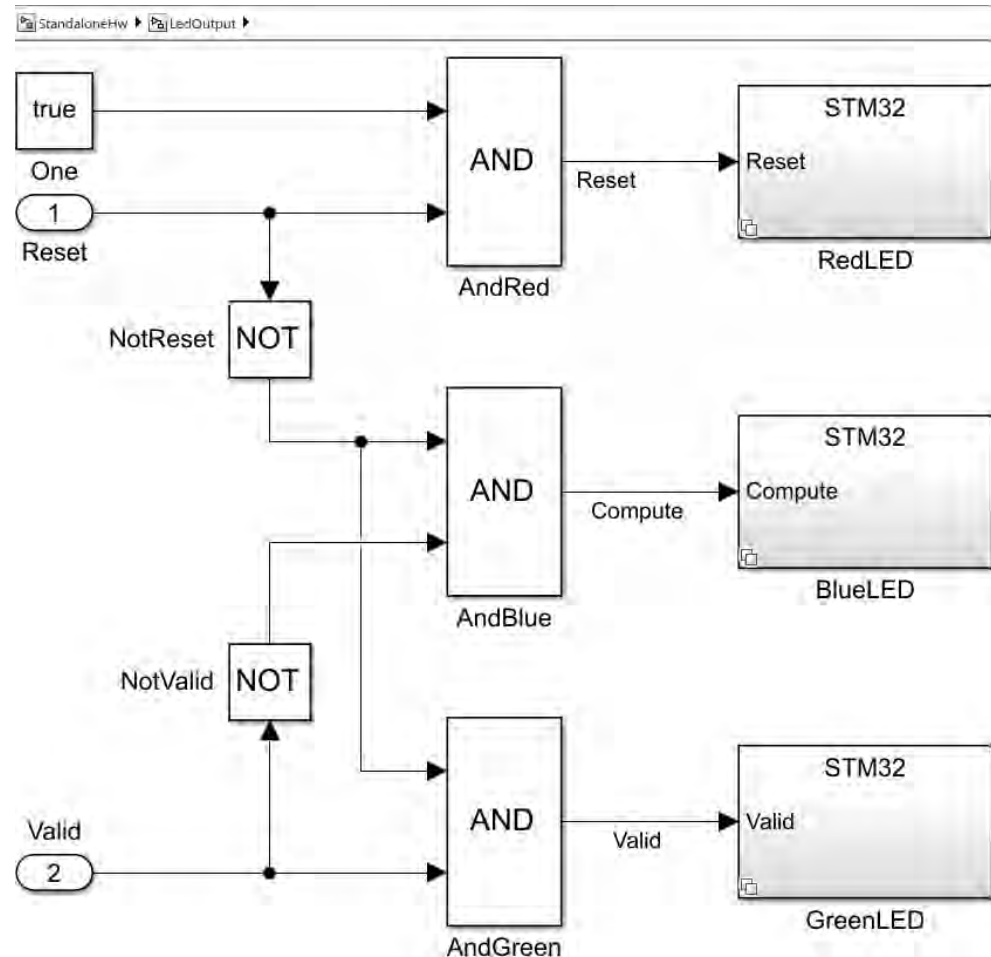


Figure 7.4: Logic controlling the red, green, and blue LEDs

Remark: for each system object described in the following sections, their intrinsic behavior’s description automatically produces a *subsystem mask* with the inputs that the end user has to fulfill. A comparison on the produced masks, but not on the source code, is always done between the Stm32 and Mbed implementations. Otherwise, too much source code would be shown in the thesis without being especially useful. Therefore, the source code is only shown once for the system objects accessing the digital I/O’s, but not for the others.

7.2.1 Digital read

The digital read driver block allows to get the value ‘0’ or ‘1’ from a digital pin on the board. In §3.3.1, the standard pin names used by Mbed are explained in details. Basically, for a digital read operation, the allowed pin names are “BUTTON0” and “BUTTON1”, as well as “D0” to “D15”. On the evaluation board (4.2), the only available built-in button is “BUTTON1”; that is why “BUTTON0” is not proposed in the pin name menu of the Mbed digital read system object (7.5a).

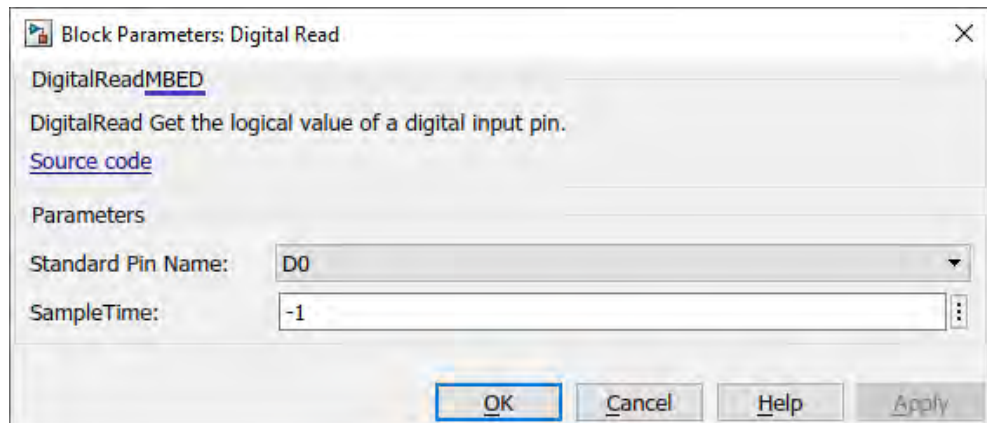


Figure 7.5a: Digital read system object masks for the Mbed implementation

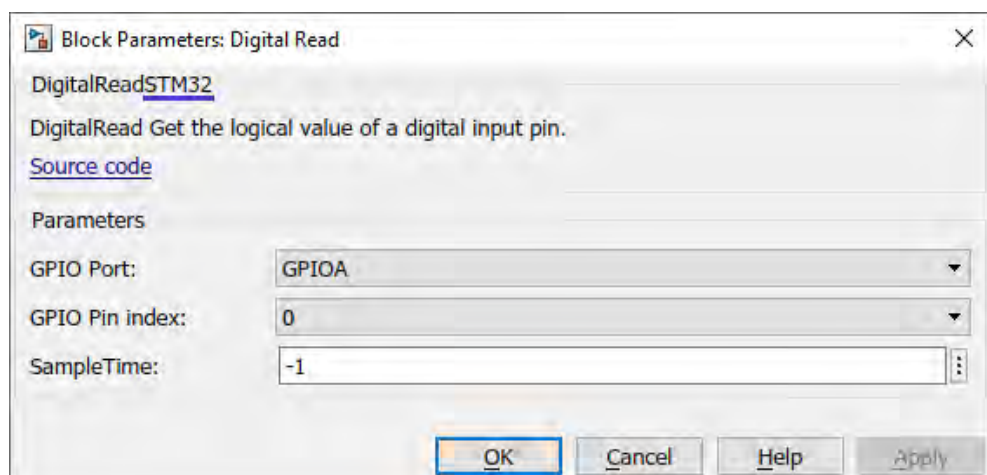


Figure 7.5b: Digital read system object masks for the Stm32 implementation

The digital read masks (7.5a) and (7.5b) are not that different from the end user point of view. In the Mbed case (7.5a), there is one selection menu out of which only the allowed pin names are available. For the Stm32 version (7.5b), two selection menus are provided; the first one to select the digital I/O port, and the second one to select the pin index within the selected port. In order to select the right pin on the board, the user must know where it is mapped onto the corresponding pin on the processor. Therefore, a tight look at the pin mappings (7.6a) and (7.6b) for the evaluation board (4.2) is requested for the Stm32 implementation. This requires some understanding of the used hardware, as pins can be used in various modes of operation.

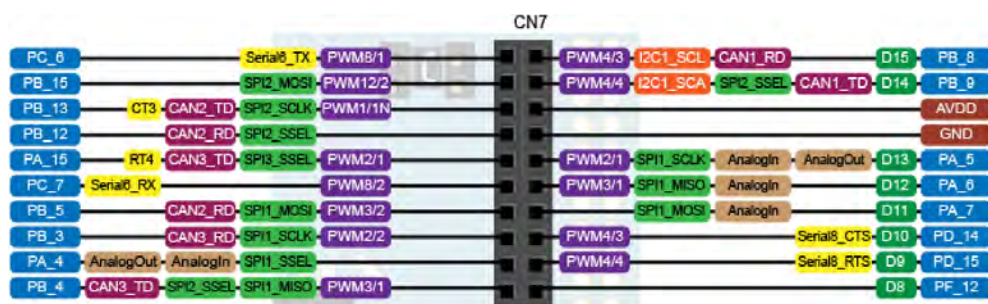


Figure 7.6a: Pin mapping of the digital inputs and outputs of the connector 7

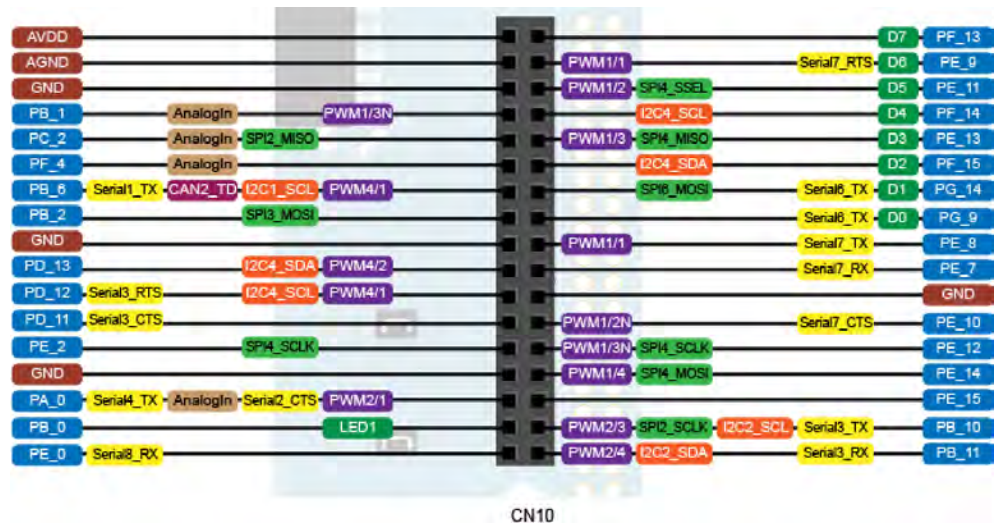


Figure 7.6b: Pin mapping of the digital inputs and outputs of the connector 10

The Mbed mask (7.5a) is automatically generated based on the *non-tunable* properties defined in its corresponding system object code (7.7a).

```

classdef DigitalReadMBED < realtime.internal.SourceSampleTime &...
    coder.ExternalDependency

    %DigitalRead Get the logical value of a digital input pin.
    %
    % System object of a digital read block using the MBED HAL.
    %
    %#codegen

    properties
        % Public, tunable properties.
    end

    properties (Nontunable)
        % Standard Pin Name
        % Select the number of the GPIO pin to read from as one of
        % ['BUTTON1'|{'D0'}| 'D1' | 'D2' | 'D3' | 'D4' | 'D5' | 'D6' | 'D7' |
        %     'D8' | 'D9' | 'D10'| 'D11'| 'D12'| 'D13'| 'D14'| 'D15' ].
        GpioPin = 'D0';
    end

    properties (Constant, Hidden)
        GpioPinSet = matlab.system.StringSet(...
            {'BUTTON1','D0','D1','D2','D3','D4','D5','D6','D7',...
            'D8','D9','D10','D11','D12','D13','D14','D15'});
    end

    properties (Access = protected)
        Direction = 0; % Input mode => Read pin
        MW_DIGITALIO_HANDLE;
    end

    methods
        % Constructor
        function obj = DigitalReadMBED(varargin)
            % Support name-value pair arguments when constructing the object
            setProperties(obj,nargin,varargin{:});
        end
    end
end

```

Figure 7.7a: Mbed implementation of the digital read system object (properties)


```

methods (Access=protected)
function setupImpl(obj)
    if isempty(coder.target)
        % Place simulation setup code here
    else
        %% Include all needed low-level API header files for the GPIOs
        coder.cinclude('MW_digitalIO.h');

        %% Declaration of needed local variables with their specific type
        obj.MW_DIGITALIO_HANDLE = coder.opaque('MW_Handle_Type',...
                                                'HeaderFile', 'MW_SVD.h');
        pinIdxLoc = coder.opaque('uint32_t', obj.GpioPin);

        %% Call C-function implementing device initialization

        % Configure GPIO in input mode
        obj.MW_DIGITALIO_HANDLE = coder.ceval('MW_digitalIO_open',...
                                                pinIdxLoc, obj.Direction);
    end
end
function y = stepImpl(obj)
    y = false;
    if isempty(coder.target)
        % Place simulation output code here
    else
        %% Call C-function implementing device output

        % Extract level of the selected GPIO pin
        y = coder.ceval('MW_digitalIO_read', obj.MW_DIGITALIO_HANDLE);
    end
end
function releaseImpl(obj)
    if isempty(coder.target)
        % Place simulation termination code here
    else
        %% Call C-function implementing device termination
        coder.ceval('MW_digitalIO_close', obj.MW_DIGITALIO_HANDLE);
    end
end
end

methods (Static)
function name = getDescriptiveName()
    name = 'Source';
end

function b = isSupportedContext(context)
    b = context.isCodeGenTarget('rtw');
end

function updateBuildInfo(buildInfo, context)
    if context.isCodeGenTarget('rtw')
        % MBED Digital I/O interface
        mbedDir = codertarget.mbed.internal.getRootDir;
        addIncludePaths(buildInfo, fullfile(mbedDir, 'include'));
        addIncludeFiles(buildInfo, 'MW_digitalIO.h');
        addSourceFiles(buildInfo, 'MW_digitalIO.cpp', fullfile(mbedDir, 'src'));
        addIncludeFiles(buildInfo, 'MW_MbedPinInterface.h');
    end
end
end
end
end

```

Figure 7.7b: Mbed implementation of the digital read system object (methods)

The system object code section (7.7a) is made of two main properties; the sampling time that is inherited from the class “realtime.internal.SourceSampleTime” on top of the class definition, and the “GpioPin” that follows the Arduino Uno Pin Name convention. The valid values of “GpioPin” are listed in “GpioPinSet”. When the user fulfills the requested properties in the mask, only the pre-defined values are allowed. In this code section, a standard constructor method is there for the instantiation of the object.

In the system object code section (7.7b), the focus is put on the methods realizing the effective driver’s tasks. As mentioned in §7.1, the behavior of the three main methods `setupImpl()`, `stepImpl()`, and `releaseImpl()` must be defined. These methods are respectively implementing the “initialization”, “execution”, and “termination” actions of the object’s instance.

The implementation of the functionality for each method is done in MATLAB code and the access to the Mbed HAL in C code is done via calls to the MATLAB Coder directives. The first directive that is used in the `setupImpl()` method is called “`coder.target`”. It allows to select the code to execute based on its effective execution. In this project, the drivers are used for deployment only, so for C code generation, but not for simulation. However, in the first if statement with “`coder.target`”, it is possible to implement the behavior of the system object when it is used for simulation which, in some cases, can be very useful.

The next directive to be used is called “`coder.cinclude`” and it allows to include one or more external header files to the project. Here, the Mbed HAL is implemented via the “`MW_digitalIO.h`” header file that contains the prototype functions to initialize, execute, and terminate the access in read or write mode to a digital I/O pin.

Then, the “`coder.opaque`” directive is used to declare and initialize a local variable, like for example: `myVar = coder.opaque('uint32_t', 0);` that produces `uint32_t myVar = 0;`. In the `setupImpl()` method, the handle to the object’s instance (a void pointer) is initialized before getting its pointer value from the call to the C function ‘`MW_digitalIO_open()`’. The “`coder.ceval`” directive is used when it comes to call and execute a C function, like ‘`MW_digitalIO_open()`’. The first input argument of “`coder.ceval`” is the name of the C function, and the remaining ones correspond to the input arguments of the called C function. Its output argument, in this case the pointer to the handle, is retrieved on the left-hand side of the equal sign.

