



**POLITECNICO
DI TORINO**

Collegio di Ingegneria Elettronica, delle Telecomunicazioni e Fisica

Corso di Laurea Magistrale in Ingegneria Elettronica

Tesi di Laurea Magistrale

**Develop of ARM Mbed OS support for
BlueNRG-2, a SoC capable to run applications
based on Bluetooth Low Energy protocol.**

Relatore:

Chiar.mo Prof. Maurizio MARTINA

Correlatore:

Chiar.mo Ing. Antonio VILEI

Candidato:

Antonio ORLANDO - 231012

Marzo-Aprile 2019

Contents

Introduction	1
1 Bluetooth Low Energy Architecture	5
1.1 Basic Rate (BR) vs Bluetooth Low Energy (LE)	6
1.1.1 LE Network Topologies	9
1.2 LE Protocol Stack	12
2 Bluetooth Low Energy Stack Design and Organization	17
2.1 PHY - Physical Layer	17
2.2 LL - Link Layer	20
2.3 HCI - Host Controller Interface	22
2.3.1 HCI packet standard	25
2.4 L2CAP - Logical Link Control and Adaption Protocol	28
2.5 SM - Security Manager	29
2.6 ATT - Attribute Protocol	32
2.7 GATT - Generic Attribute Profile	33
2.8 GAP - Generic Access Profile	35
2.8.1 GAP Modes	37
2.8.2 GAP Procedures	37
3 BlueNRG-2 STMicroelectronics System-On-Chip	39
3.1 ARM Cortex-M0 Core Architecture	41
3.2 Peripherals	44

3.2.1	GPIO	44
3.2.2	Wake up Controller and Reset	47
3.2.3	NVIC	47
3.2.4	MFT	49
3.2.5	UART	51
3.2.6	Memory	53
3.2.7	BLE	55
4	ARM Mbed OS 5	57
4.1	HAL Architecture	58
4.1.1	Layer description	60
4.1.2	ARM Cordio BLE Host	64
4.2	Design Tools	64
4.2.1	Mbed Online Compiler	65
4.2.2	Mbed CLI	66
4.2.3	Exporting	66
5	Porting	69
5.1	Setting up (Hardware and Software)	71
5.1.1	Target Description	71
5.2	Hardware API and Peripheral Drivers	74
5.2.1	Startup Routine and Linker Script	74
5.2.2	IRQ and NVIC	78
5.2.3	GPIO	79
5.2.4	Serial	82
5.2.5	Microsecond Ticker	85
5.3	Connectivity	86
5.3.1	BLE API	87
5.4	Low power mode	91
5.4.1	Sleep	92
5.4.2	Deep Sleep	93

5.5	Results and further developments	94
5.5.1	Code size	95
5.5.2	Power performances	96
5.5.3	Final considerations	100
Appendix A - Source Code		103
	Microsecond Ticker - us_ticker_api.c	103
	DTM Command Parsing - Command Table	105
	HCI Driver - BlueNrgHCIDriver.cpp	108
	Low Power Mode - sleep_api.c	115
	ARM Mbed OS HRM (Heart Rate Monitor) - main.cpp	126
Appendix B - Toolchain Setup		131
Bibliography		137

List of Figures

1.1	Bluetooth technology in IoT.	6
1.2	Broadcast Topology [35].	10
1.3	Apple iBeacon advertising packet structure.	10
1.4	Connected Topology [35].	11
1.5	Bluetooth Low Energy hardware configurations [35].	14
1.6	X-NUCLEO-IDB05A1 connected to a NUCLEO-F401RE over SPI through the Morpho Connector.	15
2.1	Bluetooth Low Energy stack organization	18
2.2	BLE Frequency channels [35]	19
2.3	Link Layer state machine [28]	21
2.4	HCI command packet fields [21]	26
2.5	HCI event packet fields [21]	27
2.6	HCI ACL data packet fields [21]	28
2.7	LE Pairing Phases [21]	30
2.8	GATT Data hierarchy [35]	34
3.1	BlueNRG-2 pin out top view (QFN32) [30].	40
3.2	BlueNRG-2 datapath architecture (basic blocks). [30]	41
3.3	Cortex-M0 three-stage pipeline.	42
3.4	Average interrupt current consumption comparison between different architectures. [37]	42
3.5	Cortex-M0 based microcontroller architecture. [37]	43
3.6	BlueNRG-2 wake up logic and reset generation [30].	47

3.7	BlueNRG-2 power-up sequence [30].	48
3.8	BlueNRG-2 MFT mode 3 block diagram [30].	50
3.9	BlueNRG-2 memory address space.	53
4.1	Mbed OS 5 IoT infrastructure (ARM Pelion-based). [10]	58
4.2	Mbed OS 5 architecture	59
4.3	CMSIS Core File Structure. [15]	61
4.4	Mbed Online Compiler.	65
4.5	Mbed CLI project exporter help.	67
5.1	Mbed OS target hierarchical organization [10]	72
5.2	STVAL_IDB008Vx [29]	74
5.3	RESET_HANDLER flowchart (startup part 1).	75
5.4	MBED_SDK_INIT flowchart (startup part 2).	76
5.5	Building process flow.	76
5.6	BlueNRG-2 GPIO driver adaption layer to Mbed OS interrupt HAL.	81
5.7	BLE stack - DTM adaption layer (at HCI)	89
5.8	STMicroelectronics “PowerShield” board and target BlueNRG-2 module expansion power measurement setup	97
5.9	ARM Mbed OS HRM current trend during execution	98
5.10	Sample LED Blink application with <i>sleep mode</i>	99
5.11	Sample LED Blink application with <i>deep sleep mode</i>	100

List of Tables

1.1	Bluetooth Radio Technology [22].	7
1.2	Bluetooth Topology Options [22].	9
2.1	HCI command example on BlueNRG-2 API	23
2.2	HCI event callback example on BlueNRG-2 stack	23
2.3	HCI ACL data transmit command on BlueNRG-2 stack	24
2.4	HCI ACL data receive event on BlueNRG-2 stack	24
2.5	L2CAP layer implementation of BlueNRG-2 ARM Mbed porting (ARM Cordio host)	29
2.6	SM layer implementation of BlueNRG-2 ARM Mbed porting (ARM Cordio host)	32
2.7	ATT - Attribute Example [28]	33
2.8	ATT layer implementation of BlueNRG-2 ARM Mbed porting (ARM Cordio host)	33
2.9	GATT application program interface for BlueNRG-2 ARM Mbed porting	35
2.10	GATT layer implementation of BlueNRG-2 ARM Mbed porting (ARM Cordio host)	35
2.11	GAP modes and role applicability [28]	37
2.12	GAP procedures and mode applicability [28]	37
3.1	BlueNRG-2 IO functional map [30].	45
3.2	BlueNRG-2 GPIO registers [30].	47
3.3	BlueNRG-2 ISR vector table [30].	49
3.4	BlueNRG-2 MFTx registers [30].	51
3.5	BlueNRG-2 UART registers [30].	53
3.6	BlueNRG-2 FLASH - CONFIG register description [30].	55

5.1	Mbed OS HRM compiler report (develop configuration)	96
5.2	Mbed OS HRM compiler report (debug configuration)	96
5.3	HRM code size and SRAM occupation	96

Introduction

Several market analysis about the future of Internet forecast a strong increase in terms of connected devices. Most of daily use objects (domestic appliances, wearable devices, etc.) are becoming “smart” and in turn connected to the network: they will create the so-called *Internet of Things*. Most of these devices will be battery-powered, in need for a low power oriented design.

Looking at the current scenario a potentially valid enabling connectivity technology seems to be *Bluetooth Low Energy*. This work takes inspiration from the following question: “how the market leading companies are planning to drive this new market?”. The aim of this thesis is thus to understand, by looking at the future in which this technology will be to the attention of many developers, how simple, fast and low cost could be a design based on it.

A possible answer to the latter question is: provide rapid prototyping (with potential validity at release time), cross-platform, open source, connected to cloud. Once identified a platform compliant to this research proposal (*ARM Mbed OS*), it has been implemented on a product providing connectivity in compliance with the mentioned “low cost” requirement (*STMicroelectronics BlueNRG-2*).

This study however, because of its experimental purpose, is not intended to contextualize the discussed technology and forecast the trend in the smart objects market (only assumption could be made), due to the fact that it is still very uncertain. It is intended to be a design of one valid proposal between possible solution, but further future studies are required to determine the true market location of such a solution. What is hoped to last long and to draw inspiration for other studies are the idea and the methodology regarding this work.

Thesis Organization

The document is organized as in the following.

Chapter 1 provides the state of art of Bluetooth Low Energy: first of all there is a general overview, an analysis to contextualize its current market and which features are required to BLE-based products; then, there is an explanation on what are the key features of the BLE made up from a comparison between Bluetooth Low Energy and Basic Rate/Enhanced Data Rate technologies. Afterwards the most widespread architectures on which BLE products are developed, enhanced by examples of the latter used to develop the thesis work are presented.

Chapter 2 presents the stacked architecture of BLE, with a specific section dedicated to each layer. Every section explains general concepts of these building blocks; however this is not intended to be as a complete stack characterization, so it does not cover any aspect of each building block, but rather an experience-based description of how these concepts have been applied during this thesis period on the specific case of Mbed OS porting on BlueNRG-2. This kind of exposition have been preferred because it allows to highlight the key point of the performed experimentation.

Chapter 3 contains an architectural description of BlueNRG-2 System-On-Chip, in particular its core logic and instruction set, peripherals and interconnection circuitry. It shows the whole addressing mechanism for peripheral and core register and memory system (Flash and SRAM). At the end, how to optimize the BLE controller initialization to reduce data memory occupation is explained.