The code of the `stepImpl()` method is simply made of the initialization of the output Boolean variable ‘`y`’ and the call to the C function ‘`MW_digitalIO_read()`’ that under the hood calls the ‘`read()`’ method introduced in the Mbed API for digital I/O’s (3.5).

The code of the `releaseImpl()` method is simply calling the C function ‘`MW_digitalIO_close()`’ that releases the used resources; especially the handle to the digital I/O pin object’s instance.

The `updateBuildInfo()` method is there to include the required header and source files, and their folders locations, like the “`mbedDir`”. This is here that the Mbed related C files are linked to the generic MathWorks files, allowing to connect them to any C interface, like the Mbed HAL.

The properties defined in the system object code for the Stm32 implementation (7.8a) are a bit different. Indeed, the digital pin parameter is divided into its logical port and its pin number. Moreover, there are seven ports providing 16 pin indices each, but the eighth's one (GPIOH) has only two pins. This means that the mask of the system object must dynamically adapt its pin index selection menu based on the selected GPIO port, which makes it already a bit more complex than the Mbed implementation. This also implies that the end user knows that for example the Arduino Uno Pin Name 'D6' is indeed 'PE_9', with the logical port 'GPIOE' and the pin index '9'. This requires some technical knowledge on the used hardware target.

```

classdef DigitalReadSTM32 < realtime.internal.SourceSampleTime &...
    coder.ExternalDependency

    %DigitalRead Get the logical value of a digital input pin.
    %
    % System object of a digital read block.
    %
    %#codegen

    properties
        % Public, tunable properties.
    end

    properties (Nontunable)
        % GPIO Port
        % Select a GPIO port to read from as one of
        % [{'GPIOA'} | 'GPIOB' | 'GPIOC' | 'GPIOD' |
        % 'GPIOE' | 'GPIOF' | 'GPIOG' | 'GPIOH' ].
        GpioPort = 'GPIOA';

        % GPIO Pin index
        % Select the number of the GPIO pin to read from as one of
        % [{'0'} | '1' | '2' | '3' | '4' | '5' | '6' | '7' |
        % '8' | '9' | '10' | '11' | '12' | '13' | '14' | '15' ].
        GpioPin = '0';

        % GPIO Pin index
        % Select the number of the GPIO pin to read from as one of
        % [{'0'} | '1' ].
        GpioPinH = '0';
    end

    properties (Constant, Hidden)
        GpioPortSet = matlab.system.StringSet(...
            {'GPIOA', 'GPIOB', 'GPIOC', 'GPIOD', 'GPIOE', 'GPIOF', 'GPIOG', 'GPIOH'});
        GpioPinSet = matlab.system.StringSet(...
            {'0', '1', '2', '3', '4', '5', '6', '7', '8', '9', '10', '11', '12', '13', '14', '15'});
        GpioPinHSet = matlab.system.StringSet({'0', '1'});
    end

    methods
        % Constructor
        function obj = DigitalReadSTM32(varargin)
            % Support name-value pair arguments when constructing the object
            setProperties(obj, nargin, varargin{:});
        end
    end
end

```

Figure 7.8a: Stm32 implementation of the digital read system object (properties)

In the Stm32 implementation (7.8b), the method `isInactivePropertyImpl()` handles the dynamic adaptation of the pin index selection menu based on the selected port.

The code of the `setupImpl()` method for the Stm32 implementation (7.8b) is more complex than for the Mbed one (7.7b). First, the low-level modules to include are target specific, like `'stm32f7xx_ll_gpio.h'` and `'stm32f7xx_ll_bus.h'`; this means that they must be adapted if the target changes. Then, the right macros and local variable used in the C source files must be built based on the mask's parameters. Once, this is done, the initialization of the GPIO can be done. However, not only the pin must be set to read or write, but also the GPIOs clock must be enabled, which was not requested with the Mbed HAL.

```

methods (Access=protected)
function flag = isInactivePropertyImpl(obj,prop)
    flag = false;
    % Filter the Pin selection related properties
    switch (prop)
        case 'GpioPinH'
            if ~strcmp(obj.GpioPort, 'GPIOH')
                flag = strcmpi(prop, 'GpioPinH');
            end
        case 'GpioPin'
            if strcmp(obj.GpioPort, 'GPIOH')
                flag = strcmpi(prop, 'GpioPin');
            end
        otherwise
            % That is an error
        end
    end
end
function setupImpl(obj)
    if isempty(coder.target)
        % Place simulation setup code here
    else
        %% Include all needed low-level API header files for the GPIOs
        coder.cinclude('stm32f7xx_ll_gpio.h');
        coder.cinclude('stm32f7xx_ll_bus.h');

        %% Recomposition of macro names

        % Set the pin mode to input
        pinModeMacro = 'LL_GPIO_MODE_INPUT';
        % Extract the GPIO port and index
        pinPortMacro = obj.GpioPort;
        if ~strcmpi(obj.GpioPort, 'GPIOH')
            pinIndexMacro = ['LL_GPIO_PIN_' obj.GpioPin];
        else
            pinIndexMacro = ['LL_GPIO_PIN_' obj.GpioPinH];
        end

        %% Declaration of needed local variables with their specific type
        gpioClockLoc = coder.opaque('uint32_t', ['LL_AHB1_GRP1_PERIPH_' pinPortMacro]);
        pinIndexLoc = coder.opaque('uint32_t', pinIndexMacro);
        pinModeLoc = coder.opaque('uint32_t', pinModeMacro);
        pinPortLoc = coder.opaque('GPIO_TypeDef *', pinPortMacro);

        %% Call C-function implementing device initialization

        % Enable the GPIOs clock
        coder.ceval('LL_AHB1_GRP1_EnableClock', gpioClockLoc);
        % Configure GPIO in input mode
        coder.ceval('LL_GPIO_SetPinMode', pinPortLoc, pinIndexLoc, pinModeLoc);
    end
end

```

Figure 7.8b: Stm32 implementation of the digital read system object (methods part 1)

The same explanation do apply to the `stepImpl()` and `releaseImpl()` methods. It is also important to note that the reading of the GPIO pin level ('0' or '1') is done in two steps; the GPIO port is read and then a pin index mask is applied to get the pin level.

```
function y = stepImpl(obj)
    y = false;
    if isempty(coder.target)
        % Place simulation output code here
    else
        %% Recomposition of macro names

        % Extract the GPIO port and index
        pinPortMacro = obj.GpioPort;
        if ~strcmpi(obj.GpioPort, 'GPIOH')
            pinIndexMacro = ['LL_GPIO_PIN_' obj.GpioPin];
        else
            pinIndexMacro = ['LL_GPIO_PIN_' obj.GpioPinH];
        end

        %% Declaration of needed local variables with their specific type
        pinPortLoc = coder.opaque('GPIO_TypeDef *', pinPortMacro);
        pinIndexLoc = coder.opaque('uint32_t', pinIndexMacro);

        %% Call C-function implementing device output

        % Get current GPIO port levels
        y = coder.ceval('LL_GPIO_ReadInputPort', pinPortLoc);
        % Extract level of the selected GPIO pin
        y = coder.ceval('LL_GPIO_IsInputPinSet', pinPortLoc, pinIndexLoc);
    end
end

function releaseImpl(obj)
    if isempty(coder.target)
        % Place simulation termination code here
    else
        %% Recomposition of macro names

        % Extract the GPIO port and index
        pinPortMacro = obj.GpioPort;

        %% Declaration of needed local variables with their specific type
        pinPortLoc = coder.opaque('GPIO_TypeDef *', pinPortMacro);
        %% Call C-function implementing device termination
        coder.ceval('LL_GPIO_DeInit', pinPortLoc);
    end
end

methods (Static)
    function name = getDescriptiveName()
        name = 'Source';
    end
    function b = isSupportedContext(context)
        b = context.isCodeGenTarget('rtw');
    end
    function updateBuildInfo(~, context)
        if context.isCodeGenTarget('rtw')
            % Rely on the files of the Hardware Support Package
        end
    end
end
end
```

Figure 7.8c: Stm32 implementation of the digital read system object (methods part 2)

The `updateBuildInfo()` method of the Stm32 implementation (7.8c) is empty, as all the requested include and source files are provided by the STMicroelectronics Nucleo Boards hardware support package (HSP) that is installed in MATLAB.

7.2.2 Digital write

The digital write driver block allows to set the value '0' or '1' to a digital pin on the board. The same "D0" to "D15" pins as described in §7.2.1 can be used for Mbed. In addition to these pins, the other allowed pin names are "LED1", "LED2", "LED3" and "LED4". These pin names are proposed in the menu of the Mbed digital write system object (7.9a).

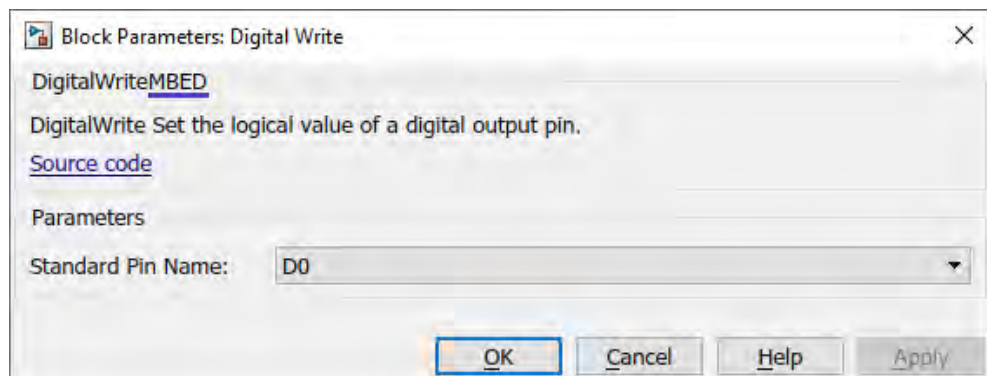


Figure 7.9a: Digital write system object masks for the Mbed implementation

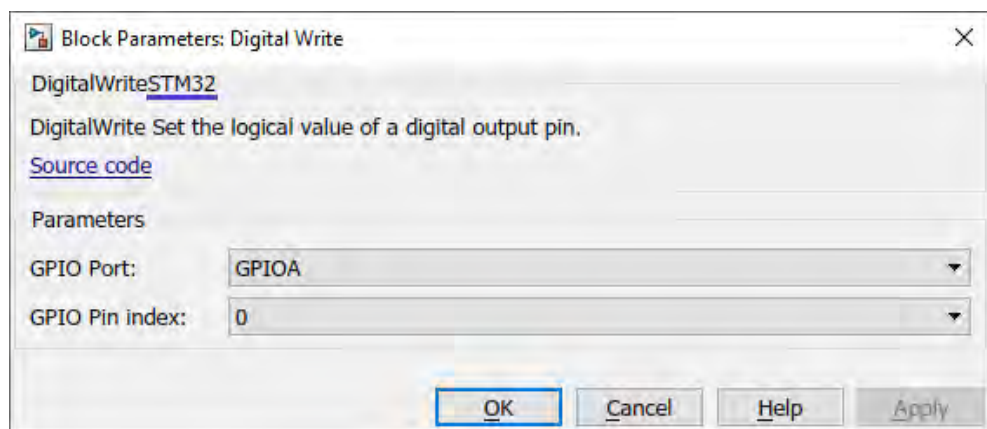


Figure 7.9b: Digital write system object masks for the Stm32 implementation

As for the digital read masks (7.5a) and (7.5b), the digital write masks (7.9a) and (7.9b) have respectively the same options than the digital read masks, and are generated in the same way.

The properties of the Mbed digital write system object (7.10a) are the same than for the digital read one (7.7a). Only the "LEDx" values replace the "BUTTONx" ones, and the pin "Direction" is set to the output mode to write to the selected pin.

```

classdef DigitalWriteMBED < matlab.System &...
    coder.ExternalDependency
    %DigitalWrite Set the logical value of a digital output pin.
    %
    % System object of a digital write block using the MBED HAL.
    %
    %#codegen

    properties
        % Public, tunable properties.
    end

    properties (Nontunable)
        % Standard Pin Name
        % Select the number of the GPIO pin to write to as one of
        % [{"D0"} | 'D1' | 'D2' | 'D3' | 'D4' | 'D5' | 'D6' | 'D7' |
        %     'D8' | 'D9' | 'D10' | 'D11' | 'D12' | 'D13' | 'D14' | 'D15' |
        %     'LED1' | 'LED2' | 'LED3' | 'LED4'].
        GpioPin = 'D0';
    end

    properties (Constant, Hidden)
        GpioPinSet = matlab.system.StringSet(...
            {'D0', 'D1', 'D2', 'D3', 'D4', 'D5', 'D6', 'D7', 'D8', 'D9', 'D10', ...
            'D11', 'D12', 'D13', 'D14', 'D15', 'LED1', 'LED2', 'LED3', 'LED4'});
    end

    properties (Access = protected)
        Direction = 1; % Output mode => Write pin
        MW_DIGITALIO_HANDLE;
    end

    methods
        % Constructor
        function obj = DigitalWriteMBED(varargin)
            % Support name-value pair arguments when constructing the object
            setProperties(obj, nargin, varargin{:});
        end
    end
end

```

Figure 7.10a: Mbed implementation of the digital write system object (properties)

The methods for the digital write system object (7.10b) are the same than for the digital read one (7.7b). The only difference is that the `setUpImpl()` method configures the GPIO in output mode this time, and the code of the `stepImpl()` method simply calls the C function `'MW_digitalIO_write()'` with the input argument `'u'` that defines the pin level. Under the hood, this function calls the `'write()'` method introduced in the Mbed API for digital I/O's (3.5).


```

methods (Access=protected)
function setupImpl(obj)
    if isempty(coder.target)
        % Place simulation setup code here
    else
        %% Include all needed low-level API header files for the GPIOs
        coder.cinclude('MW_digitalIO.h');

        %% Declaration of needed local variables with their specific type
        obj.MW_DIGITALIO_HANDLE = coder.opaque('MW_Handle_Type',...
                                                'HeaderFile', 'MW_SVD.h');
        pinIdxLoc = coder.opaque('uint32_t', obj.GpioPin);

        %% Call C-function implementing device initialization

        % Configure GPIO in output mode
        obj.MW_DIGITALIO_HANDLE = coder.ceval('MW_digitalIO_open',...
                                                pinIdxLoc, obj.Direction);
    end
end

function stepImpl(obj, u)
    if isempty(coder.target)
        % Place simulation output code here
    else
        %% Call C-function implementing device output

        % Set the level of the selected GPIO pin
        coder.ceval('MW_digitalIO_write', obj.MW_DIGITALIO_HANDLE, u);
    end
end

function releaseImpl(obj)
    if isempty(coder.target)
        % Place simulation termination code here
    else
        %% Call C-function implementing device termination
        coder.ceval('MW_digitalIO_close', obj.MW_DIGITALIO_HANDLE);
    end
end

methods (Static)
function name = getDescriptiveName()
    name = 'Sink';
end

function b = isSupportedContext(context)
    b = context.isCodeGenTarget('rtw');
end

function updateBuildInfo(buildInfo, context)
    if context.isCodeGenTarget('rtw')
        % MBED Digital I/O interface
        mbedDir = codertarget.mbed.internal.getRootDir;
        addIncludePaths(buildInfo, fullfile(mbedDir, 'include'));
        addIncludeFiles(buildInfo, 'MW_digitalIO.h');
        addSourceFiles(buildInfo, 'MW_digitalIO.cpp', fullfile(mbedDir, 'src'));
        addIncludeFiles(buildInfo, 'MW_MbedPinInterface.h');
    end
end
end
end

```

Figure 7.10b: Mbed implementation of the digital write system object (methods)

The properties defined in the digital write system object code for the Stm32 implementation are exactly the same than for the digital read one (7.8a). That is why the code for it is not repeated here. The same remark does apply for the `isInactivePropertyImpl()` and `setupImpl()` Stm32 methods (7.8b) as well. The only difference is that the pin mode is set to output ('LL_GPIO_MODE_OUTPUT') instead of input ('LL_GPIO_MODE_INPUT').