Chapter 4 describes how Mbed OS simplifies IoT design based on ARM Cortex-M architecture devices. It sets out its *Hardware Abstraction Layer* architecture, which avoids a complex (and not portable) *bare metal* programming providing a full C/C++ instruction set, and how the ecosystem deals with the code size increase introduced as drawback, in terms of a modular approach. After that, the most significant modules are shown, together with the developing technologies and tools enabling the design on the Mbed OS platform.

Chapter 5 shows the actual Mbed OS porting on BlueNRG-2 SoC, ideas and solutions to the encountered issues. The provided contents structure however presents also the whole project workflow, indeed it is organized according the project “timeline”. Starting from bare metal requirements, passing through HAL API and BLE API porting and then in a low power features study, it faithfully runs through the time (nicely) spent for developing this thesis in STMicroelectronics Lecce site. Finally, what learned by this experience and the future scenario deriving from this activity is presented.

Chapter 1

Bluetooth Low Energy Architecture

In 2018, nearly 4 billion devices has been shipped with Bluetooth technology. Thanks to Bluetooth mesh networking and the arrival of Bluetooth 5, released by the *Bluetooth Special Interest Group (SIG)* on 7th December 2016, Bluetooth is now poised as an industrial-grade connectivity solution and this suggests Bluetooth is about to become the wireless constant in the Internet of Things (IoT) for decades to come.

Since its inception (almost 20 years ago), Bluetooth has continuously evolved, expanding the universe of innovative ways for things to connect — driving innovation creating new categories of devices. Whether it is a connection for wireless audio, wearable devices, tracking assets, or automating buildings, Bluetooth is the innovative force creating new consumer, commercial, and industrial markets. [22]

Bluetooth 5 Core Specification defines two different configurations:

- Basic Rate (*BR*, the “classical” technology);
- Low Energy (*BLE* or *LE*, introduced in 2010, with Core 4.0 specification).

Bluetooth 5 is fully back compatible with previous versions of the Core, ensuring the correct interoperability among devices implementing different specification versions. *LE* is back compatible down to Core 4.0 (its first release), hence Bluetooth devices qualified on any specification version prior to 4.0 cannot communicate in any way with a *BLE* device. Bluetooth wireless technology shares some similar features between *BR* and *LE* stack protocols, such as

device *discovering*, *advertising*, *connection establishment* procedure, nevertheless these technologies are intended for different scenarios and they are not mutually compatible. [35].

Unless otherwise noted, this document uses the Bluetooth Core 5 Specification as reference [21].

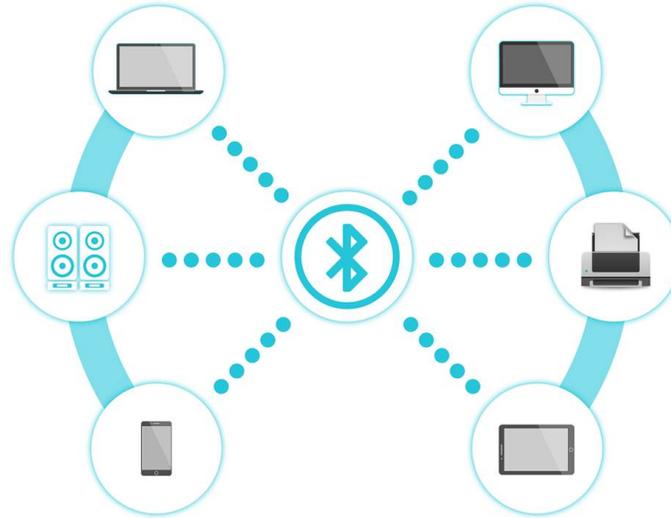


Figure 1.1: Bluetooth technology in IoT.

1.1 Basic Rate (BR) vs Bluetooth Low Energy (LE)

Bluetooth operates at frequencies between 2400 MHz and 2483.5 MHz, including band guards 2 MHz wide at bottom end and 3.5 MHz wide at top. This is in the globally unlicensed (but not unregulated) Industrial, Scientific and Medical (ISM) 2.4 GHz short-range radio frequency band. Table 1.1 sums up the most important characteristics of the *Physical Layer (PHY)* of both Bluetooth technologies [23].

	Bluetooth LE	Bluetooth BR/EDR
<i>Optimized For...</i>	Short burst data transmission	Continuous data streaming
<i>Frequency Band</i>	2.4GHz ISM Band (2.402 – 2.480 GHz Utilized)	2.4GHz ISM Band (2.402 – 2.480 GHz Utilized)

<i>Channels</i>	40 channels with 2 MHz spacing (3 advertising channels/ 37 data channels)	79 channels with 1 MHz spacing
<i>Channel Usage</i>	Frequency Hopping Spread Spectrum (FHSS)	Frequency Hopping Spread Spectrum (FHSS)
<i>Modulation</i>	GFSK	GFSK, $\pi/4$ DQPSK, 8DPSK
<i>Power Consumption</i>	$\sim 0.01x$ to $0.5x$ of reference (depending on use case)	1 (reference value)
<i>Data Rate</i>	LE 2M PHY: 2 Mb/s LE 1M PHY: 1 Mb/s LE Coded PHY (S=2): 500 Kb/s LE Coded PHY (S=8): 125 Kb/s	EDR PHY (8DPSK): 3 Mb/s EDR PHY ($\pi/4$ DQPSK): 2 Mb/s BR PHY (GFSK): 1 Mb/s
<i>Max Tx Power</i>	Class 1: 100 mW (+20 dBm) Class 1.5: 10 mW (+10 dBm) Class 2: 2.5 mW (+4 dBm) Class 3: 1 mW (0 dBm)	Class 1: 100 mW (+20 dBm) Class 2: 2.5 mW (+4 dBm) Class 3: 1 mW (0 dBm)
<i>Network Topologies</i>	Point-to-Point (incl. piconet) Broadcast Mesh	Point-to-Point (incl. piconet)

Table 1.1: Bluetooth Radio Technology [22].

BR system is essentially thought to low-power continuous data transfers. It uses a radio technology called Frequency-Hopping Spread Spectrum (FHSS), transmitting data into packets, each packet on one of 79 designated Bluetooth channels. Every channel has a bandwidth of 1 MHz. It usually performs 1600 hops per second, with Adaptive Frequency-Hopping (AFH) enabled. BR includes also *Enhanced Data Rate (EDR)*, *Alternate Media Access Control (MAC)* and *Physical (PHY) layer extension (AMP)*; this system offers synchronous and asynchronous connections with data rates of 721.2 kb/s for BR, 2.1 Mb/s for EDR and up to 54 Mb/s with the 802.11 (i.e. WiFi™) AMP - also called *Bluetooth HS (High Speed)* - by using Bluetooth to

establish the connection and WiFi to transport the actual data.

BLE uses the same FHSS technique, but with several differences in the channel distribution (detailed in Section 2.1). It is oriented to very low power applications, in particular to a market whose wireless-connected devices are designed to be powered by a *coin-cell battery*, such as the transmitting very small packets of data - 8-27 octets - at low rates with longer transmission intervals (up to 10 seconds, consuming a small fraction of power with respect to BR, EDR, HS devices) [28]. In addition to that BLE is also capable to setup a connection and start a transmission in less than 10 milliseconds (with respect to BR, which takes up to 1 second). These features allows BLE to work in conditions where the Bluetooth radio is switched off for long time windows, thus achieving the discussed low power performances and making this technology perfect both in point-to-point and broadcast connections and also in the *Personal Area Network (PAN)* context, especially with *Bluetooth Mesh* topology, as clarified by Table 1.2 [24].

	Bluetooth LE	Bluetooth BR/EDR
<i>Point-to-Point (1:1 device communication)</i>		
<i>Setup time</i>	< 6 ms	100 ms
<i>Max connections/device (piconet)</i>	Unlimited (implementation specific)	7
<i>Max payload size</i>	251 byte	1021 byte
<i>Security</i>	128 bit AES, user defined application layer	64 bit/ 128 bit, user defined application layer
<i>Service definition</i>	GATT Profiles	Traditional Profiles
<i>Broadcast (1:m device communication)</i>		
<i>Max payload size</i>	Primary Channel: 31 byte Secondary Channel: 255 byte Chaining of packets for larger messages	Not Applicable
<i>Security</i>	User defined application layer	

<i>Service definition</i>	Beacon Formats (not specified by Bluetooth SIG)	
<i>Mesh (m:m device communication)</i>		
<i>Max nodes</i>	32767	Not Applicable
<i>Max subnets</i>	7	
<i>Message addressing</i>	Unicast, Multicast, Broadcast Up to 16,384 group addresses Supports publish/subscribe addressing	
<i>Message forwarding</i>	Managed flood	
<i>Max payload size</i>	29 byte payload	
<i>Security</i>	128-bit AES Device, network and application levels	
<i>Service definition</i>	Mesh Models, Mesh Properties	

Table 1.2: Bluetooth Topology Options [22].

1.1.1 LE Network Topologies

A BLE device can communicate with the rest of the world in two ways: *broadcasting* and *connection*. Each mechanism has its own advantages and limitations, and they are both subject to the guidelines established by the *Generic Access Profile (GAP)*, which defines device *roles* in the communication [35] and is described in detail in Section 2.8.

Broadcasting And Observing

In BLE connectionless *broadcasting* mode is possible to send data out to any scanning device or receiver in listening range. As shown in Figure 1.2 this mechanism essentially allows to send data out *one-way* to any actor that is capable of picking up the transmitted data.

In this topology one can identify two kind of devices: *broadcaster* (or *beacon*) and *observer*.