The difference within the `stepImpl()` method is that when the input level 'u' must be written to the pin, either the low-level C function 'LL_GPIO_SetOutputPin' is called to write a '1', or the low-level C function 'LL_GPIO_ResetOutputPin' is called to write a '0'.

The `releaseImpl()` and `updateBuildInfo()` methods are the same than in the Stm32 implementation of the digital read system object (7.8c).

```
function stepImpl(obj,u)
    if isempty(coder.target)
        % Place simulation output code here
    else
        %% Recomposition of macro names

        % Extract the GPIO port and index
        pinPortMacro = obj.GpioPort;
        if ~strcmpi(obj.GpioPort,'GPIOH')
            pinIndexMacro = ['LL_GPIO_PIN_' obj.GpioPin];
        else
            pinIndexMacro = ['LL_GPIO_PIN_' obj.GpioPinH];
        end

        %% Declaration of needed local variables with their specific type
        pinPortLoc = coder.opaque('GPIO_TypeDef+',pinPortMacro);
        pinIndexLoc = coder.opaque('uint32_t',pinIndexMacro);

        %% Call C-function implementing device behaviour
        if isequal(logical(u),true)
            % Set GPIO pin to high level
            coder.ceval('LL_GPIO_SetOutputPin',pinPortLoc,pinIndexLoc);
        else
            % Set GPIO pin to low level
            coder.ceval('LL_GPIO_ResetOutputPin',pinPortLoc,pinIndexLoc);
        end
    end
end
```

Figure 7.11: Stm32 implementation of the digital write system object (`stepImpl()` method)

7.3 Analog-to-digital converter

The analog-to-digital converter (ADC) peripheral converts the analog input signal coming from the signal acquisition board (4.3) into a digital signal for the processor. The driver for such a peripheral is much more complex than for the digital I/O's described in §7.2. Multiple modes of operation are available for an ADC and these are listed in the §7.3.1 for the Stm32 implementation.

7.3.1 ADC modes of operation

On the STMicroelectronics Nucleo evaluation board (4.2), three 12-bit ADCs are available and each of them shares up to 16 external channels, performing conversions in single-shot or scan mode. In scan mode, automatic conversion is performed on a selected group of analog inputs. The ADC can be served by the DMA controller. An analog watchdog feature allows very precise monitoring of the converted voltage of one, some or all selected channels. An interrupt is generated when the converted voltage is outside the programmed thresholds. To synchronize A/D conversion and timers, the ADCs could be triggered by any of its eight timers (TIMx). The ADC operation's modes table (7.1) shows all possible ADC configurations when they work in the "Independent" mode. Indeed, it is even possible to configure the ADCs to work together, but this is not explained here.

ADC Settings	ADC Regular Conversion Mode	ADC Injected Conversion Mode	DMA Settings	WatchDog
Clock Prescaler [1/2, {1/4}, 1/6, 1/8]	Number Of Conversions [{1}; 16]	Number Of Conversions [{1}; 4]	Mode [{Normal}, Circular]	Enable Analog WatchDog Mode * [{Disabled}, Enabled]
Bit resolution [6, 8, 10, {12}]	External Trigger Conversion Source [Software], Timer 1 CC 1, Timer 1 CC 2, Timer 1 CC 3, Timer 2 CC 2, Timer 3 CC4, Timer 4 CC 4, Timer 1 TRGO, Timer 1 TRGO 2, Timer 2 TRGO, Timer 4 TRGO, Timer 5 TRGO, Timer 6 TRGO, Timer 8 TRGO, Timer 8 TRGO 2 *	External Trigger Conversion Source [Software], Timer 1 CC 4, Timer 2 CC 1, Timer 3 CC 1, Timer 3 CC3, Timer 3 CC4, Timer 4 CC 4, Timer 8 CC 4, Timer 1 TRGO, Timer 1 TRGO 2, Timer 2 TRGO, Timer 4 TRGO, Timer 5 TRGO, Timer 6 TRGO, Timer 8 TRGO, Timer 8 TRGO 2*	Stream - ADC1 [DMA2 Stream 0], DMA2 Stream 4] - ADC2 [DMA2 Stream 2], DMA2 Stream 3] - ADC3 [DMA2 Stream 0], DMA2 Stream 1]	*Watchdog Mode [Single Regular], Single Injected, Single Both, All Regular, All Injected, All Both]
Data Alignment [Left, {Right}]	External Trigger Conversion Edge [None] *[{Raising}, Falling, Both]	External Trigger Conversion Edge [None] *[{Raising}, Falling, Both]	Direction [{Memory}]	*Analog WatchDog - ADC1, ADC2 [Channel 0] - ADC1, ADC2 [Channel 3]
Scan Conversion Mode * [{Disabled}, Enabled]	Rank – Channel => Auto handling	*Injected Conversion Mode [None], Discontinuous, Auto]	Priority [Low], Medium, High, Very High]	*High Threshold [0]; 4095]
Continuous Conversion Mode [{Disabled}, Enabled]	Rank – Sampling Time => Auto handling	Rank – Channel => Auto handling	Use Fifo * [{Disabled}, Enabled]	*Low Threshold [0]; 4095]
*Discontinuous Conversion Mode [{Disabled}, Enabled]		Rank – Sampling Time => Auto handling	* Threshold [1/4, 1/2, 3/4, {1}]	*Interrupt Mode [{Disabled}, Enabled]
*Number Of Discontinuous Conversion [1 ; 8]		Rank – Injected Offset => Auto handling	* Data Width => Auto handling	
DMA Continuous Requests [{Disabled}, Enabled]			* Burst Site => Auto handling	
End Of Conversion Selection [EOC flag at the end of single channel conversion, EOC flag at the end of all conversions]				

Table 7.1: ADC independent operation's modes

The listing of all possible independent operation's modes for an ADC is needed to be able to write its corresponding Stm32 system object correctly.

Based on this, the ADC mask (7.12) is generated from its system object's code.

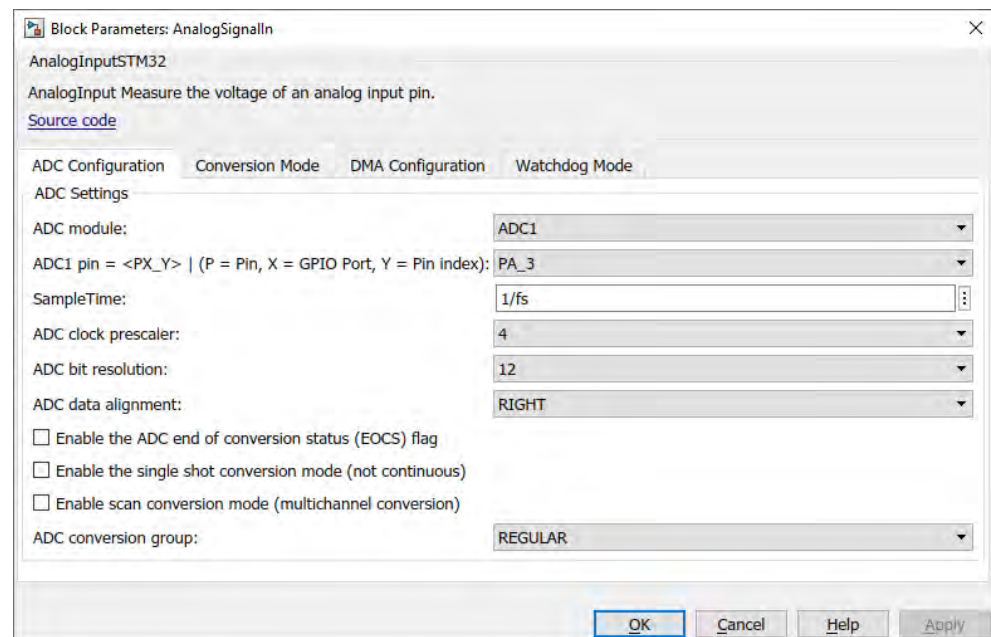


Figure 7.12: ADC system object mask for the Stm32 implementation

The Stm32 ADC mask (7.12) adapts its contents dynamically as some settings depend on others. At that level of details it is required that end users know how the ADC are connected to the processor and how they want to use it. However, for users who are not that deep inside technical details, the goal is definitely to abstract all this low-level complexity to them.

Regarding the Mbed implementation of the ADC (7.13), only three input parameters are provided to the end users. The first one is the Arduino Uno Pin Name for an analog input that goes from A0 to A5 as explained in §3.3.1. The second one is an additional output flag that allows to see if ADC conversions are done successfully or not. The last one, called 'SampleTime' represents the ADC sampling time that is inherited from the class "realtime.internal.SourceSampleTime". For the Mbed implementation, the ADC works in the "Regular Conversion Mode" with the "Software Trigger Conversion Source" only. This mode corresponds to the two first cells of the orange column of the ADC operation's modes table (7.1). All other required settings are already set to their default/most common values.

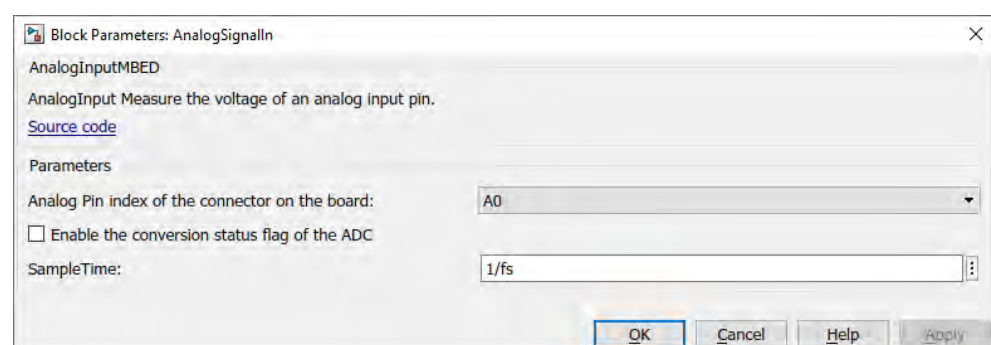


Figure 7.13: ADC system object mask for the Mbed implementation

7.4 Communication's protocols

Three communication's protocols are used by the STM Nucleo embedded system (4.2) to send data over networks. UDP and TCP/IP are used to send data over a local private network. On the target side, the UDP and TCP/IP send blocks provided by its HSP are directly used. On the host side, the UDP and TCP/IP receive blocks from the Instrument Control toolbox are directly used as well. The third communication's protocol opens an IoT channel to send data up to the MathWorks cloud solution called *ThingSpeak* directly over the Internet. On the target side, a unique system object has been created to send data in a ThingSpeak channel.

For this project, there are five main signals to send over these networks: the raw ECG signal, the mean BPM value, the BPM standard deviation, the valid flag and the reset flag.

7.4.1 UDP and TCP/IP

The UDP protocol consists of sending data packets over a network to a specific remote IP address like '192.168.1.100' as shown in the UDP send mask (7.14), or to broadcast them to all IP addresses by using '255.255.255.255' without waiting on any received confirmation from the remote receiver(s). This means that if a data packet is not transmitted successfully, it will not be resent and its data are then lost.

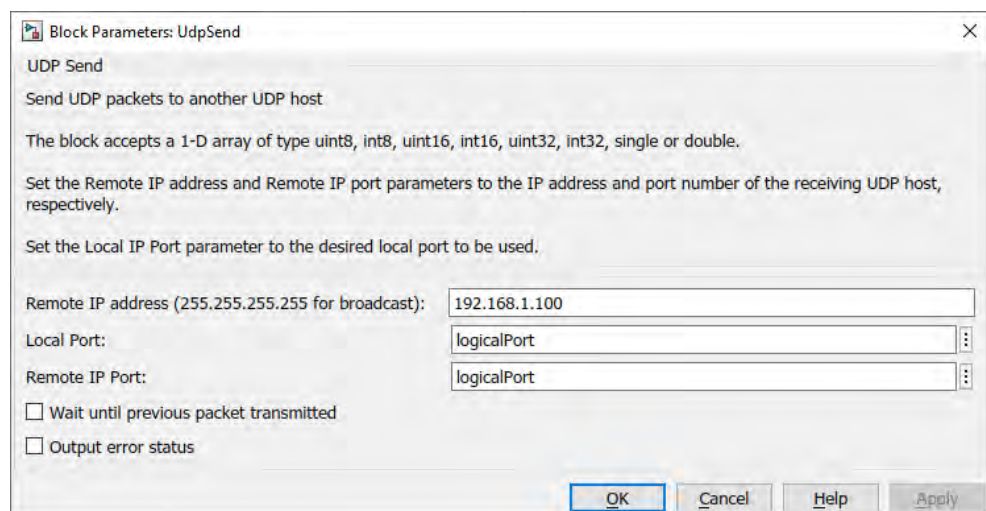


Figure 7.14: UDP send system object's mask

The TCP/IP protocol consists of a client/server architecture. When the embedded system is set in 'Server' mode as shown in the TCP/IP send mask (7.15), it will send data packets over the network to a remote host and wait on a confirmation that each data packet is received correctly. Indeed, the local IP port acts as a listening port on the TCP/IP server. If a data packet is not transmitted successfully, it will be resent to avoid a loss of data.

It is also possible to set the embedded system in 'Client' mode. In this case, it is required to provide the server IP address and its corresponding IP port so that data are sent to the right server.

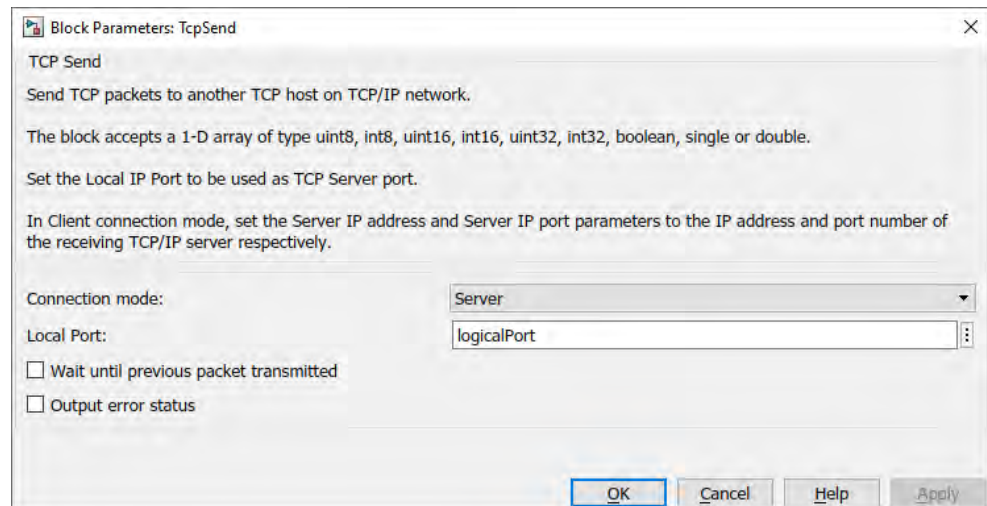


Figure 7.15: TCP/IP send system object's mask

In both cases, UDP (7.14) and TCP/IP (7.15), the local and remote logical ports are set by the parameter 'logicalPort' that has the value '25000' to make it simple. In fact, any available logical port value can be set to the local and remote logical port fields and they can also be different.

The Ethernet settings (7.16) of the standalone model (6.29) deployed onto the embedded system are set for a local network with the private IP address '192.168.1.10'. The remote host IP address is directly set in the gateway field '192.168.1.100'. This way the end user can be sure that both devices are in the same private local network.

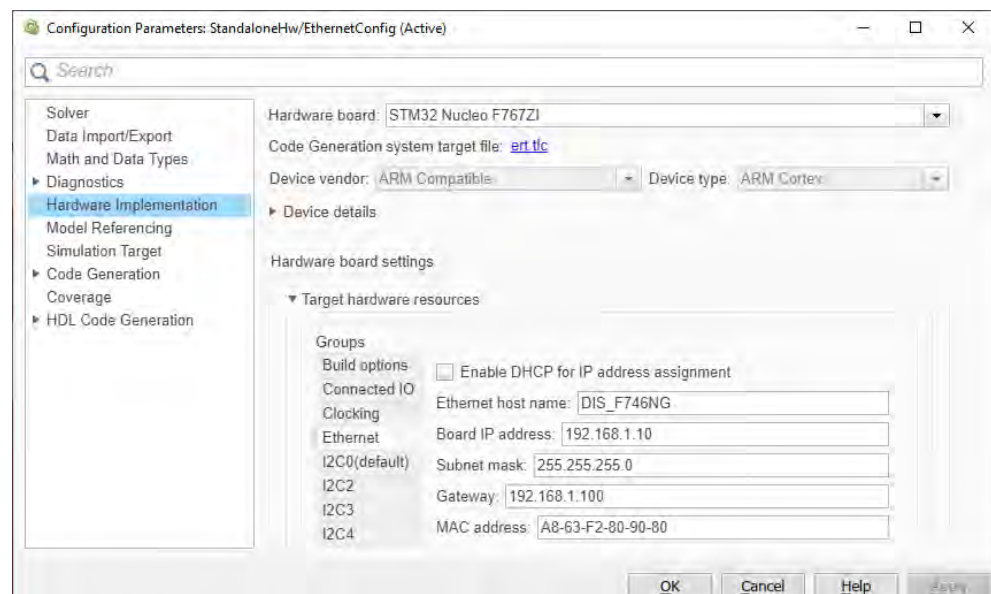


Figure 7.16: Ethernet configuration of the embedded target within a private local network

Remark: it can be that the firewall present on the remote host blocks the incoming data packets. Therefore, a rule should be setup in the firewall to allow such UDP or TCP/IP packets to go through it.

7.4.2 IoT channel

In this case, the goal is to setup a communication's channel to send data over the Internet up to the ThingSpeak cloud solution created by MathWorks. This is an IoT analytics platform service that allows to collect, process and visualize data remotely in the cloud. The diagram (7.17) shows on the left-hand side what is called the "edge node". In this project, it consists of the embedded system (4.2) processing and sending the ECG data. In the middle, the data sent by the edge node are collected in the "operational technology node", also known as the cloud via the ThingSpeak channel. On the right-hand side, the MATLAB environment present on the cloud allows to process and visualize data.

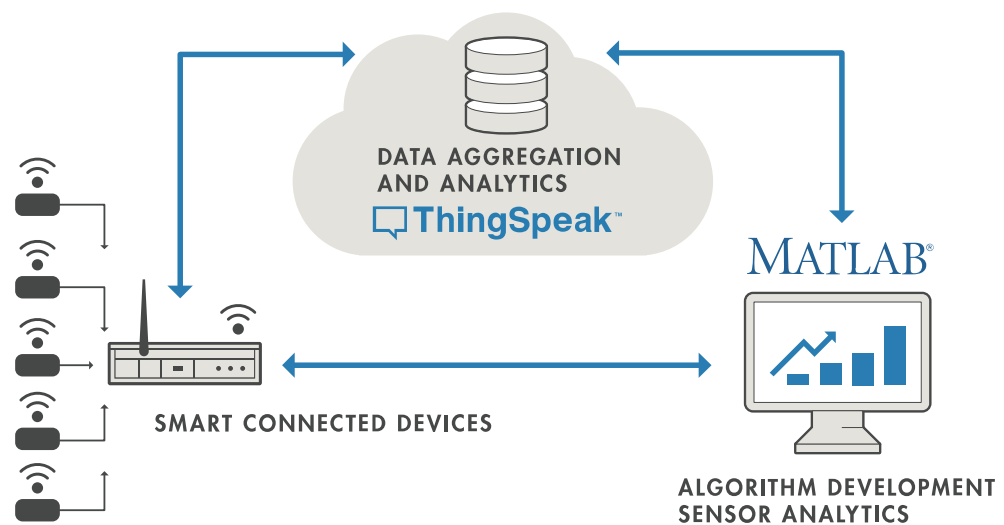


Figure 7.17: ThingSpeak IoT systems [21]

In order to send IoT data over a ThingSpeak channel some rules must be applied. Indeed, there are some limitations with regards to the amount of data that can be transferred over time and there is also a specific data structure to respect.

The first limitation is that a set of data cannot be sent at a faster rate than one every second, so at a frequency of 1 Hz. The second limitation is that only URL encoded characters can be sent through the communication's channel. This means that:

- numerical data have to be split into separate bytes. For example, a double precision floating point value, which is encoded using 64 bits or 8 bytes, must be split into 8 unsigned integer of 8 bits each, so `8 x uint8`
- as represented in the structure of a ThingSpeak channel (7.18), there are eight custom fields that can contain data. Each of them can be filled with a string having a maximal length of 255 characters
- URLs can only be sent over the Internet using the *ASCII*¹ character-set or with a "%" followed by two hexadecimal digits for special/unsafe characters, like *space*

At the beginning of §7.4, it is stated that five signals must be sent over the network. As shown in the data structure (7.18), each of them is assigned to its own field.

¹ASCII = American Standard Code for Information Interchange

It is then required to shape them following the aforementioned constraints so that they are transmitted properly up to the ThingSpeak cloud.

Private View
Public View
Channel Settings
Sharing
API Keys

Channel Settings

Percentage complete 50%

Channel ID 1434455

Name

Description

Field 1	<input type="text" value="Raw"/>	<input checked="" type="checkbox"/>
Field 2	<input type="text" value="Bpm"/>	<input checked="" type="checkbox"/>
Field 3	<input type="text" value="Deviation"/>	<input checked="" type="checkbox"/>
Field 4	<input type="text" value="Valid"/>	<input checked="" type="checkbox"/>
Field 5	<input type="text" value="Reset"/>	<input checked="" type="checkbox"/>
Field 6	<input type="text" value=""/>	<input type="checkbox"/>
Field 7	<input type="text" value=""/>	<input type="checkbox"/>
Field 8	<input type="text" value=""/>	<input type="checkbox"/>

Metadata

Tags

Figure 7.18: Structure of a ThingSpeak channel

A unique channel ID is assigned to each ThingSpeak channel, and two unique API keys are generated and assigned to each of them making their read and write access secured and only allowed to the users having them.

The standalone mode model (6.29) contains the “Network” variant subsystem with the implementation of the “IoT” subsystem sending the required data to ThingSpeak.

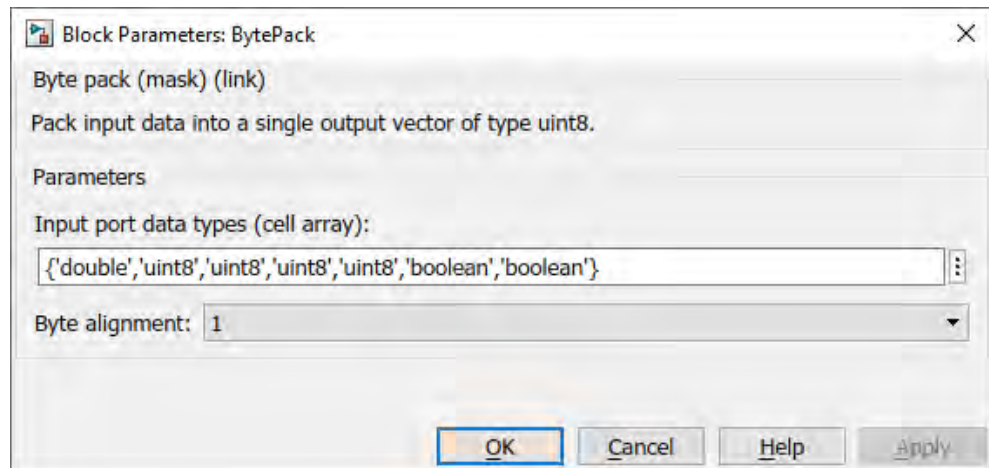


Figure 7.19: Mask of the BytePack subsystem packing data as one uint8 vector

At the “Bytes” input port #1 of the IoT subsystem (7.20) data are coming in from the “BytePack” subsystem present in the standalone mode model (6.29). This subsystem converts its input data into a single output vector of uint8 bytes, which is very useful when data are packed in data frames of whatever communication’s protocol. In the mask of the “BytePack” subsystem (7.19), the end user enters the data type of each input so that it is aware of how to organize them as one single uint8 vector. The byte alignment parameter is set to ‘1’ which means that the smallest entity is a byte. It can also have the values ‘2’, ‘3’ or ‘4’. With a value of ‘2’, this would mean that a Boolean made of one byte only will have a blank byte added right after it, as in this case, the smallest possible entity is of two bytes.

As the *Raw* ECG signal is handled separately for the “IoT” subsystem, via the “Raw” input port #2, its bytes data packed by the “BytePack” subsystem are simply ignored by the “SelectorScalar” block placed right after the “Bytes” input port #1. In other words, the bytes representing the ‘double’ data type signal are not selected. Also two ‘uint8’ bytes from the BytePack mask (7.19) representing the “BpmMin” and “BpmMax” signals are ignored as they are not sent to the ThingSpeak cloud.

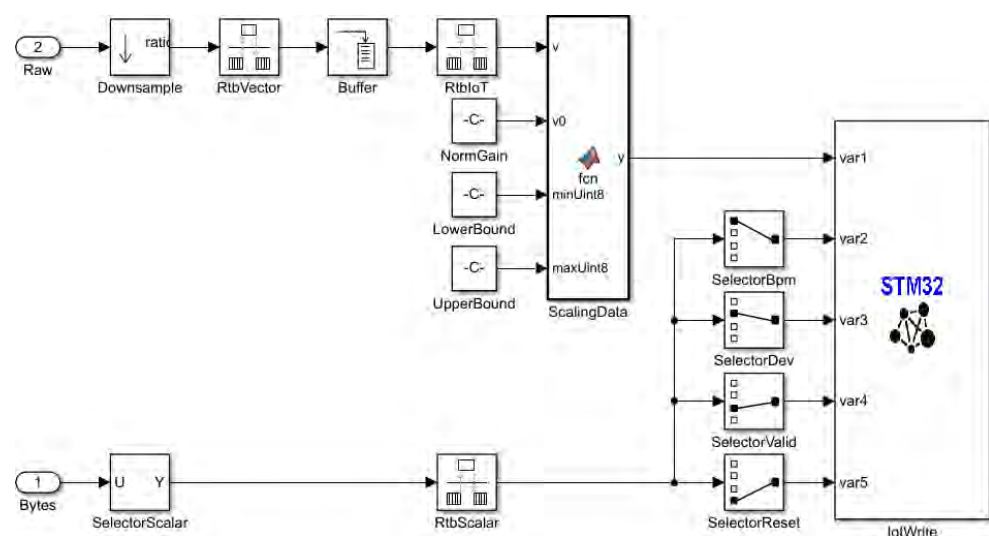


Figure 7.20: Algorithm of the IoT subsystem sending data to ThingSpeak

The inputs “var2” to “var5” of the “lotWrite” subsystem of (7.20) represent respectively the “BpmMean”, “BpmDev”, “Valid” and “Reset” signals that have been byte packed and are seen as “scalar values” by the “lotWrite” subsystem. So that the signal that is provided to the “var1” input is also considered as a “scalar value”, it requires a reshaping of the “Raw” signal.

The “Raw” signal is updated at the sampling frequency of 510 Hz (cf.(6.1)). It is a double precision data type signal, so it is made of eight bytes per sample. That means that there are per second $510 \cdot 8 = 4'080$ bytes to send on the network. These are too much bytes to send than what is allowed for a single field within a ThingSpeak channel (255 bytes/second). Therefore, the amount of data to send for the “Raw” signal must be reduced. This is what is done between the “Raw” input port #2 and the “var1” input of the “lotWrite” system object in the IoT subsystem (7.20).

As the 510 double precision data samples per second cannot be kept, the first transformation that is done is to downsample the “Raw” signal by a ratio of at least 2, so that the sampling frequency changes from 510 to 255 Hz. As the maximal number of bytes to send per second in a single field within a ThingSpeak channel is of 255, it would work. However, it has been stated earlier in this section that only ASCII characters can be sent over the Internet in a URL encoded format. The ASCII encoding uses 7 bits, so that makes $2^7 - 1 = 127$ possible characters (the *null* character '0' cannot be used as well). All encoded characters between 2^7 and $2^8 - 1$ (128 to 255) are not allowed and not understood on the ThingSpeak server side. So, it is physically possible to send 255 bytes per second per field, but only 127 unique values can be used to represent the encoded data. This means that if 255 ASCII characters are sent, one unique value is always repeated twice, so the encoded URL character is duplicated which is not very useful and efficient. As a results the downsampling ratio N has been set to 4 so that 127 unique ASCII values can be sent through the communication's channel. Because, of this sampling frequency decrease, a “Rate transition” block is added right after the “Downsample” one to properly handle the sampling frequency change.

The next step is to buffer multiple samples together before sending them as a vector of ASCII chars. In this case, 127 ASCII values are sent and so is the size of the buffer. At that point, the data type of the values is always of double precision, so each sample is made of eight bytes. That means that $127 \cdot 8 = 1'016$ bytes are sent per second on the network. To further reduce the amount of data to send, these double precision values are converted into unsigned integers of 8 bits (uint8), and are therefore represented by only one byte each. Obviously, going from a 64 bits representation down to a 8 bits one implies a significant loss of resolution. However, the ADC converting the ECG signal from the data acquisition board (4.3) has a resolution of 12 bits (cf.4.1.1). This means that the loss of data is less dramatic than what it seems at a first sight. Indeed, the ADC provides values going from 0 to $2^{12} - 1$ (4095). At the end, the data resolution factor R is of $4096/127 \cong 32.25$. This data compression is done inside the MATLAB function block “ScalingData”, and eventually its output is provided to the “var1” input of the “lotWrite” subsystem.