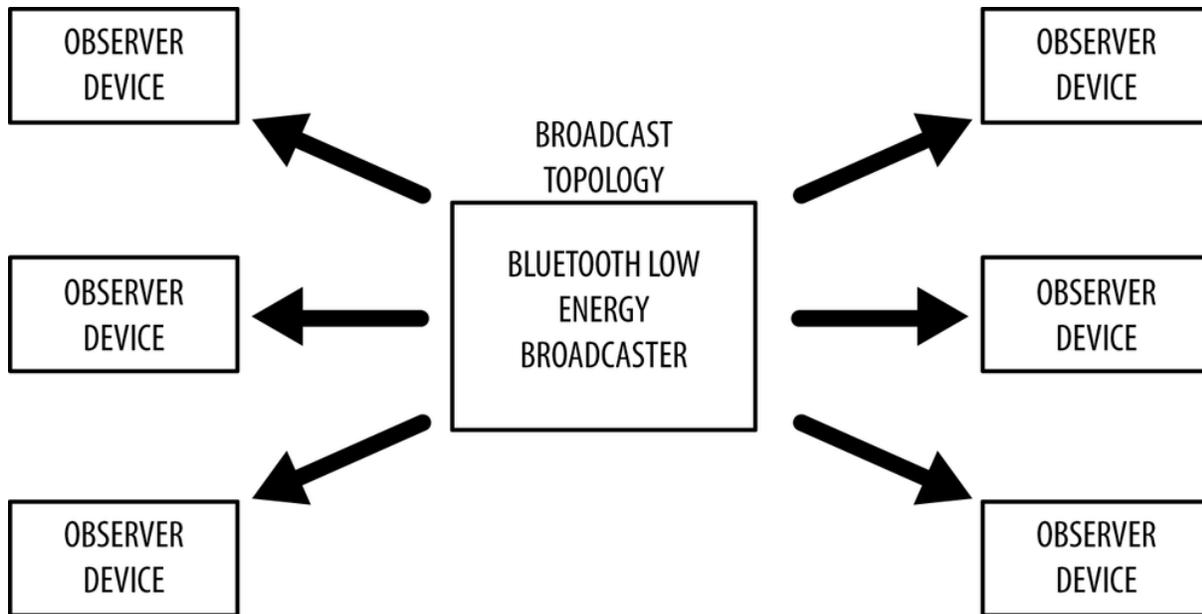


Figure 1.2: Broadcast Topology [35].

Broadcasting is the perfect choice for those scenario where the push of small amounts of data on a fixed scheduling time is required. A practical application (the “hello world” of the BLE technology) of this connection topology is the Apple® *iBeacon* indoor positioning technology [1], that consists in a 31 bytes [17] advertising packet (the organization such packets is shown in Figure 1.3) periodically dispatched with any security and privacy (this is the major issue of the broadcast architecture).

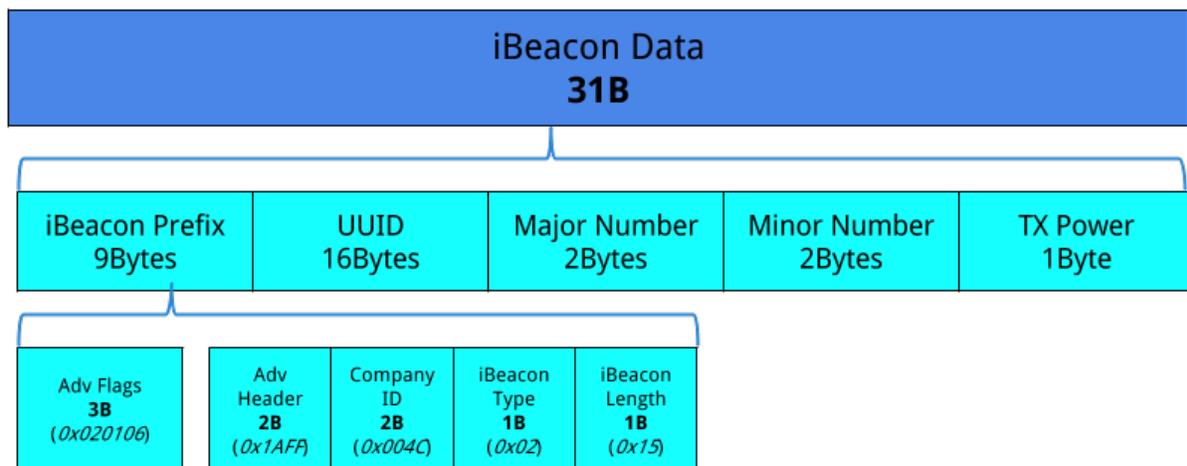


Figure 1.3: Apple iBeacon advertising packet structure.

Connection

Connected topology presents more advanced features in comparison to broadcast one. In particular it presents three main features:

- bidirectional communication;
- capability of sending higher amount of data with respect to the *advertising* payload;
- privacy, the communication takes place only between the two peers direct involved (no unadmitted sniffing even in the communication range).

Connection includes devices with two kind of roles:

- *central* or *master*, it initiates and manages the connection, the connection timing of the link;
- *peripheral* or *slave* accepts the connection request.

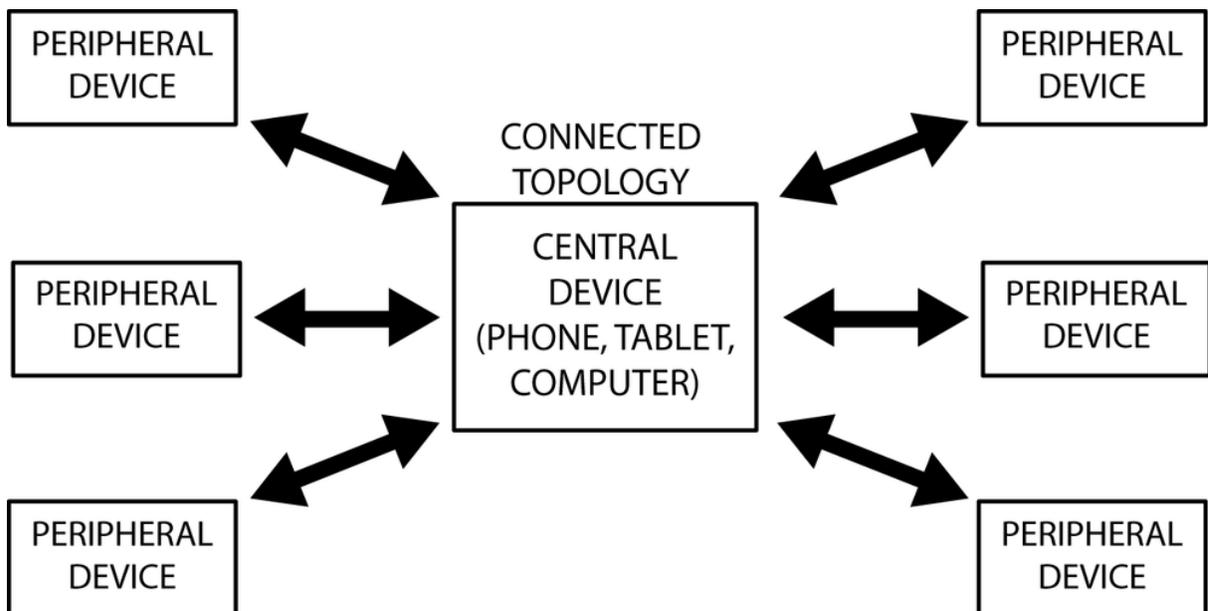


Figure 1.4: Connected Topology [35].

A device can be simultaneously central and peripheral if its link layer handles multiples connections (as in BlueNRG-2 SoC - Section 2.2), this allows to build up hybrid networks, with devices acting different roles.

Furthermore it brings with it an important advantage: lower power consumption [35]. As a matter of fact in this mode, with respect to broadcast, it is possible to extend the delay of connection events further out, to accumulate data and send them in larger chunks. This allows to power off the radio for longer period, and wake up it when the planned send event is reached, rather than continuously advertise the full payload at a fixed advertising rate.

1.2 LE Protocol Stack

Since the inception of the Bluetooth Low energy technology, formally adopted by the Bluetooth SIG starting from the Core specification version 4.0, it has been organized in a series of basic communicating hardware and software “building blocks” organized in a *stack* structure, the so called *Bluetooth Low Energy stack*.

More in detail, the BLE stack can split up in three macro groups, each of them including certain of the BLE stack layers [35], namely:

- **Controller**

- PHY (2.1)
- LL (2.2)
- HCI (controller side) (2.3)

The bottom part of the Bluetooth Low Energy protocol stack, including the radio. The reference protocols and design constraints for those layers are defined in the Bluetooth SIG core specification [21].

- **Host**

- HCI (host side) (2.3)
- L2CAP (2.4)
- SM (2.5)
- ATT (2.6)

- GATT (2.7)
- GAP (2.8)

The top part of the Bluetooth Low Energy protocol stack. It manages the communication between devices. The reference protocols and design constraints for those layers are defined in the Bluetooth SIG core specification [21].

- **Application** User application interface with the BLE stack. It implements, using the latter, the complex functionality required by the use case for which the BLE-based system is developed (i.e. Apple iBeacon). Furthermore, the Bluetooth SIG core specification document [21] defines how low energy devices acts in certain particular applications (like ones regarding health care, fitness, proximity sensing, mesh, etc.) and offers several out of the box solutions to implement them, the so called BLE *profiles* (or *services*) (as, for instance, the HeartRate Profile - HRP, Glucose Profile - GLP, Proximity Profile - PXP, etc), in a similar way as BR/EDR does (e.g. with File Transfer Profile - FTP, Headset Profile - HSP, etc.).

This stack organization is fundamental to reach the inter-operability between different devices, manufactured by different companies and implementing the same BLE stack in different ways.

In particular these layers can be implemented on a single SoC BLE device, for example this configuration is used by simple sensors, to keep Printed Circuit Board (PCB) design and footprint complexity, bill of materials and then cost low; otherwise, where the application complexity grows in runs on a different application processor, connected to the controller through by means of a ***transport layer***. These are (depicted in Figure 1.5) the most common configuration commercially available.

Dual IC with connectivity device

One IC, typically a microcontroller but sometimes an application processor (e.g. on smartphone implementations), runs the application, while a second IC runs the complete BLE stack (host + controller). Usually the transport layer between those two ICs is implemented over a

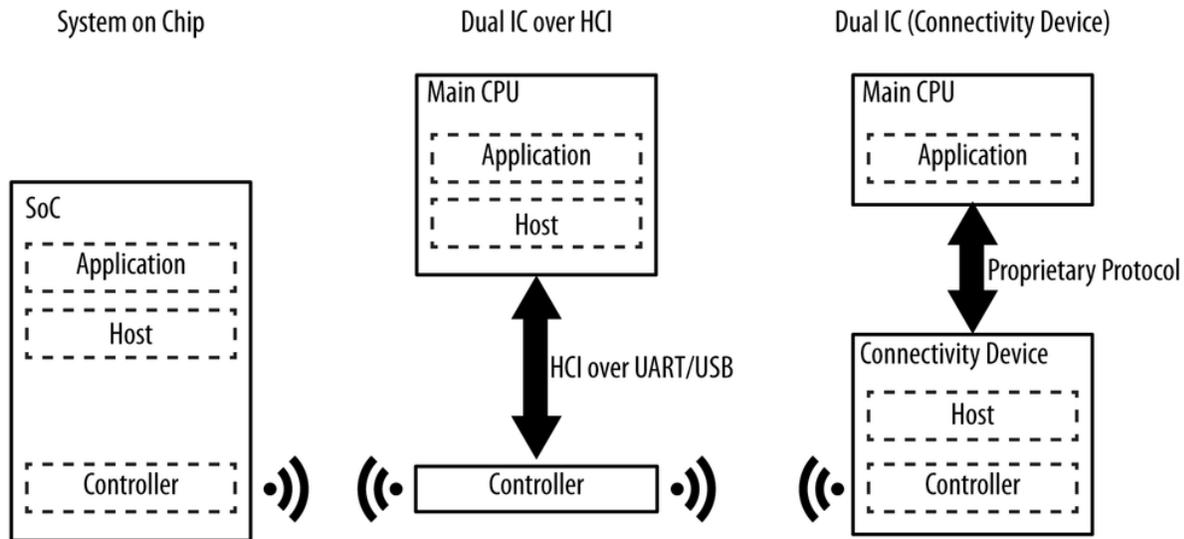


Figure 1.5: Bluetooth Low Energy hardware configurations [35].

proprietary protocol, chosen by the vendor and not included in the Bluetooth Core Specification, so the application needs to be adapted. Whether this solution is good or not depends on the final application.

Dual IC over HCI

One IC runs the application and the host, while the second runs the controller. The advantage of this configuration lies in the standardized (by the SIG) HCI interface, so it's easy to choose the two ICs, without regards for the manufacturer. The transport layer between host and controller can be implemented by using the most common standard for integrated circuits communication, such as USB, UART or SPI interface.

An example of this configuration has been used during the thesis work as initial study case for the BLE technology and moreover for advanced debugging purposes. It consists in the couple of ICs (Figure 1.6):

- STM32F401RE microcontroller over NUCLEO-F401RE board (it embeds also an on-circuit debugger and programmer ST-LINK v2.1), running an MbedOS 5.11 based application and the host ARM Cordio Stack;
- BlueNRG-MS over a X-NUCLEO-IDB05A1 (it embeds a NUCLEO-compatible morpho

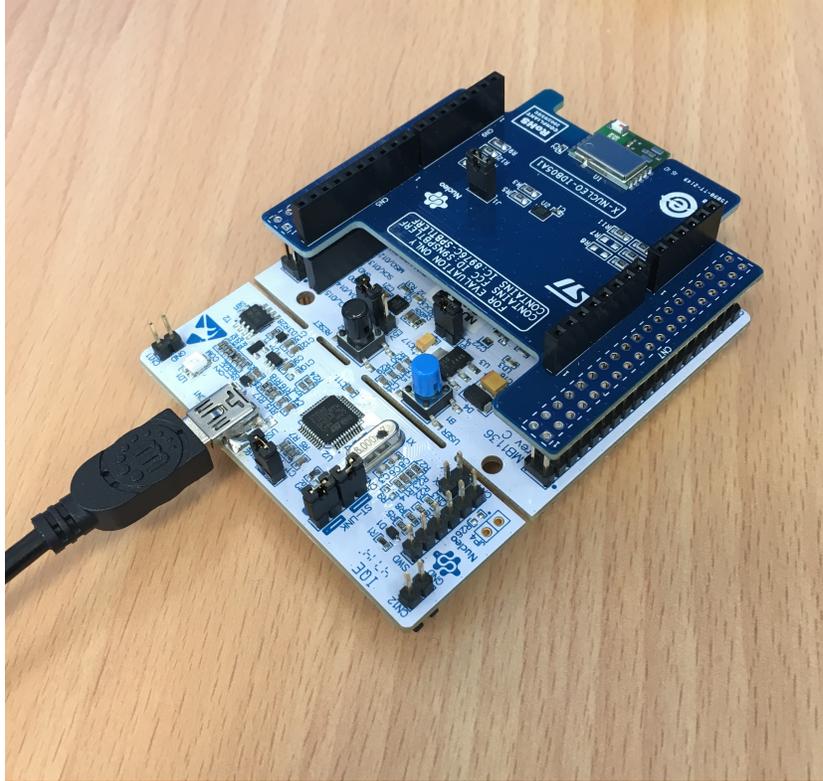


Figure 1.6: X-NUCLEO-IDB05A1 connected to a NUCLEO-F401RE over SPI through the Morpho Connector.

connector and the RF frontend to the ceramic antenna), a low-power single-mode BLE network co-processor¹.

SoC (System On Chip)

A single IC runs the application, the host and the controller. It is the case of STMicroelectronics BlueNRG-2 SoC (further information are provided by Chapter 3), the focus product of this thesis work, running an ARM MbedOS 5.11-based application, the ARM Cordio host and the STMicroelectronics controller stack on the same IC.

Chapter 2 provides more in detail a description of each building block of the BLE stack,

¹For the sake of completeness, the above configuration fall into this category because of its configuration. Indeed, the BlueNRG-MS is capable to run the full BLE stack (it is an ARM Cortex M0-based product) and to be programmed and controlled by SPI or UART also at the host level, resulting suitable also for the first category (although it has never been used in this way during the thesis work).

with a focus on the latter configuration (SoC) in the specific case of the Mbed OS porting on BlueNRG-2.

Chapter 2

Bluetooth Low Energy Stack Design and Organization

In the following sections there is a deepening on each layer of the BLE stack, mainly describing standardized features in the Bluetooth Core Specification document. Practical examples are also added, referring to the particular System On Chip stack implementation of ARM Mbed OS on BlueNRG-2 (regarding the host) and to its specific hardware accelerators (controller side).

A summary overview representing the BLE stack structure and layers relationships is given by Figure 2.1: this BLE stack description is exposed by a bottom-up approach, starting from the Radio Frequency level up to the application interface one.

2.1 PHY - Physical Layer

The intent of the *Physical Layer* is essentially to perform the translation between the micro-controller digital domain to the radio analog one of the radio. It consists of a circuitry capable of modulating and demodulating and transforming them into a sequence of digital symbols, i.e. a sequence of bits.

As mentioned while comparing BLE and BR/EDR technologies in Section 1.1, BLE uses the same FHSS technique of BR/EDR, but with 2 MHz spacing between each channel, which accommodates 40 channels, 3 of them (37, 38, 39) dedicated to *advertising* [21], as shown in