The total compression factor K (7.1) is given by the downsampling and resolution factors N and R . The impact of this data compression is discussed in §9.2.

$$K = N \cdot R = 4 \cdot \frac{4096}{127} \cong 129 \quad (7.1)$$

The “IoTWrite” system object consists of sending data packets over the Internet via a ThingSpeak’s channel. Its mask (7.21) allows the end user to insert the channel’s related information such as the ID of the channel, the API key to write in it, the number of variables to send or fields to use (five in this case) and also what is the minimum update interval in seconds knowing that the smallest possible one is of one second. An effect to also consider is the time jittering because of the Internet connection. As packets do not all travel at the same speed and do follow different paths on the Internet, there is some time jittering. On the ThingSpeak server side, it is of about 3%. To be on the safe side, a jitter of 5% is taken into account in this subsystem to define the minimum update interval U_T (7.2).

$$U_T = 1 + 0.05 = 1.05 \text{ second} \quad (7.2)$$

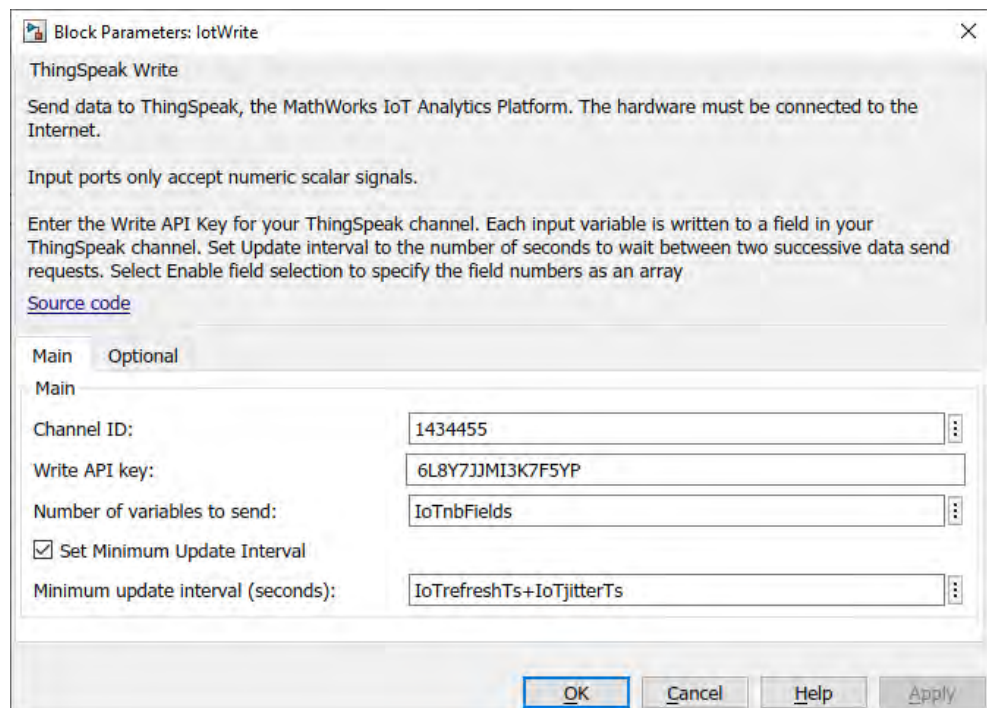


Figure 7.21: Mask of the IoT subsystem sending data to ThingSpeak

The code of the function ‘writeFieldsData’ (7.22) belonging to the “IoTWrite” system object shows how ‘scalar’ and especially ‘vector’ data are processed before being sent onto the channel. If its input ‘fieldData’ is a scalar, the low-level C function ‘MW_addFieldScalar’ can directly be called. It will simply transform the value in a set of uint8 bytes before transmitting it. However, if ‘fieldData’ is a vector of uint8 values, they are first converted into an array of char and processed by the urlencode() function. This one converts all unsafe ASCII chars to their URL encoded value made of a “%” followed by their corresponding two hexadecimal digits, like for example %20 for the space character. The remaining safe ASCII chars made of the following

regular expression subset `(-a-zA-Z_0-9.)` remain as is. As characters are coded in the UTF-8 format, it can be that some bytes have values greater than the last ASCII code 127. In this case, the `utf8ToNative()` function adds an extra byte with the value 194 in front of it, following the conversion to the native character format.

```
function [] = writeFieldsData(fieldData,fieldIndex)
    % Initialization
    nbBytesMax = 255; % Max number of bytes allowed per ThingSpeak field
    nbElements = numel(fieldData);
    % Process the field data based on its size
    if ((nbElements > 1) && (nbElements <= nbBytesMax)) % Vector
        charArray = cell(1,nbElements);
        for n=1:nbElements
            charArray{n} = char(fieldData(n));
        end
        char2send = [charArray{:}];
        data2send = urlEncode(char2send);
        % URL encoding of the
        coder.ceval('MW_addField',data2send,fieldIndex);
    elseif (nbElements == 1) % Scalar
        coder.ceval('MW_addFieldScalar',fieldData,fieldIndex);
    else % Empty data or vector size not supported by ThingSpeak
        error(message('stmmbed:blocks:ThingSpeakFieldMismatch'));
    end
end
function res = urlEncode(charsUtf8)
    if isempty(charsUtf8)
        res = '';
    else
        % Regular expression for the subset of chars that cannot be sent directly
        % notDirect = '([^\s -a-zA-Z_0-9.\~ '])';
        % ASCII code of the corresponding subset, handle 37 => '%' chars first
        notDirect = char([37 (1:1:36) (38:1:44) 47 (58:1:64)...
            (91:1:94) 96 (123:1:125) (127:1:255)]);
        % Convert from UTF8 to native characters representation
        res = char(utf8ToNative(charsUtf8));
        for k=1:numel(notDirect)
            if contains(res,notDirect(k))
                % Replace unsafe chars with their %xx representation
                res = strrep(res,notDirect(k),['%' dec2hex(notDirect(k),2)]);
            end
        end
        res = [res char(0)]; % Concatenate the null char at the end of the chain
    end
end
function native = utf8ToNative(u8code)
    % Initialization
    nbBytes = numel(u8code);
    native = zeros(1,2*nbBytes,'uint8'); % Maximal possible number of bytes
    j = 0;
    % Look for bytes greater than the ASCII code 127 and insert the extra byte 194
    for i=1:numel(u8code)
        j = j+1;
        if (u8code(i) >= 2^7)
            native(j) = uint8(194);
            j = j+1;
        end
        native(j) = u8code(i);
    end
    % Remove useless trailing bytes
    native(j+1:end) = [];
end
```

Figure 7.22: Writing mechanism to the ThingSpeak's data fields in the system object

The Ethernet settings (7.23) of the standalone model (6.29) deployed onto the embedded system are different than the ones set for the UDP and TCP/IP local network connection (7.16). This time the edge node is represented by a router which is connected to the Internet, and the STM32 Nucleo board (4.2) is locally connected to the router. That is why its private IP address is '192.168.0.25' and the gateway field has the router IP address '192.168.0.1'. The router gets a public IP address from an Internet provider so that it is connected to the Internet and forward the data coming from the board to the ThingSpeak cloud.

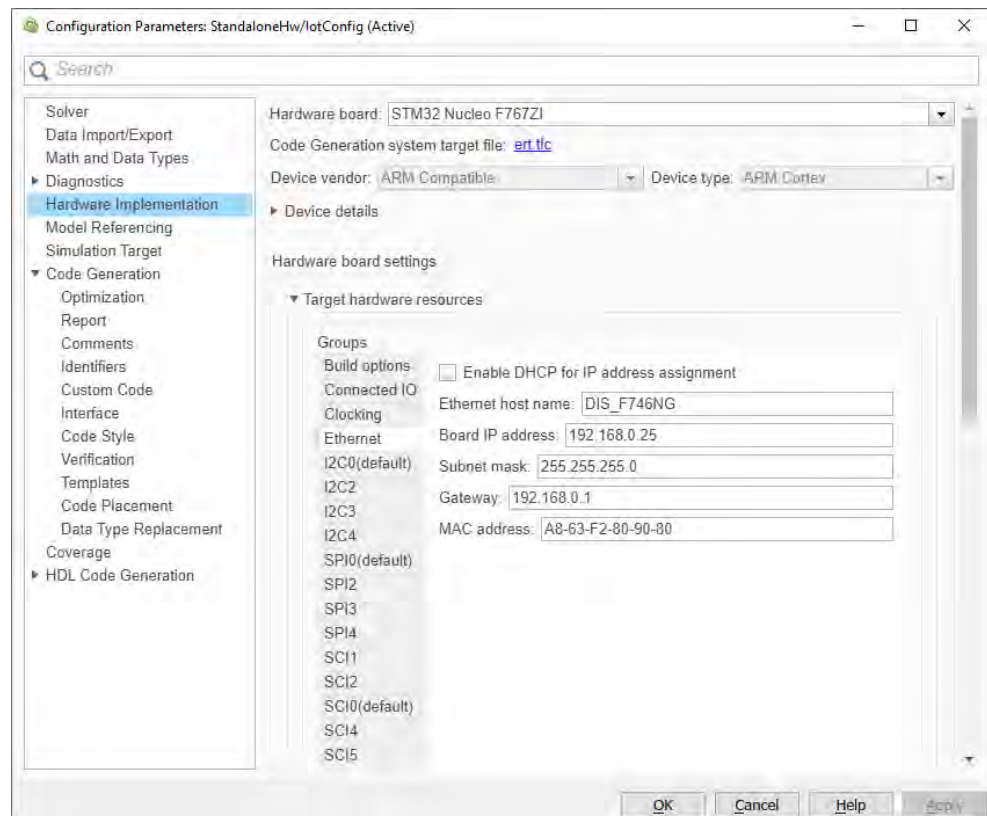


Figure 7.23: Ethernet configuration of the embedded target for IoT connectivity

Both Ethernet settings (7.16) and (7.23) are used for the same standalone model (6.29). The selection of one or the other configuration is handled via the variant mechanism explained at the beginning of §7.

Once the communication's channel is established between the edge node and the ThingSpeak cloud, data are transmitted at the pace defined by the update interval U_T (7.2). As introduced in §6.2.1, a measurement of at least 30 seconds is required to get a valid beats per minutes (BPM) result.

ThingSpeak provides an interface, that can be accessed via a web browser to visualize data on-the-fly and also process and export them. The web interface (7.24) provides pre-defined plots to graphically display results. Some parameters can be set, like the update interval and the data range for example. In this case, the plots of scalar signals (mean, deviation, valid and reset) are refreshed every second and do not require any specific post-processing of the collected data.

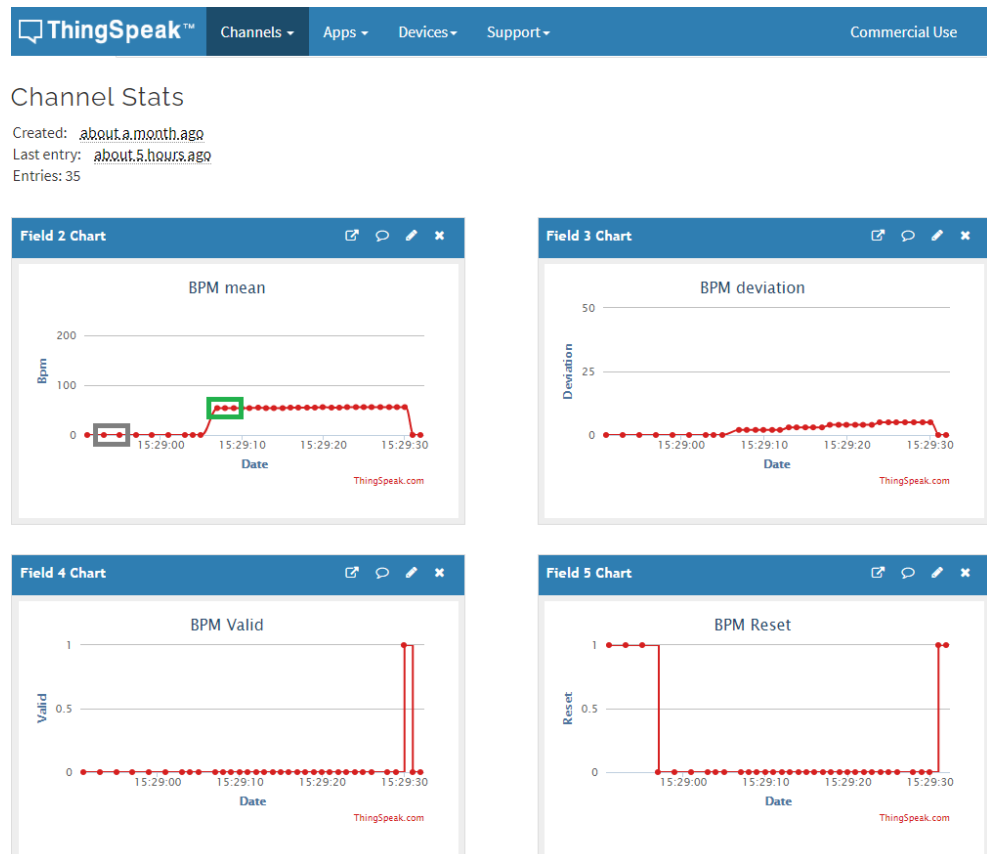


Figure 7.24: ThingSpeak web interface to manage collected data

For this specific measurement of the ECG signal #6, the results provided by the web interface (7.24) show that 35 data points or entries have been collected, which makes a measurement of at least 35 seconds. This is confirmed by the time tags ThingSpeak automatically adds on the time axis of each plot.

On the edge node, it is clear that the data samples are output a regular pace of 1.05 second (cf. (7.2)). However, on the operational technology node, if a packet does not arrive at the expected time, it is ignored by ThingSpeak. The probability is not null for such a situation to occur, because of the time jittering over the Internet and the ThingSpeak server (located in the United-States) handling many parallel connections at the same time.

For the scalar measured signals, it is not that problematic if there is a new sample every two seconds instead of every second. The end result is still accurate enough. Indeed, in the plot of the “BPM mean” value, the **green rectangle** contains three consecutive values, one every second, which is correct. In the same plot, the **gray rectangle** contains two values with a third one missing between the two. This means that one packet did not arrive within the expected time window and it has been rejected by ThingSpeak. However, when it comes to the measure of the raw ECG signal, losing one sample is much more problematic as it consists of a vector containing 127 values. In the case of this measurement, the last 29 samples have been collected correctly without interruption which is good to provide a nice representation of the raw ECG signal as shown in the figure (7.25).

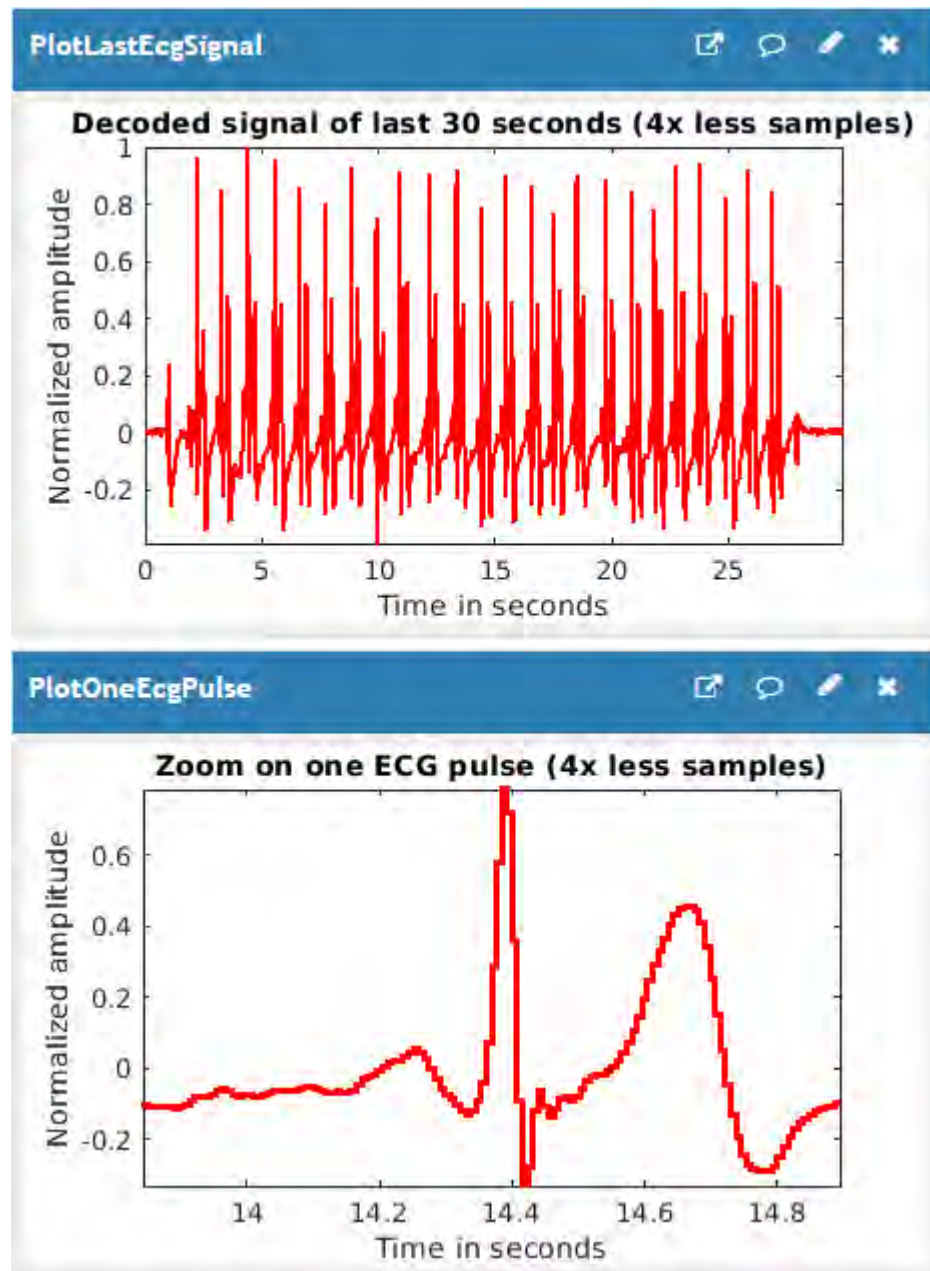


Figure 7.25: Raw ECG signal #6 displayed by the ThingSpeak web interface

The ECG signal and pulse plots provided by the ThingSpeak web interface (7.25) are custom visualizations obtained by writing specific MATLAB scripts decoding the ECG vector data. This means that the received frames of chars are first URL decoded, then expressed in the UTF-8 format, converted into uint8 numbers and the amplitude of the recovered signal is finally normalized with respect to its maximal value. The zoom on a single pulse is done by focusing on a high frequency peak located in the middle of the measured data.