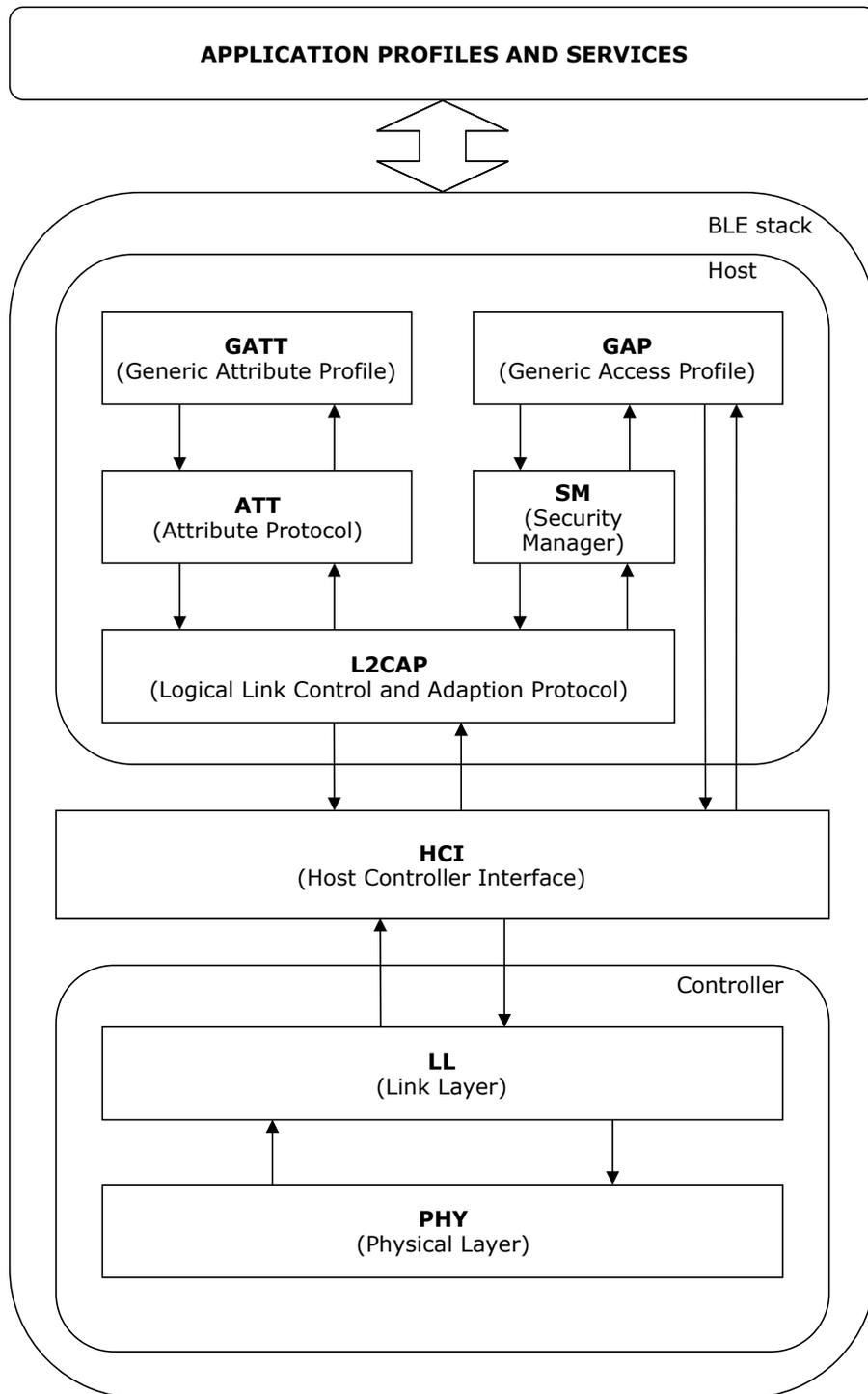


Figure 2.1: Bluetooth Low Energy stack organization

Figure 2.2. This technique minimizes the effect of any interference potentially present in the 2.4 GHz band, especially from radio (classic Bluetooth, IEEE 802.11 - WiFi, IEEE 802.15 - WPAN,

i.e. Wireless Personal Area Network), but also from high power analog devices (microwave ovens), whose strong transmission power can affect the activity of low energy devices [19].

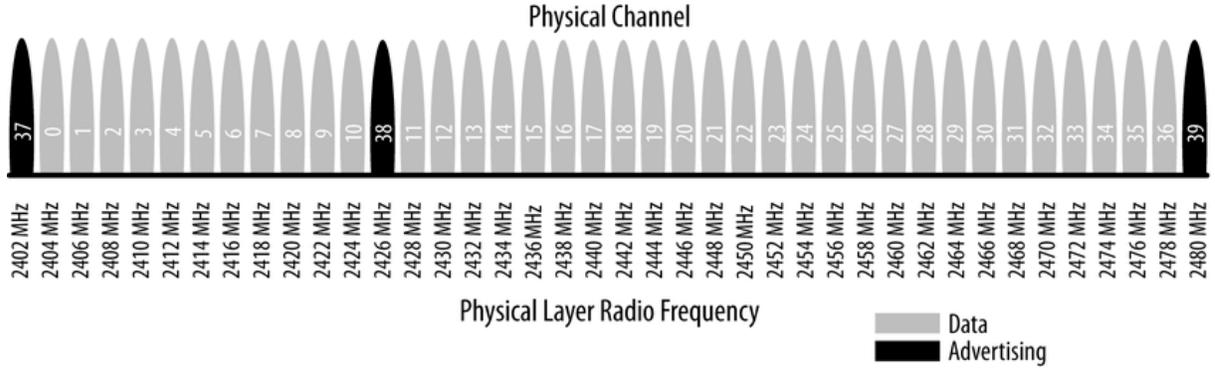


Figure 2.2: BLE Frequency channels [35]

More in details BLE uses an *adaptive frequency hopping (AFH)* technology, exploiting only a subset of all the available frequencies in order to avoid transmissions on those ones used by other devices with no-adaptive technologies. Those frequencies are centered at [28]:

$$f_c = 240 + k * 2 \text{ MHz, where } k = 0.39.$$

The radio *hops* between channels is communicated on each connection event using the formula [35]:

$$ch_{new} = (ch_{old} + hop) \text{ mod } 37$$

where *hop* is a randomly generated number.

Random Number Generator (RNG)

On *BlueNRG-2* those numbers are generated by a Random Number Generator hardware peripheral. The latter is based on a continuous analog noise that provides a 16-bit value when the read acts and its throughput is 1 number every $1.25\mu s$. Even if this unit is for the stack working purposes (it is used also for some Link Layer features), its generated value can also be read from *user context*, since RNG is addressed through *AHB (AMBA¹ High-Performance Bus)*.

¹Advanced Microcontroller Bus Architecture

This implies, on Cortex-M0, that the peripheral access shall be in 32 bit (refer to Section 3.2.6), otherwise an hard fault is raised [30].

Additional details about PHY radio are provided in Section 3.2.7.

2.2 LL - Link Layer

The *Link Layer* lies on the PHY interface (as illustrated in Figure 2.1) and it is usually implemented with a custom hybrid combination of hardware and software (depending on the manufacturer) [35]. Considering stack design, link layer is the most complex part, its implementation is abstracted to the upper layers of the host by the *HCI* (details in Section 2.3), whose purpose is in fact to standardize the access to LL. Its complexity lies in the fact that this layer works with hard real-time constraints, indeed it implements the architecture for timing management according to the requirements of the Bluetooth Low Energy Core Specifications.

As mentioned above it is an hybrid HW/SW design: starting from the hardware part, it includes hardware accelerators for intrinsically automated functionality and computationally expensive features, to avoid the overloading of *control unit*, that has to process all the host (and also the application in a SoC implementation like BlueNRG-2) with additional complex tasks. On BlueNRG-2 these LL hardware accelerators are:

- RNG (described in Section 2.1);
- Public Key Accelerator (PKA) [30].

Public Key Accelerator (PKA)

On BlueNRG-2 the Public Key Accelerator unit is used for those application requiring security over the link. It is involved in the computation of cryptographic public keys primitives through *Elliptic Curve Cryptography (ECC)*, by using a predefined curve and a predefined prime modulus.

This peripheral is addressable and accessible through AHB. PKA core is clocked at $f_{ck}/2$, while memory is clocked at f_{ck} ; after reset the PKA core and memory are clock-gated, so before its utilization it requires the correct clock gate initialization.

The main feature of the PKA unit are:

- Elliptic curve Diffie-Hellman (ECDH) public-private key pair calculation accelerator;
- fast modular multiplication based on the Montgomery modular algorithm;
- AHB slave interface with reduced command set;
- PKA internal RAM available for the system when the hardware accelerator is not used.

Software LL

The software part of the link layer manages the link state of the radio and establishes the role of devices in the communication (advertiser-broadcaster/observer-scanner or central-master/peripheral-slave).

On BlueNRG-2 the LL software is implemented through a Finite State Machine with 5 states, shown in Figure 2.3.

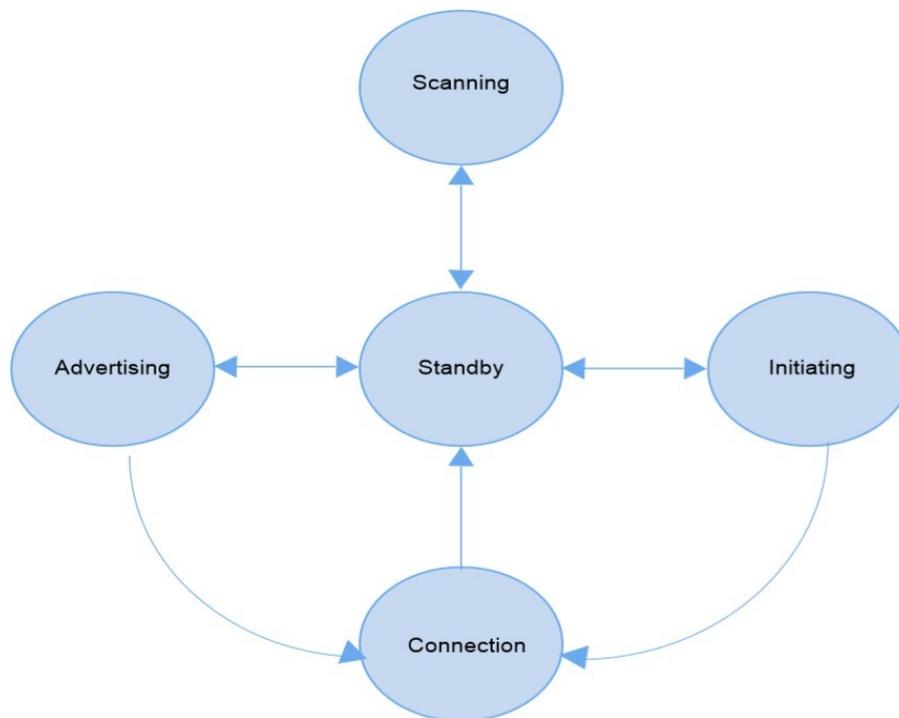


Figure 2.3: Link Layer state machine [28]

BlueNRG-2 supports up to 8 simultaneous link, i.e. is capable of concurrently processing up to 8 of these finite state machines. By the way, using even more links increases both flash

and RAM occupation, so there is a tradeoff with the reachable application complexity that decreases along with the increasing of the links number. It would be useful to explain more in detail certain BlueNRG-2 link layer features by presenting some key pieces of its code. However this is not feasible, since its source code is STMicroelectronics-classified and not released. Moreover, link layer has non of its *API (Application Program Interface)* exposed. To access the controller functionality STMicroelectronics releases a static library available in the “*BlueNRG DK (Development Kit) - version 3.0.0*” [31], named “*libbluenrg1_stack.a*”, even if, the “lowest” layer on which a developer can find exposed API is the *Host Controller Interface*.

2.3 HCI - Host Controller Interface

As already discussed, Bluetooth Low Energy devices allows several different configurations in their implementation, they are based on the chip count and the application complexity. In its Core Specification document, Bluetooth SIG defines HCI layer in terms of standard protocol that permits to the host, across a serial interface, to communicate with the controller, and vice versa. Depending on the implementation - dual chip over HCI or single chip - several additional layers could be added in the construction of the BLE stack.