A comparison on the quality of the received signal with respect to the original one is done in the §9.2. To see if the received signal would produce the same ECG results than the original one, it is also passed through the full MATLAB signal processing chain as described in §6.4.

8 Profiling

One of the main goal of this Master thesis is to see the impact of the Mbed hardware abstraction layer (HAL) onto the system and its development. This is done based on the analysis of quantitative data measured directly in the code in a static way, but also dynamically onto the hardware running it.

The peripheral drivers are implemented using two distinct approaches. The first one consists of implementing them at the lowest possible level, by interacting directly with the hardware registers; these are called bare metal or target specific drivers for the Stm32 platform. The second approach consists of implementing them at a higher level of abstraction using the Mbed HAL to be target agnostic (cf. §3.2).

These two approaches allow to see what are the pros and cons of using or not the Mbed HAL. A profiling of the system is done by comparing both implementation techniques with static and dynamic performance criteria that are:

- **Static code metrics**

The number of code lines, the number of function calls, the size of the program code in FLASH memory, the size of the used RAM memory for initialized data, as well as uninitialized data within the memory for dynamic memory allocation

- **Dynamic execution metrics**

The average and maximum task execution time and CPU utilization percentage

8.1 Static code metrics analysis

Within the ARM processor of the embedded system, there are various memory areas. The comparison of their level of utilization when the Mbed HAL is used or not provides some information on the memory footprint of the Mbed HAL. A detailed representation of it is given in the table (8.1).

Mbed HAL used	Number of text lines	Number of code lines	“text” in bytes	“data” in bytes	“bss” in bytes	“dec” in bytes
yes	29'358	20'855	135'600	18'064	68'424	222'088
no	27'792	19'878	134'328	16'688	68'464	219'480
difference	1'566	977	1'272	1'376	-40	2'608

Table 8.1: Comparison of static code metrics with or without the Mbed HAL

The table (8.1) contains the information about the number of lines of text and code. The difference between the two is that the number of code lines simply ignores all lines that are comments (in C they can be identified with the tags `//` or `/* */`). The counting of the number of text and code lines stops when the code in source files (.c/.cpp and .h) is hardware specific; meaning when the Stm32 Low-Level layer or the Mbed HAL layer contents is accessed. At that point, the code is hardware specific and is the same for both cases, which would not provide additional information.

The difference with regards to the number of code lines between both implementations is of 977. Therefore, there is 4.9% more lines of code when the Mbed HAL is used. This is normal as an extra software layer is added between the application and the low-level drivers. Regarding the size of the generated C code, the impact of the Mbed HAL remains minimal in this case.

The table (8.1) is made of three distinct memory sections. The first one that is stored in FLASH is called “text”. This is where the program code containing the instructions, the constant data and the interrupt vector table is stored.

The second memory section that is stored in RAM is called “data”. This is where the initialized data are stored. Nevertheless, their constant initialization values are stored in FLASH. In the start-up code, a copy of their initial values from FLASH to RAM is done. This means that the data for such variables is counted twice; once in RAM and once in FLASH. However, it is not counted within the “text” section.

The third memory section that is stored in RAM is called “bss”. It means *Block Started by Symbol* and this is where the uninitialized data are stored. In the start-up code, the complete bss section is initialized with zeros.

Finally, the “dec” field represents the sum of the three aforementioned memory sections. The last line of the table (8.1) represents the difference in size between each sections. Based on these values, it is noticeable that the Mbed HAL has a minimal impact on the memory footprint. Overall, it adds 2.608 Kbytes of data which represents 1.19% of additional data to handle and store. It is also interesting to see that for the uninitialized data section, a bit less data are present when the Mbed HAL is used.

In §4.1.1, it is stated that there are 2 Mbytes of FLASH and 512 Kbytes of RAM memory that are available. Based on the data from the table (8.1), only

$$\frac{135'600}{2'000'000} \cdot 100 = 6.78\% \text{ of the FLASH memory and}$$

$$\frac{86'488}{512'000} \cdot 100 = 16.89\% \text{ of the RAM memory} \quad (8.1)$$

are used when the Mbed HAL is present (this can be seen as the worst case scenario).

By looking at the generated C code for the full model, it is already important to highlight the fact that the C code for all variants within the model are generated. Indeed, it is only at the compilation or binding time that the desired variants are kept and others are removed for the generation of the binary files for deployment. This procedure is called *derivation* or *instantiation* in the product line engineering approach.

Based on the static code metrics analysis done in this section, it is clear that the usage of the Mbed HAL has an impact on the generated C code and its usage on the FLASH and RAM memories. Nevertheless, this impact is rather low for this electrocardiogram (ECG) application, which is an advantage when it comes to do rapid prototyping.

8.2 Dynamic execution metrics analysis

In this project, there is one main task, running at 510 Hz, handling the execution of the signal processing algorithm that continuously processes the incoming ECG signal. When the processed data are sent over an Ethernet network, other tasks are also executed. Therefore, the dynamic profiling focuses on all these tasks and their utilization of the processor. These two characteristics allow to see how much overhead is present when the Mbed HAL is used.

The profiling of the running code helps to determine if the generated code meets the execution time requirements of the developed application when it is deployed onto a real-time embedded system. It also provides inputs on code sections that could potentially require some execution speed improvements. This type of run-time profiling is generally done in External/Processor-In-the-Loop mode.

Function executed	Mbed HAL used	Maximum execution time in μs	Average execution time in μs	Maximum % CPU utilization	Average % CPU utilization	# of function call
initialize at the start	yes	196.0	196.0	-	-	1
	no	181.0	181.0	-	-	1
terminate at the end	yes	6.0	6.0	-	-	1
	no	6.0	6.0	-	-	1
step task 0 at 1020Hz	yes	10.0	6.6	1.020	0.674	144
	no	9.0	6.8	0.918	0.694	240
step task 1 at 510Hz	yes	64.0	36.1	3.264	1.839	73
	no	37.0	32.0	1.887	1.630	121
step task 2 at 127.5Hz	yes	5.0	4.4	0.064	0.056	20
	no	5.0	4.1	0.064	0.052	32
step task 3 at 1Hz	yes	34.0	34.0	0.003	0.003	1
	no	34.0	34.0	0.003	0.003	1
total CPU utilization	yes	113.0	81.1	4.351	2.572	-
	no	85.0	76.9	2.872	2.379	-

Table 8.2: Comparison of dynamic execution metrics with or without the Mbed HAL

Out of the profiling data gathered in the table (8.2), it is already good to see that when the Mbed HAL is used at both ends of the signal processing algorithm, the maximum utilization percentage of the CPU is of less than 4.5%. This means, that even if the Mbed HAL is used, there are still more than enough processing power available to run other tasks that would be much more demanding, like a human machine interface (HMI) or a graphical display for example. Moreover, running a real-time operating system (RTOS) like Mbed OS (cf.§3.1) would be possible as well. The CPU utilization corresponds to the relative amount of time or percentage of CPU time assigned to one task. Its value is computed by dividing the task execution time by the sample time (which is the inverse of the sampling frequency FS (cf.(6.1))).

On the other hand, it is noticeable that the usage of the Mbed HAL does add some overhead on the task execution time with regards to the direct low-level Stm32 implementation. This overhead τ_o (8.2) can be quantified for the worst case scenario which is the sum of the maximum execution time of all four step functions with the Mbed HAL, minus the sum of the maximum execution time of all four step functions without the Mbed HAL.

$$\begin{aligned}\tau_o &= \sum_{k=0}^3 \tau_k^{Mbed} - \sum_{k=0}^3 \tau_k^{Stm32} = (10 + 64 + 5 + 34) - (9 + 37 + 5 + 34) = 113 - 85 \\ &= 28 \mu s\end{aligned}\quad (8.2)$$

At the end, the computed overhead (8.2) is normal and such results were expected. Nevertheless, the usage of the Mbed HAL has a non-negligible impact on the processor utilization. The additional utilization U_{add} it overall takes is given by the expression (8.3).

$$U_{add} = \left(1 - \frac{85}{113}\right) \cdot 100 = 24.78\% \quad (8.3)$$

This means that the usage of the Mbed HAL adds a task's execution overhead of one fourth with regards to the Stm32 low-level implementation. When it comes to do rapid prototyping on a portion of a complex system, or on an application like this ECG processing, it is fine. At this stage, there would be no direct need to migrate from the Mbed HAL to the low-level Stm32 implementation. However, if other software components are going to be part of the system, this 25% overhead can be problematic and should not be ignored. For the end users such information is very important, as they are aware that there is still some room for optimization if required later in the development phase of the final production product.

In the C code generation process, three main tasks or functions are always created: the initialize(), terminate() and step() functions. By looking a bit more to the details of the table (8.2), the initialize() and terminate() functions can be easily identified and are executed only once, respectively at the beginning and the end of the program execution. The step() function is divided into four separate tasks called "step0", "step1", "step2" and "step3" working at different sampling rates that are respectively of 1'020, 510, 127.5 and 1 Hz. This means that this is a *multirate system*. Indeed, this is one of the most important advantage of Simulink to be able to manage multiple rates within the same system. In this ECG application, the multirate subsystems are mainly located in the "Network" variant subsystem, and especially in its "IoT" implementation for the communication with ThingSpeak.

Simulink allows to highlight the different rates present in a model as shown in the IoT subsystem (8.1). The step0 task runs at the base rate of 1'020 Hz and is the scheduler task of the system. Indeed, it is the greatest common denominator among all the other tasks, or in other words, it has the fastest sampling rate of the model. The step1 task runs at the main rate of the model which is of 510 Hz as defined in (6.1) and is highlighted in **green**. After the downsample block, the sampling frequency is reduced by a factor of 4 to reach 127.5 Hz and is highlighted in **blue**.

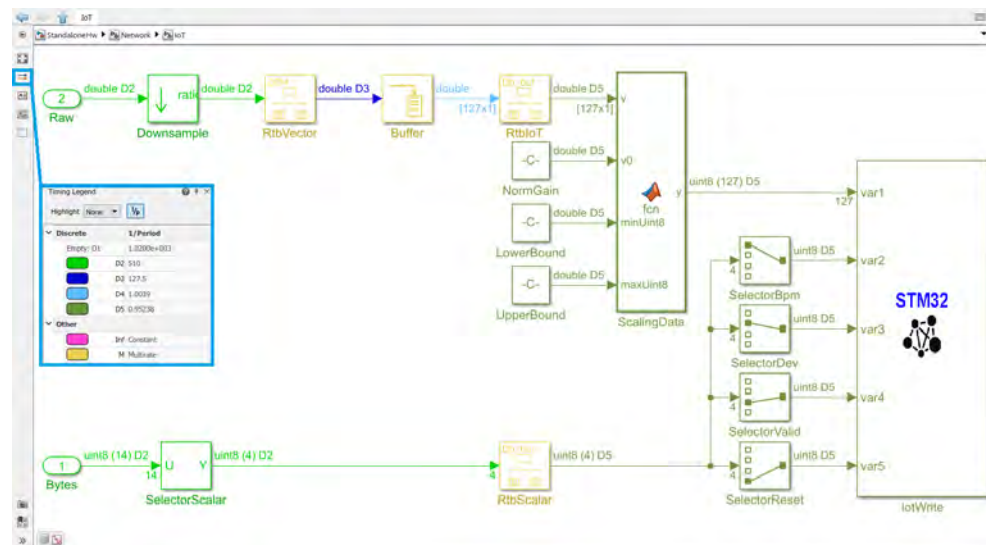


Figure 8.1: Multirates within the IoT subsystem

After the buffer block, the sampling frequency reaches 1 Hz as 127 data samples are stacked together. This frequency is highlighted in **cyan**. Finally, the data are sent to the ThingSpeak channel at a rate a bit smaller than 1 Hz to take the time jittering (7.2) into account. Its sampling frequency is highlighted in **olive**.

The **yellow** blocks handle multiple rates at the same time. For example, the *rate transition blocks* (RTBs) help to transition from one sampling rate to another via various mechanisms like semaphores, unit delay, double or triple buffers and so on. One or the other method is selected based on the activation or not of two parameters called “Data integrity” and “Deterministic data transfer”.

8.3 Discussion on profiling results

By looking at the static and dynamic profiling metrics, the first observation that stands out is that the resources provided by the STM32 board and their characteristics described in §4.1.1 are accessed and used with parsimony even when the Mbed HAL is present. According to the memory consumption values (8.1), less than 7% of the FLASH and less than 17% of the RAM are used. Moreover, the overall CPU utilization is, in the worst case scenario, of less than 4.5% according to the table (8.2). This allows the end users to quickly develop a prototype of the multirate system without fiddling around with what is hardware specific and abstracting it with the Mbed HAL. This has the advantage of increasing the productivity in the development of such products.

The generated code is efficient, because the algorithm has been designed following a step-by-step approach starting from its initial investigation defining its structure and parameters with the help of simulation up to its integration with the embedded system. Moreover, at the integration phase, code replacements libraries (CRLs) are used to take advantage of the specific ARM Cortex-M architecture and optimize the generated C code.

9 Results Analysis

The aim of this chapter is to focus on the accuracy of the produced results and their consistency through the various development steps. It is of paramount importance to be able to compare the obtained results once the code is deployed onto the embedded system with the simulation and reference results.

As the produced data are sent over a local network or on the Internet, it is also key to ensure that the received data are correct and can be reused for further processing remotely (onto a desktop computer or in the cloud).

9.1 Comparison of numerical results

In this project, the verification and validation of the application is done via the analysis of the produced numerical results based on defined test signals. The idea is to ensure that the numerical results, out of the electrocardiogram (ECG) measurements, are similar for the different implementations that have been created through the development process.

In practice, this is not enough to prove that the deployed algorithm works correctly. Indeed, model and code coverage as well as a static analysis of the model and code must be done to ensure that the control and data flows are correct. Moreover, some checks at the model and code levels are requested by safety standards, like the IEC 62304 standard mentioned in §2.2. On top of that, a full bi-directional traceability among the requirements, model and code is required. In this project, the product's requirements have not been formally authored and cannot be traced as such. All the aforementioned verification and validation steps have not been done in this project as it would have requested much more time than what is available to do it.

ID _{ECG}	Count	Watch	MATLAB	Simulink	External	Standalone	Δ_{ECG}
1	92	91	91±6	90±5	90±5	90±5	2
2	70	69	70±5	69±5	69±5	69±5	1
3	70	71	71±5	70±5	70±5	70±5	1
4	68	67	68±6	67±5	67±5	67±5	1
5	70	71	71±6	70±5	70±5	70±5	1
6	56	56	56±5	56±5	56±5	56±5	0
7	90	90	92±19	91±9	92±7	92±7	2
8	70	70	71±5	70±5	70±5	70±5	1
9	99	99	100±8	100±7	100±7	100±7	1
10	78	78	79±5	78±3	78±3	78±3	1

Table 9.1: Comparison of numerical results among all implementations

In the numerical results table (9.1), there is a comparison of the beats per minutes (BPM) results for each ten ECG signals. The manually counted ECG pulses as well as the BPM results produced by the smart watch are taken as reference values.