In general, one can identify three different typologies of HCI communication:

- host sends to controller *HCI commands*;
- controller notifies to the host *HCI events*;
- host and controller bidirectionally exchanges (TX and RX) *HCI ACL data (Asynchronous ConnectionLess)*.

In dual chip solutions host-controller communication is usually implemented across the most widespread serial peripherals, i.e. UART, SPI, SDIO or USB; in this case the overhead introduced in the BLE stack is given by peripheral drivers (both HCI host-side and controller-side). This is the so called *HCI Transport Layer* [35].

In single-chip design, this implementation depends on the SoC hardware architecture. For instance, on BlueNRG-2, HCI command layer (host side) is accessible through ST stack functions, whose name starts by “*hci_**”. They are designed in compliance to Core specification

documents [21], Volume 2 part E. A command prototype is reported as example in Table 2.1, full HCI API is available in “*bluenrg1_api.h*” [31].

<i>tBleStatus</i> ² <i>hci_le_set_advertising_data</i>	
uint8_t Advertising_data_length	The number of significant octets in the following data field (input parameter)
uint8_t* Advertising_data[31]	31 octets of data formatted as defined in Vol. 3 Part C, Section 11 [21] (input parameter)

Table 2.1: HCI command example on BlueNRG-2 API

Concerning HCI events on BlueNRG-2, its notification to the host is performed by a series of *event callbacks*, one for each event specified by the Core document [21]. The full HCI event API is declared in the file “*bluenrg1_events.h*”; an example of BlueNRG-2 HCI event callback is provided in Table 2.2.

<i>void hci_le_read_remote_used_features_complete_event</i>	
uint8_t Status	Standard error code from Bluetooth specification [21], Vol. 2, part D (input parameter)
uint16_t Connection_Handle	Connection handle to be used to identify the connection with the peer device (input parameter)
uint16_t* LE_Features[8]	Bit Mask List of used LE features, according to LE Link Layer specification [21] (input parameter)

Table 2.2: HCI event callback example on BlueNRG-2 stack

HCI ACL data transmission in BlueNRG-2 is performed by the command shown in Table 2.3:

²Unsigned byte indicating success or error code.

<i>tBleStatus hci_tx_acl_data</i>	
uint16_t connHandle	Connection handle for which the command is given. Range: 0x0000-0x0EFF (0x0F00 - 0x0FFF Reserved for future use) (input parameter)
uint8_t pbFlag	Packet boundary flag (input parameter)
uint8_t bcFlag	Broadcast flag (input parameter)
uint16_t dataLen	Length of PDU data in octets (input parameter)
uint8_t* pduData	PDU (Protocol Data Unit) data pointer (input parameter)

Table 2.3: HCI ACL data transmit command on BlueNRG-2 stack

while the ACL data receive event callback is shown in Table 2.4. Notice that this event is the only one in the entire BlueNRG-2 HCI layer returning a *tBleStatus* byte. This allows to notify the link layer FSMs any possible errors in ACL data delivering (for instance disable the controller ACL indication, or set a timeout, to avoid incoming buffer saturation).

<i>tBleStatus hci_rx_acl_data_event</i>	
uint16_t connHandle	Connection handle for which the command is given. Range: 0x0000-0x0EFF (0x0F00 - 0x0FFF Reserved for future use) (input parameter)
uint8_t pbFlag	Packet boundary flag (input parameter)
uint8_t bcFlag	Broadcast flag (input parameter)
uint16_t dataLen	Length of PDU data in octets (input parameter)
uint8_t* pduData	PDU (Protocol Data Unit) data pointer (input parameter)

Table 2.4: HCI ACL data receive event on BlueNRG-2 stack

In addition to that, HCI ACL data receive event handler is available only if BlueNRG-2 is reset into a special mode: the so-called *link layer only* mode. This mode has resulted fundamental in the Mbed driver development (further details are discussed in Section 3.2.7).

BlueNRG-2 HCI API is exposed, contrary to link layer; the implementation however, as

well as link layer case (Section 2.2), is STMicroelectronics classified and cannot be shown in this document.

The single chip BlueNRG-2 case of study offers a point of view on how implement use HCI layer in those kind of devices. Even if BlueNRG-2 HCI design is totally compliant to Bluetooth SIG Core Specification, it is not so clarifying to understand how communication at this point (commands, events, ACL data) takes place, as it is impossible to identify a transport layer between host and controller.

Nevertheless find the connection between the transport-based design of the HCI interface (as in ARM Cordio stack), described by Bluetooth SIG in the form of standardized packets, and the function-callback structure previously discussed, is a key point of the design the Cordio-ST driver developed during this thesis work and shown in Chapter 5.

2.3.1 HCI packet standard

Whether they are commands, events or ACL data, HCI format present a set of common rules [21]:

- fields of packets shall be intended in *Little Endian* form, unless otherwise specified;
- negative values are expressed as two's complement, where admitted;
- the order of parameters in HCI command function, as well as in HCI event callbacks, is the same as in the HCI packet;
- all parameter values are expressed and received in Little Endian format, unless otherwise specified;
- parameter values or opcodes not defined by an implementation shall be ignored and the operation shall be completed (i.e. host or controller shall not stop functioning because of receiving an incorrect value).

HCI Command Packet

It is sent to the controller from the host. A BLE controller shall support commands with up to 258 byte: 3 byte header (opcode + length) + 255 byte payload; packet structure is shown in

Figure 2.4.

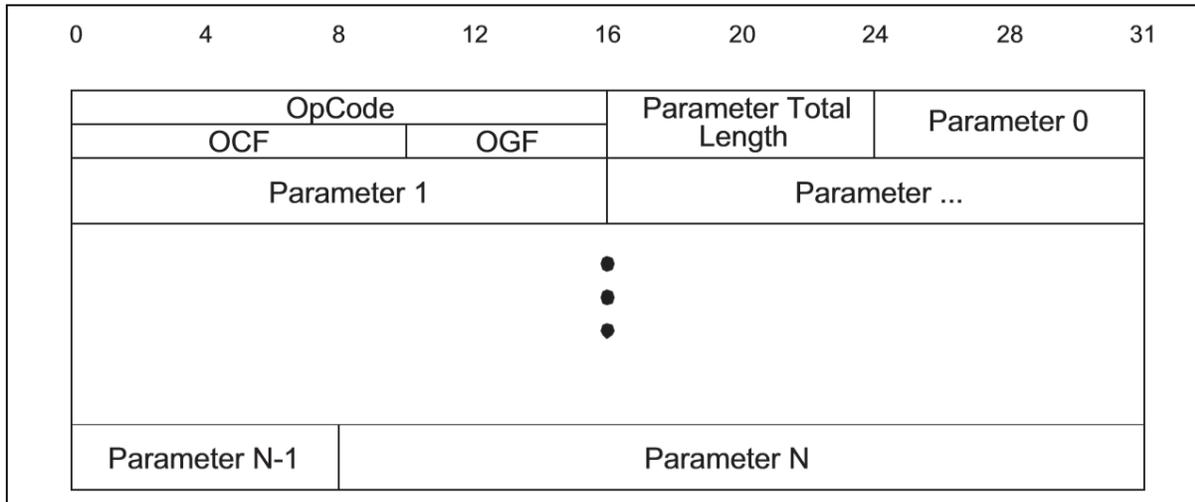


Figure 2.4: HCI command packet fields [21]

Each packet is composed by 2 bytes opcode for univocal command identification. Opcode parameter is divided into 2 fields:

- *OGF* Opcode Group Field, the 6 MSB;
- *OCF* Opcode Command Field, the 10 LSB.

OGF with all bits equal to 1 (OGF = 0x3F) represents *vendor specific commands*. An example of a vendor specific command is given by *aci_hal_write_config_data* discussed in Section 3.2.7.

Each command has a certain number of parameters with the following structure: the first octet represents the length of the parameter in octets, the following ones are the parameter values.

HCI Event Packet

It is sent to the host from the controller to signal an occurred event. Host shall support packet events with up to 255 byte size plus the HCI event header (2 byte); its structure is shown in figure 2.5.

Fields of the HCI event packet are described in the following:

- *Event code* (1 octet) is used to uniquely identify different types of events;

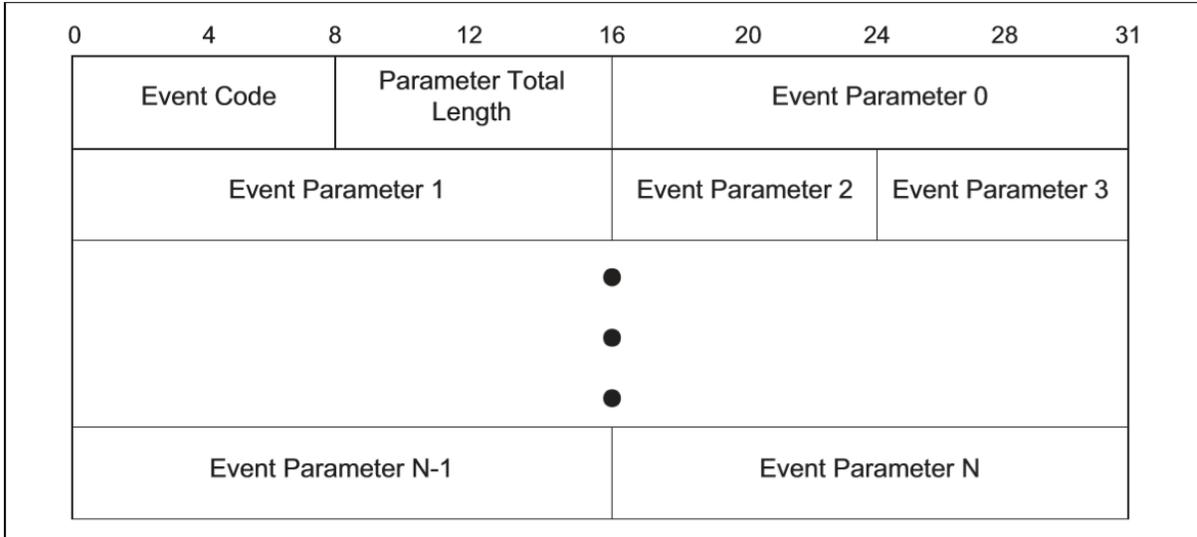


Figure 2.5: HCI event packet fields [21]

- *Parameter Total Length* (1 octet) is the length of all the parameters in the event packet, specified in number of octets;
- *Event Parameter* is the payload (the size and number of parameters for each events is specified).

Low Energy specific events are identified by an additional byte - a 0x3E placed always at the beginning - in the event packet.

HCI ACL data Packet

HCI ACL data packets are bidirectionally exchanged between host and controller and can be divided in two groups [21]: *Automatically-Flushable*, whose flush is based on the setting of an automatic flush timer; *Non-Automatically-Flushable*, not controlled by the timeout mechanism described above, that shall be kept alive and handled in a different way. The format of the HCI ACL Data Packet is shown in Figure 2.6

An ACL data packet has size up to 31 octets (including a *L2CAP* header inside the Data payload, 2.4); it is structured in the following way:

- *Handle* (12 bits) represents the connection handle which identifies the primary controller transmitting a packet (or a segment);

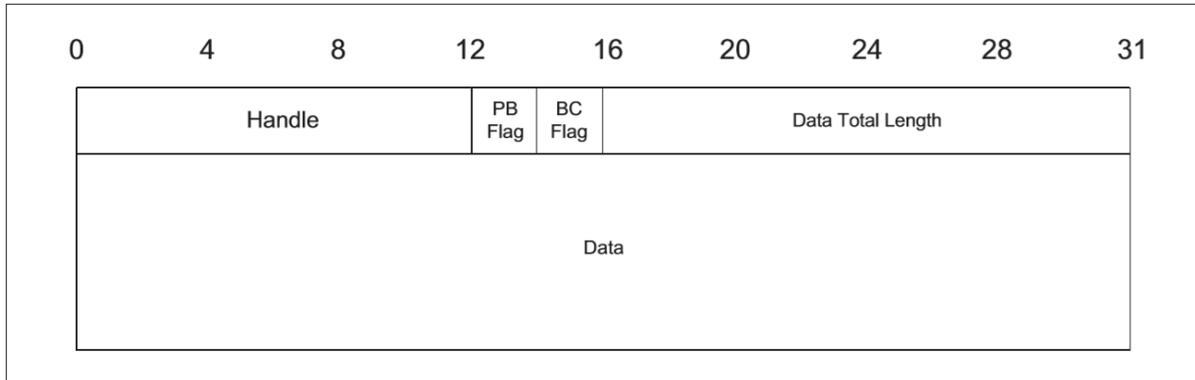


Figure 2.6: HCI ACL data packet fields [21]

- *PB Flag* (2 bits) is the Packet Boundary Flag, it identifies if when a packet have been fragmented (if the controller does not support packet Length Extension);
- *BC Flag* (2 bits) discriminates between point-to-point and broadcast data.
- *Data Total Length* (2 octets) is the length of the *Data* payload.

2.4 L2CAP - Logical Link Control and Adaption Protocol

HCI is the layer in charge of performing packets exchange between host and controller. However it acts only at transport level, it means that HCI does not take into account anything about physical buffers' size (controller side) or memory organization (host side). L2CAP is in charge to handle the data organization inside packets.

First of all, in case of transmission, L2CAP performs fragmentation of packets to fit the controller size: it takes large data chunks coming from the upper layers and shrink them up to fit the 27 bytes³ payload of BLE Link Layer packets. In reception L2CAP parses multiple fragmented packets, assembles them in a data chunk and sends it to the upper layers. A developer shall take into account that, on the 27 byte payload, 4 byte are occupied by L2CAP header, so the real payload from the Application Layer point of view is 23 byte.

In addition to that this layer acts as protocol multiplexer [35], collecting data coming from different protocols and with different meaning and merging the whole into a BLE packet.

³If the controller does not support DLE (Data Length Extension) 27 byte is the minimum required

During the porting activity on BlueNRG-2, L2CAP implementation has not been significantly explored and studied: this layer is provided by ARM as off-the-shelf part in the ARM Mbed BLE Cordio stack. Its implementation is not provided in the “Appendix A” of this document (the amount of source code is huge), however it is possible to find it inside the ARM Mbed GitHub repository [7], by following the path specified by Table 2.5.

L2CAP layer (BLE stack - Mbed OS 5.11)

Repository	<ul style="list-style-type: none"> • https://github.com/ARMmbed/mbed-os/tree/master/features/FEATURE_BLE/targets/TARGET_CORDIO/stack/cordio_stack/ble-host/sources/stack/l2c
Commit ID	1d2ab42d275fce26717df2781c537ffbb996a856

Table 2.5: L2CAP layer implementation of BlueNRG-2 ARM Mbed porting (ARM Cordio host)

2.5 SM - Security Manager

Security Manager is the BLE stack layer in charge of providing secure procedures for generating and exchanging security keys. This is the basis for setting up encrypted communication channels, remote devices trusted identification, avoid malicious tracking by hiding the public address.

SM defines two roles [35]:

- *Initiator* corresponds to the Link Layer *master* (and therefore the GAP *central*);
- *Responder* corresponds to the Link Layer *slave* (and therefore the GAP *peripheral*).

From the architectural point of view, security features are designed taking into account that responding devices have less computing resources (memory) than initiators. Even if SM is defined as part of host, it involves both the host and the controller: the host part uses hardware features (depending on the cryptography algorithms) of the underlying link layer to generate cryptographic keys and HCI provides access methods to those features for SM.

Link Layer supports encryption and authentication by using the CBC-MAC (Cipher Block-Chaining Message Authentication Code) algorithm and a 128-bit AES-CCM (block cipher) [28].

Using security features implies the appending of additional 4 byte MIC (Message Integrity Check) to the data packet payload.

Bluetooth SIG Core Specifications defines a set of authentication methods depending on the I/O capabilities of initiator and responder, i.e. presence of input peripheral to select yes/no or a numeric keyboard rather than no input, on the output side the presence/absence of a display [28]; the higher the hardware complexity, the higher is the reachable security level. Those methods are (from the simplest to the most secure one): *Just Works*, *Numeric Comparison*⁴, *Passkey Entry* and *OOB (Out Of Band)* [21].

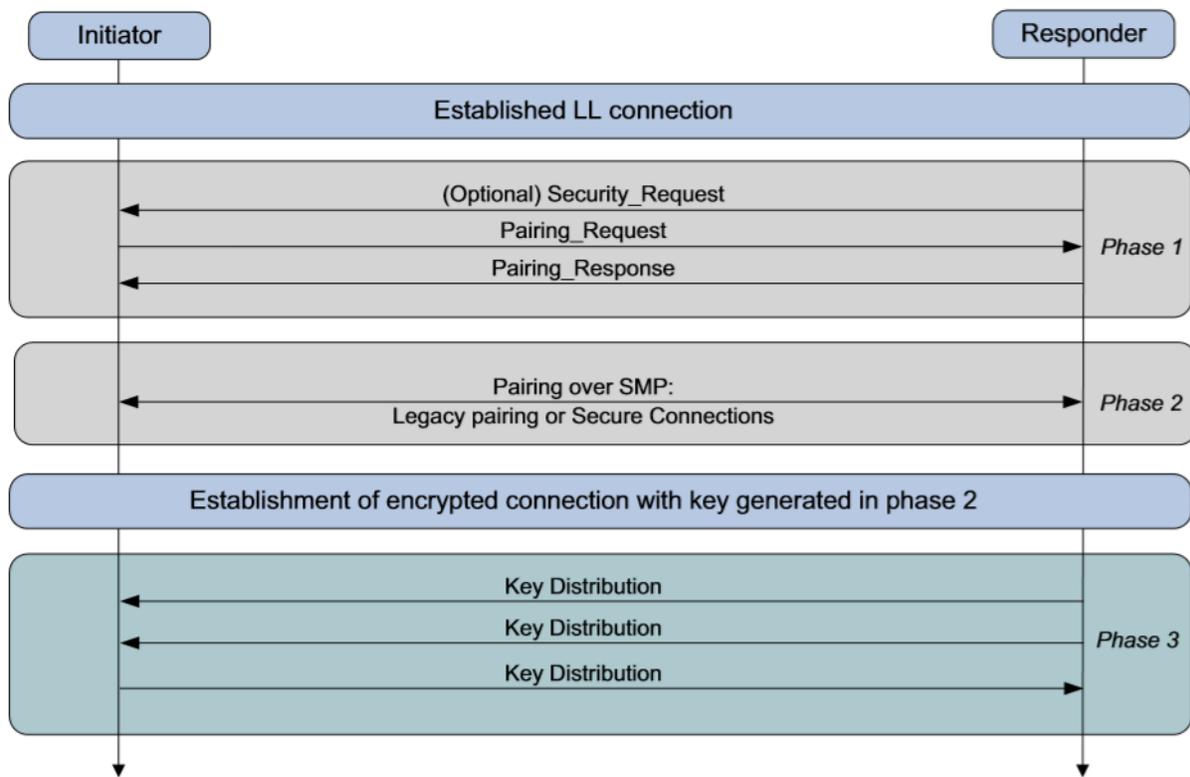


Figure 2.7: LE Pairing Phases [21]

Consortium defines also that encrypted communications on BLE architecture shall start by SM performing the *LE Pairing* procedure: during this phase two devices exchange their identity information and create security keys, used for a trusted relationship establishment. A sequence diagram showing the interaction between initiator and responder during pairing procedure is reported in Figure 2.7. Pairing takes place in 3 phases.