The results produced by all three implemented algorithms (in MATLAB, Simulink and STM Nucleo deployment) must be verified so that the application can be validated. This is requested by the IEC 62304 standard. This is called “back to back testing”.

The first observation out of the numerical results table (9.1) is that the obtained BPM values are similar. For the ECG signal #6, the results are even fully identical. The maximal delta, with respect to the reference count value, that is provided on the right-hand side of the table shows that the maximal variation that can be observed is of 2 BPMs. This happens for the ECG signals #1 and #7 that have quite some noise and glitches in them to test the robustness of the signal processing algorithm (6.5).

The fact that there are very little variations between the results in MATLAB, Simulink and the STM Nucleo deployment is normal. Indeed, in all these cases, going from one representation to another is not a one to one transition. For example, in the MATLAB implementation, the processing of the data is done at once on the ECG frame/vector signal. Whereas, in the Simulink simulation, the processing of the data is done sample by sample one after the other at each simulation step. When it comes to the implementation of the deployment model, it works similarly to the simulation model, but this time the connectivity with hardware peripherals must be handled with or without the Mbed hardware abstraction layer (HAL). For example, in the case of the signal acquisition via the hardware ADC, the resolution of one sample is of 12 bits and not 64 bits like for a double in simulation mode. This adds some noise and inaccuracy to the signal.

Based on these observations, the results from the table (9.1) proves that the signal processing algorithm (6.5) has been designed correctly. The standard deviation or uncertainty on the results is an indicator of the signals quality; the smaller the deviation the better the signal quality. This table summarizes the results obtained at the end of each measurement, when the valid flag is true. However, it does not provide any information about how the BPM mean and standard deviation are evolving during the measurement and its processing to converge to its final value.

In the MATLAB implementation, *frame-based processing* is done over all samples at once, so the evolution of the measurement and its processing cannot be observed. However, when it comes to the Simulink implementation for simulation and the STM Nucleo deployment, *sample-based processing* is done and the BPM result is updated after each new sample basically. Simulink models can be instrumented to measure and log signals on-the-fly at various places inside the signal processing algorithm (6.5). This has been done for the ten ECG signals in simulation, external and standalone deployment modes. The results of the BPM mean and standard deviation can be observed in the figures (9.1) for the ECG signals #1 to #5 and (9.2) for the ECG signals #6 to #10.

In simulation mode, the triggering of the reset event can be automated and repeated in a deterministic way. There are ways to do it in external mode as well. However, for the standalone deployment mode, the end user must physically press on the reset button to trigger a reset event. This implies that there is some jittering on the time axis and that the signals are not perfectly aligned from one run to the other.

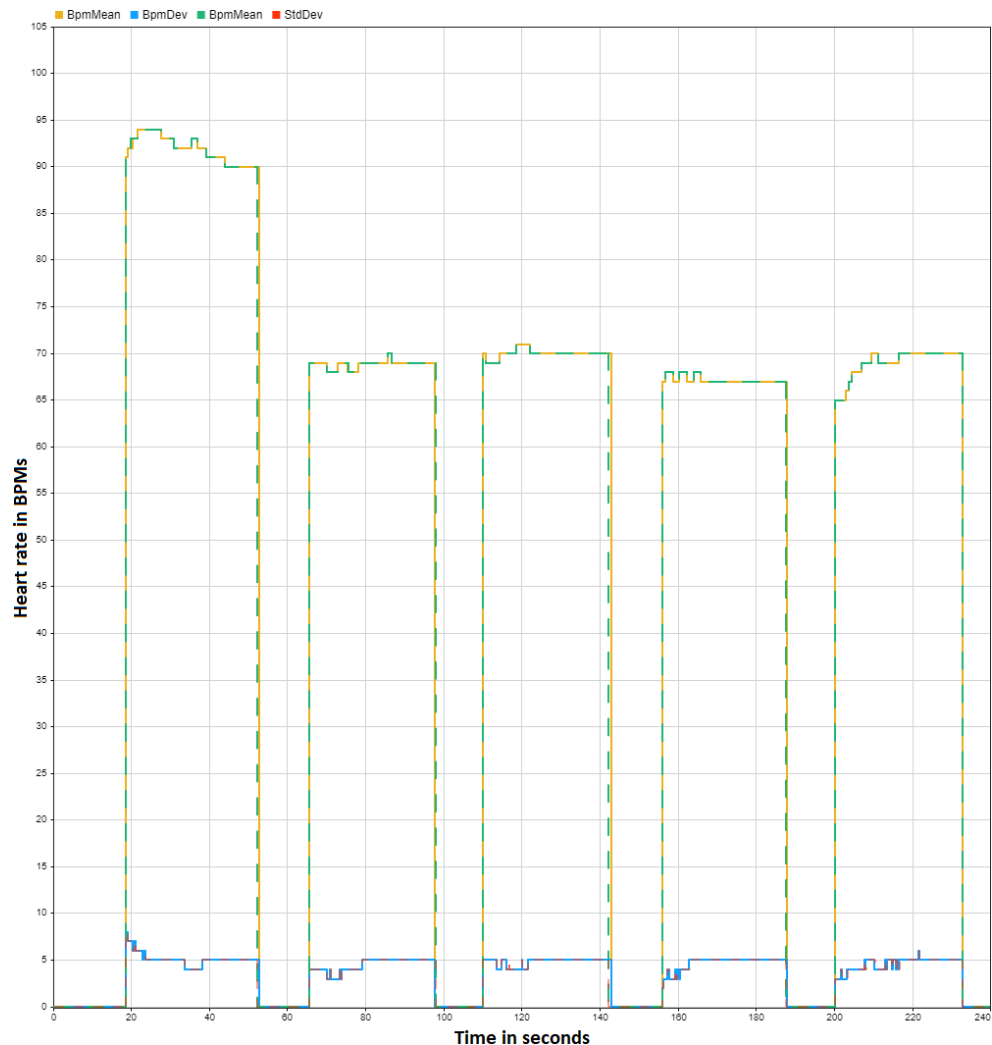


Figure 9.1: Simulation vs deployment modes results for the ECG signals #1 to #5

In the results of the figure (9.1), the green and red dashed lines represent respectively the BPM mean and standard deviation measured in deployment mode onto the hardware. Indeed, the external and standalone deployment modes runs do produce exactly the same numerical results. That is why the comparison results are not repeated for these two modes and is simply referred to as deployment mode. The orange and blue lines represent respectively the BPM mean and standard deviation measured in simulation mode on the desktop computer. Even if there is a bit of time jittering, it is obvious that the simulation and deployment modes produce exactly the same numerical results which confirms that the generated C code for the embedded system behaves the same way than the simulation model.

In the results of the figure (9.2), the color coding of the signals is exactly the same than in the figure (9.1). Here again the remaining signals produce exactly the same numerical results in both modes, except for the ECG signal #7. The two gray rectangles highlight the difference of results between the two implementations. The explanation of this variation during the evolution of the results comes from the fact that the raw ECG signal #7 has a lot of noise as already mentioned in §9.1. In deployment mode, the ADC also adds some extra noise that is not present in the

simulation mode. As there are some rounding blocks in the algorithm, it can be that at one specific point in time, because of the accumulation of noise, one signal is then rounded towards the next upper value instead of the lower value as it is supposed to. By looking at the two **gray rectangles**, this is exactly what happens just before 300 seconds of time.

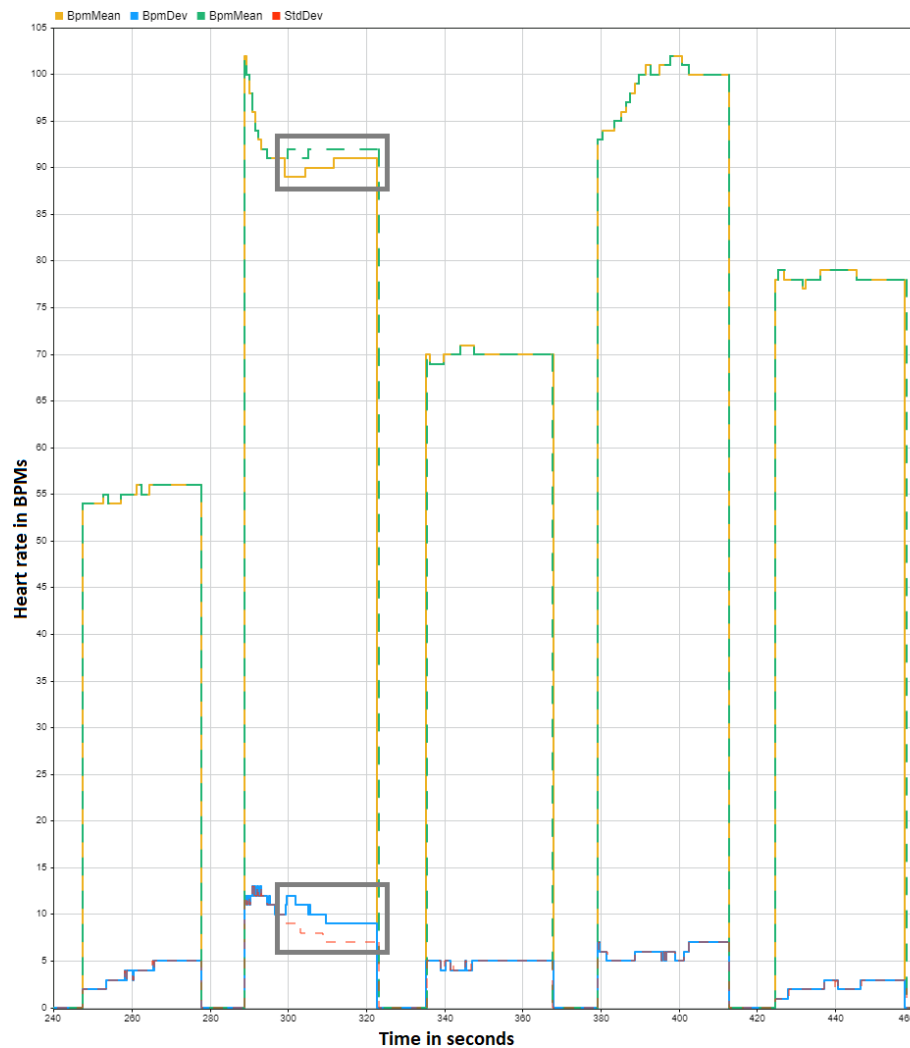


Figure 9.2: Simulation vs deployment modes results for the ECG signals #6 to #10

It is mentioned previously that the external and standalone deployment modes are referred to as deployment mode because both produce exactly the same numerical results. The only difference between the two is how signals are logged. In the external mode case, the Simulink model is instrumented to log signals onto the target and send them to the desktop computer via XCP as explained in §6.6.2.

In the standalone mode case, the Simulink model is not instrumented to log signals onto the target, but is connected to a local Ethernet network via UDP or TCP/IP as explained in §7.4.1. Therefore, it is needed to receive and decode the sent data packets from the embedded system. The Simulink model (9.3) has been created to get Ethernet data packets. The “Network” variant subsystem allows to select either the UDP or TCP/IP communication’s protocol. Then, the received packets just need

to be unpack correctly to retrieve and display the logged signals.

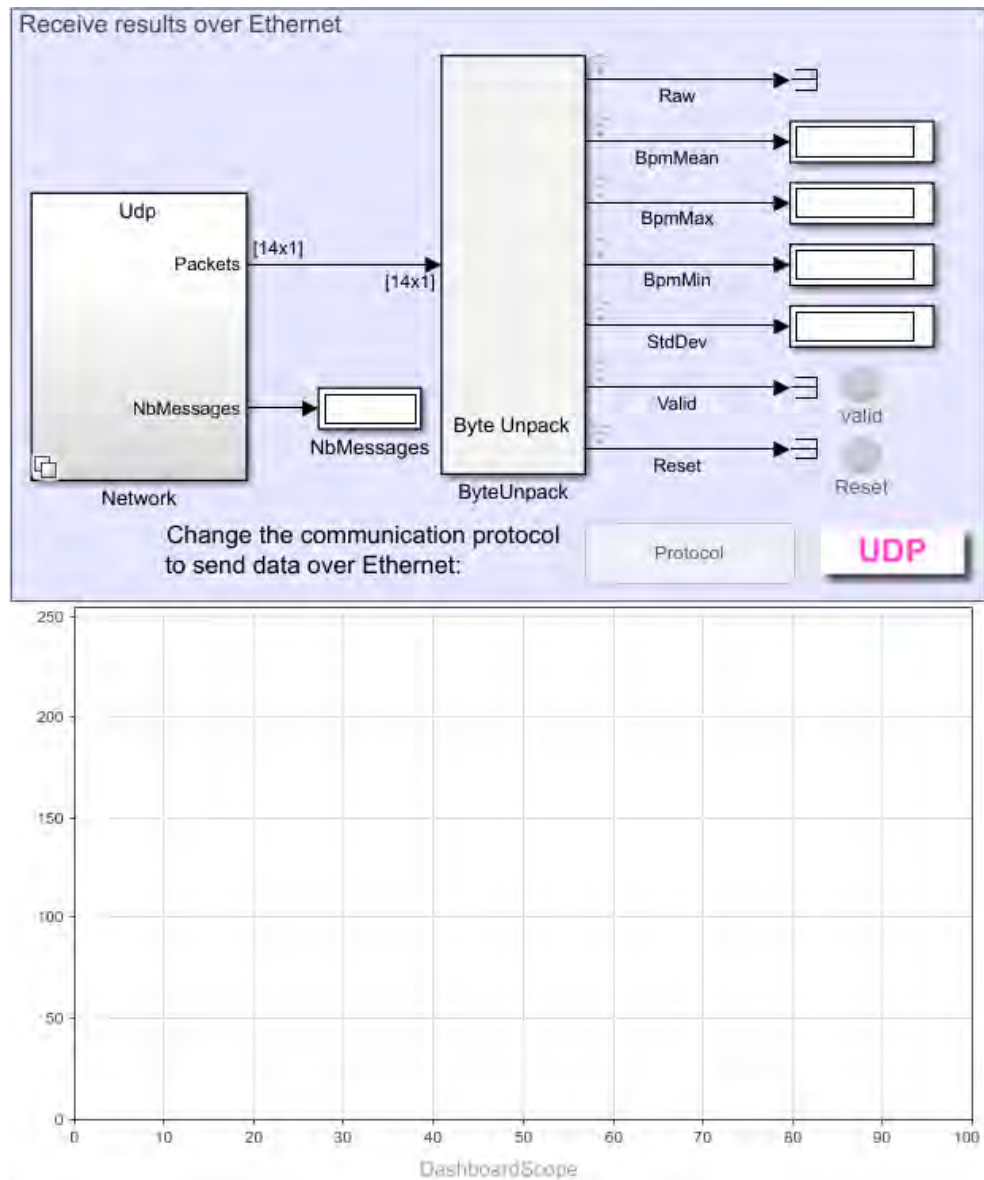


Figure 9.3: Simulink model to gather Ethernet data coming from the embedded system

In this section, the distinction between the STM32 low-level and the Mbed HAL implementations has not been done because in both cases, the produced numerical results are exactly the same. This was expected, as the way drivers are written should not affect the data.

9.2 ThingSpeak data compression

When it comes to send data to the ThingSpeak cloud, they must be compressed to fulfill the ThingSpeak's data requirements detailed in §7.4.2. Obviously when there is data compression, this means that there will be a loss of data precision and resolution. In this section, the goal is to see the impact of this data compression and if the received data are still usable for further processing.

The graph (9.4) compares the original and the ThingSpeak received ECG signal #6. Basically, compressed data have been sent from the Stm32 board running in standalone mode via a ThingSpeak communication's channel. The data have been decoded with a MATLAB script and stored in a *MATLAB Executable* (MAT) data file to be compared against the original ECG signal.

The first observation is that both signals are aligned in the time domain as shown with the **green rectangle**. As explained at the end of the §7.4.2, if a data packet arrives too late in the ThingSpeak cloud, it is rejected. That is why it is not always guaranteed to receive all the sent data on the operational technology node. Moreover, the time jittering can also interfere with the time alignment. Therefore, it is not especially easy to align the received signal with its original one in the time domain.

Regarding the amplitude values, it can be seen that they are not exactly the same. For the high frequency peaks, they are almost always the same. For the low frequency ones, they follow the same trend, but are overall a bit smaller. This is due to the fact that only one sample every four is sent, so if the top of one peak is not sent, this results in a loss of amplitude's dynamic. The fact that the resolution of the samples goes from 4096 to 127 values, as shown in the expression (7.1), reduces the small signal's oscillations as well.

Overall, the shape of the sent signal is preserved which is the most important if it is requested to do further processing on the data in the cloud.

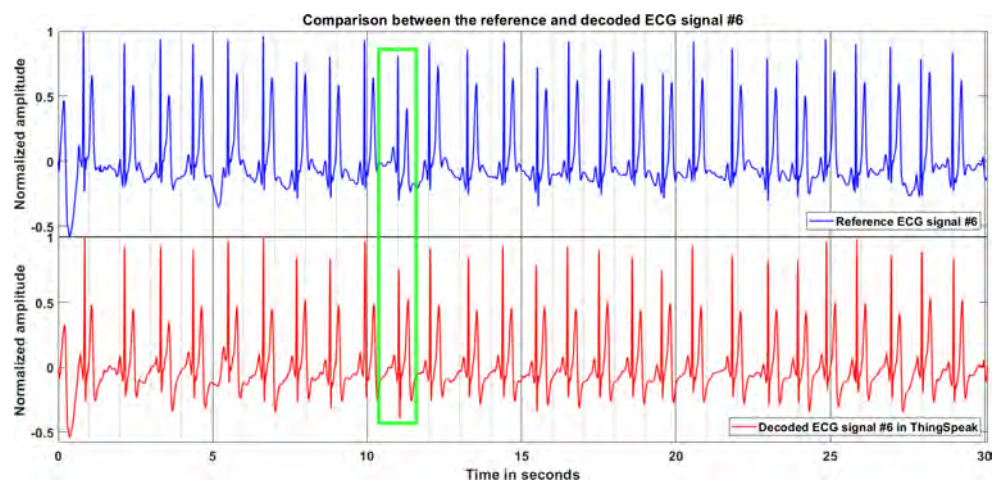


Figure 9.4: Comparison between the original and the ThingSpeak received ECG signal #6

By zooming on the highlighted **green pulses** as in the figure (9.5), it is easier to see to what extent the original and the ThingSpeak received ECG signals are similar. The shape and the amplitude of the received signal are well preserved even after such a compression of the data.

Based on these observations, it is then possible to pass this compressed signal through the signal processing algorithm in MATLAB, described in §6.4, to see if the computed BPM result is the same than the one obtained with the original ECG signal #6 (6.13). As the BPM result of the ThingSpeak received ECG signal #6 (9.6) is the same than the one obtained with the original one, which is of 56 ± 5 BPM, this demonstrates that it is working as expected.

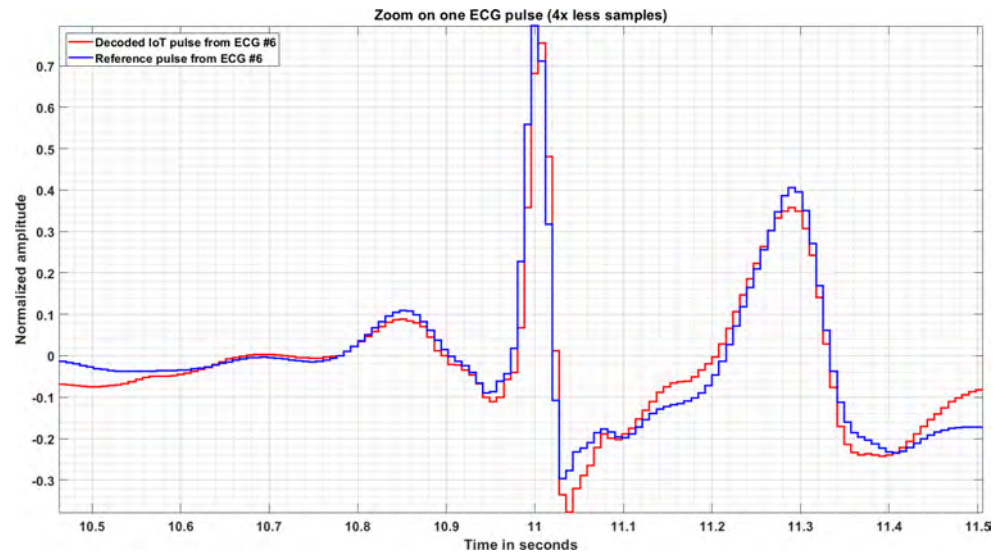


Figure 9.5: Comparison between the original and the ThingSpeak ECG pulse at 11 seconds

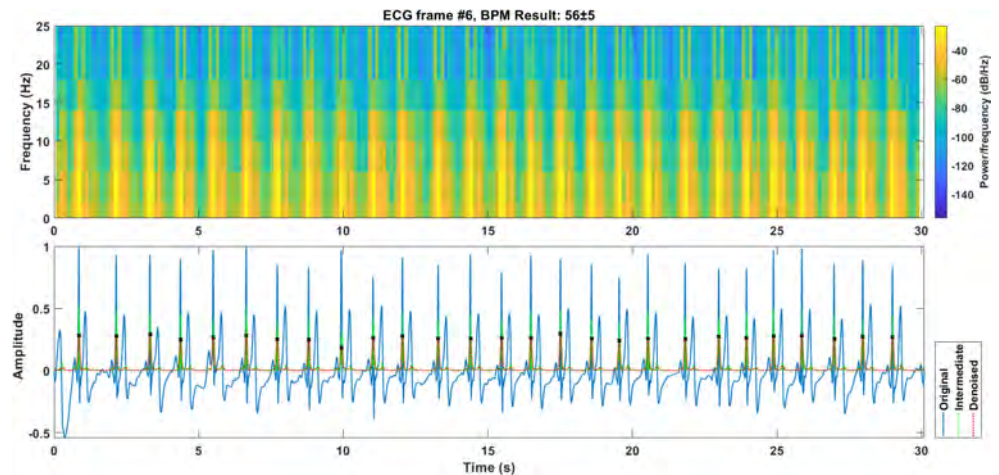


Figure 9.6: BPM results of the ThingSpeak received ECG signal #6

To be able to obtain the BPM result from the received ThingSpeak signal (9.6), a parameter had to be modified in the signal processing algorithm. Indeed, as the sampling frequency of the signal went down by a factor of 4 to reach 127.5 Hz, this means that the Nyquist criteria mentioned in §6.2.2 must also go down up to $127.5/2 = 63.75$ Hz. The only parameter that has a value greater than 63.75 Hz is the second cut-off frequency of the Bandpass Butterworth filter defined in the filters parameters table (6.1) that has a value of 100 Hz. To respect the Nyquist criteria, this cut-off frequency #2 is then saturated at 63.75 Hz.

10 Conclusion

An important aspect of this Master thesis has been to use techniques that allow to abstract complex implementation's tasks from the engineer's view points. It has the advantages of focusing on the development of the algorithm itself rather than on how to program it and to make the engineer more productive. Indeed, one of the main point to highlight in this project has been that it was possible to quickly modify parts of the algorithm and directly test it onto the hardware. Then, if one error or wrong behavior was detected, it was straight forward to adapt the simulation model accordingly to reproduce the observed behavior and correct it in the simulation model. The abstraction of complexity made it easier to go back and forth between the deployment model and its corresponding simulation model.

The abstraction of complexity has been done at three different levels. The first one was to follow a Model-Based Design (MBD) approach by using MATLAB and Simulink. A first investigation of a possible signal processing algorithm has been done in MATLAB, before being implemented as a simulation model in Simulink. After getting satisfying results out of it, a deployment model has been created taking into account the hardware peripherals from the embedded target. Following a product line engineering approach, the reuse of developed components has been done from one model to the other, as well as the usage of variants to make the selection of peripheral drivers easily flexible. In this case, variants have been used to implement various communications protocol for an Internet of things (IoT) application and also to use or not the Mbed hardware abstraction layer (HAL).

The second layer of abstraction was to implement and support the Mbed HAL via Simulink. This Mbed HAL is an initiative from ARM to allow engineers to program multiple ARM Cortex-M targets by reusing the same application's code without having to re-adapt the low-level drivers. It also aims at simplifying the rapid prototyping of IoT applications. In this project, the focus has been put on the Mbed HAL to connect to hardware peripherals. An Mbed real-time operating system (RTOS) is also provided by ARM, but was not used in this project because it would have taken too much time to have it supported in Simulink. The implementation of the Mbed HAL for the peripherals drivers was done by using the MATLAB system objects technology which has the flexibility of mixing MATLAB and C/C++ codes together. It also has the advantage of removing the implementation's complexity of doing all in C code only, like creating the C wrapper code for example.

The third layer of abstraction was to use the automatic C/C++ code generation capability out the Simulink models for deployment. One of the main advantage of it was the time it took to generate code. In average, it took one minute to generate more than 20'000 lines of code, compile, link and deploy them onto the target. Regarding flexibility and productivity, this had an important impact during the rapid prototyping phase. Moreover, the generated code was optimized for ARM Cortex-M based processor by using code replacements libraries (CRLs) to take advantage of

the specific ARM Cortex-M architecture. It can be that an engineer could even more optimized the code manually, but is it worth the effort? Here, the *Pareto principle* can be mentioned. Indeed the engineer would spend 20% of the effort to create a model and generate code that is already 80% efficient. Does it make sense to spend 80% more effort to only gain 20% more efficiency, for example to reduce the number of code lines or some RAM memory space? My personal opinion is that it is not worth such an effort as today the embedded targets have a lot of available resources.

An important aspect of this Master thesis was the usage of the Mbed HAL and what it could offer. Based on the work done in this project, the first advantage it provides is the abstraction of the hardware complexity. A very good example of it is the control of the analog-to-digital converter (ADC) peripheral. This peripheral has multiple modes of operation, but when it is used via the Mbed HAL, only one of them is used and must be implemented, which significantly simplifies its implementation and its usage as well. Another advantage is that the produced code that is added when the Mbed HAL is used remains relatively low; it does not impact a lot the number of code lines and the memory footprint. However, it impacts the task execution time and the utilization of the CPU to some extends, so this is a side effect to keep in mind once the prototyping phase is over and the product enters its development phase for production.

An other objective was to be able to send data from the embedded system or the edge node over a local Ethernet network or to the cloud via the Internet in real-time. This is actually the first mega-trend within the medical devices industry that is called the Internet of medical things (IoMT). An achievement that can be highlighted is the creation of a MATLAB system objects that is able to send scalar and vector data every second from the edge node to the operational node in the cloud via the setup of a ThingSpeak communication's channel. Indeed, this is the very first subsystem in Simulink being able to do this. It will be officially released with the release R2022a in April 2022. Even if the electrocardiogram (ECG) data had to be compressed before being sent to the cloud, they could be decoded and passed through the original signal processing algorithm in MATLAB and right results were obtained confirming that the sending of compressed vector data works correctly.

An analysis of the numerical results as well as a profiling of the static and dynamic metrics has been done to confirm that what was achieved after the various phases of the development workflow produced correct and acceptable results as this is requested by the IEC 62304 standard in the medical devices industry. Based on these analysis, it has been confirmed that the numerical results among the various phases produced similar results. Moreover, the impact of the Mbed HAL could be quantified by quantitative metrics so that an objective discussion could take place on the pros and cons of its usage.

On the MathWorks side, this project had an impact internally as well. The end result is that this project is going to be reused as a tool shipping demo that will be available within MATLAB in the release R2022a. The Mbed implementation as well as the ThingSpeak IoT communication will be there as an example of utilization for customers. Another project that arises from this is the creation of one dedicated

Mbed hardware support package (HSP) with capabilities to program more than 150 processors and boards through Mbed following a model-based design approach. That is really gratifying to see that such a project will continue to grow at MathWorks and be available to our customers. Moreover, some extension's work could be added to this Master thesis, like the added of the Mbed OS support for example.

Finally, this Master thesis was an intense journey with a broad set of topics to cover, to get familiar with and to put in application in order to present what has been learned in the last six months in a clear and concise way.

11 Annex

All project's files allowing the redaction of this Master thesis can be found online at the following OneDrive location: [MbedProject](#). Only people having this link can access the related documents.

12 Bibliography

- [1] ISO Standard (2018). *ISO 26262-6: Product development at the software level*.
[ISO 26262-6:2018](#).
- [2] ISO Standard (2006). *IEC 62304 standard for Medical device software — Software life cycle processes* (update 2015).
[IEC 62304:2006/AMD 1:2015](#).
- [3] Medical Industry User Story (2016). *IMT Developed a highly complex ventilator called Bellavista*.
[MATLAB EXPO Switzerland 2016](#).
- [4] Mbed platform. *Mbed Rapid IoT device development*.
[Mbed website](#). Accessed: 30 April 2021.
- [5] Arnab Ray, Raoul Jetley, Paul L. Jones, Yi Zhang (2010). *Model-Based Engineering for Medical-Device Software*.
[ResearchGate](#).
- [6] L. Hofland and J. van der Linden (2010). *Software in MRI Scanners*.
[IEEE Software](#), 10.1109/MS.2010.106.
- [7] MathWorks White Paper (2020). *Developing IEC 62304–Compliant Embedded Software for Medical Devices*.
[Model-Based Design IEC-62304 for medical devices](#). Accessed: 30 April 2021.
- [8] Mbed Example Docs (2021). *Developing IEC 62304–Compliant Embedded Software for Medical Devices*.
[Mbed OS with Scilab Arduino](#). Accessed: 01 June 2021.
- [9] Xcos STM32 toolbox (2017). *Add STM32 peripherals blocks to the palette, Code Generation*.
[Xcos and Stm32 Build Application Flow](#). Accessed: 01 June 2021.
- [10] Mbed Docs (2020). *Mbed OS 6 Conceptual Architecture*.
[Mbed Architecture](#). Accessed: 30 April 2021.
- [11] AUTOSAR Standards (2018). *AUTOSAR Layered Software Architecture*.
[AUTOSAR Architecture](#). Accessed: 30 May 2021.
- [12] Mbed Docs (2019). *Mbed Library Internals*.
[Mbed Library](#). Accessed: 01 June 2021.
- [13] Jacob Beningo (2020). *Is the war between C and C++ Over?*
[C vs C++ programming for embedded systems](#). Accessed: 01 June 2021.
- [14] Mbed Docs (2020). *Arduino Uno Pin Names*.
[Arduino Uno Pin Names](#). Accessed: 01 June 2021.
- [15] STMicroelectronics Resources (2020). *STM32 Nucleo-F767ZI User's Manual*.
[STM32 Nucleo-F767ZI](#). Accessed: 01 April 2021.

- [16] ST Docs (2017). *STM32F767xx Datasheet*.
[STM32F767xx Datasheet - production data Rev 6](#). Accessed: 01 June 2021.
- [17] RM0410 Reference manual (2018). *STM32 Nucleo-F767ZI User's Manual*.
[STM32F76xxx and STM32F77xxx advanced Arm-based 32-bit MCUs](#). Accessed: 01 June 2021.
- [18] Olimex Resources (2014). *Olimex SHIELD-EKG-EMG User's Manual*.
[Olimex SHIELD-EKG-EMG](#). Accessed: 01 April 2021.
- [19] Gross-Vogt, Katharina & Frank, Matthias & Höldrich, Robert. (2019). *Focused Audification and the optimization of its parameters*.
[Journal on Multimodal User Interfaces](#).
- [20] Faculty of Medicine and Health Sciences. *The McGill Physiology Virtual Lab*.
[Einthoven's triangle](#). Accessed: 01 April 2021.
- [21] ThingSpeak Documentation (2021). *What is IoT?*.
[IoT Infrastructure](#). Accessed: 30 June 2021.