⁴(only available if *LE Secure Connection* is used)

Phase 1 - pairing feature exchange

Connected devices exchange their I/O capabilities. Those information are used to select the key generation method in *phase 2*.

Phase 2 - key generation

This phase depends on 2 different scenarios: if there is a *LE Legacy Pairing* process, then a *STK* (*Short-Term Key*) is generated, else if there is a *LE Secure Connection* follows the generation of a *LTK* (*Long-Term Key*). In both cases there is the generation and exchange of a public-private key pair for each device, based on the *ECDH* (*Elliptic Curve Diffie-Hellman*) algorithm

On BlueNRG-2 this phase involves the Link Layer (in particular PKA hardware unit).

Phase 3 - transport key distribution

Details on how security keys is distributed depends on phase 2 occurrence. However it is possible to identify some common points:

- data signing and verification after the distribution of a *CSRK* (*Connection Signature Resolving Key*);
- distribution of a *IRK* (*Identity Resolving Key*), used for identification of those devices using private addresses.

Concerning BlueNRG-2 Mbed OS porting, SM functionality on the controller side (link layer and PKA) are delegated to the STMicroelectronics static libraries “libbluenrg1_stack.a”(for HCI communication from the host to the link layer) and “cryptolib.a” [29] providing specific API for using cryptography features of the SoC and its hardware accelerators. However, the implementation of these libraries cannot be provided since they are STMicroelectronics classified.

The host side of SM layer in this porting is directly provided by ARM through its Cordio stack. Like L2CAP (Section 2.4), the amount of source code is huge and so not provided in this document, nevertheless it is fully referenced by Table 2.6

SM layer (BLE stack - Mbed OS 5.11)

Repository	<ul style="list-style-type: none">• https://github.com/ARMmbed/mbed-os/tree/master/features/FEATURE_BLE/targets/TARGET_CORDIO/stack/cordio_stack/ble-host/sources/sec/common• https://github.com/ARMmbed/mbed-os/tree/master/features/FEATURE_BLE/targets/TARGET_CORDIO/stack/cordio_stack/ble-host/sources/stack/smp
Commit ID	1d2ab42d275fce26717df2781c537ffbb996a856

Table 2.6: SM layer implementation of BlueNRG-2 ARM Mbed porting (ARM Cordio host)

2.6 ATT - Attribute Protocol

In a BLE technology, from the application point of view, one can identify two kinds of devices: the first ones embed and expose some features, the others can perform some operations on these, like read and write. This kind of structure can be associated to the *client-server paradigm*, and ATT is the layer implementing them in the Bluetooth Low Energy stack architecture.

A BLE device assumes the client or server role with no regard for its master or slave role [35]; they are logically different by the application point of view, moreover a device can be client and server simultaneously. Isolating ATT from the rest of the stack, it could be interpreted as a wire protocol between devices, where server stores attributes and answers client actions: it is the duty of the client to correct format requests and correct interpret answers, server has only to acknowledge or reject.

Nevertheless, even if ATT is mostly based on this paradigm, it is not “pure”: ATT server presents also the feature of *indication* and *notification* (an example is illustrated by “HRM” in Appendix A), it means that a server (always after a client request) can signal the client an attribute change, saving it from continuous polling cycles.

During the porting activity, ATT functional aspects have not been extensively explored (since it is hooked and hidden by GATT and L2CAP, as visible from Figure 2.1). However, in application development is important to understand ATT layer and from the data meaning

point of view: *attributes* are organized in a lookup table and Bluetooth SIG defines for them the structure depicted in Table 2.7.

Attribute Handle	Attribute Type	Attribute Value	Attribute Permissions
0x0008	“Temperature UUID”	“Temperature Value”	“Read-only, no authorization, no authentication”

Table 2.7: ATT - Attribute Example [28]

On BlueNRG-2 Mbed OS porting, this layer is provided by ARM Cordio host stack; its source code can be referenced on the ARM Mbed Cordio target repository by Table 2.8 content.

ATT layer (BLE stack - Mbed OS 5.11)

Repository	<ul style="list-style-type: none"> • https://github.com/ARMmbed/mbed-os/tree/master/features/FEATURE_BLE/targets/TARGET_CORDIO/stack/cordio_stack/ble-host/sources/stack
Commit ID	4e5240b74351ed00ad5b857715da7ff41dfde8d2

Table 2.8: ATT layer implementation of BlueNRG-2 ARM Mbed porting (ARM Cordio host)

As mentioned, ATT is only a “collection” of descriptors, a database of characteristics. A framework for using ATT attributes is provided by GATT layer.

2.7 GATT - Generic Attribute Profile

The *Generic Attribute Profile* layer defines methodologies to exchange profile and user data over a BLE connection [35]. These data consist in services, characteristics, descriptors discoverability, reading, writing, notification and indication properties [28]. Since GATT is a manager for ATT, its role are the same of the latter:

- *GATT server* stores data and provides access methods to a remote GATT client;
- *GATT client* inquires server to expose its services by performing a *service discovery* procedure and accesses them by performing *read, write, notify and indicate* operations.

It implies that, even if there is a logical separation between GATT roles and LL roles (master/slave), there is a correlation with the radio communication mechanisms; it means that a slave shall be always a GATT server, on the contrary the master cannot act as GATT client.

The attributes is in charge of ATT, that defines the following two types: *characteristics* and *services* (collection of characteristics), organized according to the hierarchy shown in Figure 2.8. Heart rate service of the example “HRM” shows how this data organization is implemented.

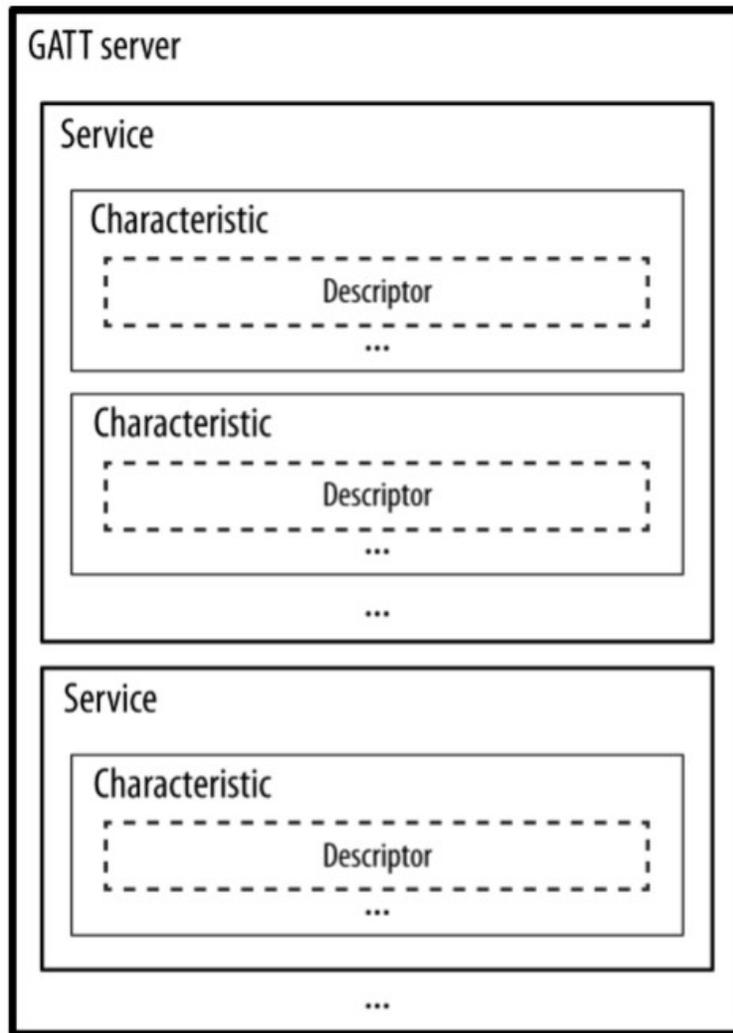


Figure 2.8: GATT Data hierarchy [35]

As shown in Figure 2.1, GATT is at the top on the host side of the stack, so Mbed environment exposes access API to this layer available at the user level for BLE application design.

The source code of this API is referenced in Table 2.9.

GATT API (BLE user API - Mbed OS 5.11)

Repository	• https://github.com/ARMmbed/mbed-os/tree/master/features/FEATURE_BLE/source/generic
Commit ID	4019efb21d8702673ea3e95310a59e0ff95e7cb1

Table 2.9: GATT application program interface for BlueNRG-2 ARM Mbed porting

From the Mbed OS implementation on BlueNRG-2 point of view, a dedicated configuration for disabling GATT features of STMicroelectronics BlueNRG-2 host, described in Section 3.2.7, avoiding its GATT manager initialization and its consequently Flash and SRAM allocation. After that, GATT implementation used in this porting is the one provided by ARM Cordio host with no modifications, whose reference is provided in Table 2.10.

GATT layer (BLE stack - Mbed OS 5.11)

Repository	• https://github.com/ARMmbed/mbed-os/blob/master/features/FEATURE_BLE/targets/TARGET_CORDIO/source/CordioGattServer.cpp
Commit ID	ce11081db79cb4e45928fba04431e6c465e5000e

Table 2.10: GATT layer implementation of BlueNRG-2 ARM Mbed porting (ARM Cordio host)

2.8 GAP - Generic Access Profile

GAP is the layer in charge to handle the advertising process and defines mechanisms related to connection. Moreover it defines the position that a device in a BLE network can cover: *broadcaster* and *observer*, with respect to Broadcast mode (Chapter 1, Figure 1.2) and *central* and *peripheral* with respect to Connection mode (Chapter 1, Figure 1.4).

Broadcaster (Beacon)

Suitable to transmit-only application where the broadcaster sends out periodically advertising packets containing the data payload. A broadcaster device could be theoretically designed with TX-only radio and link layer (but in practise this kind of radio design is never used). The broadcaster uses the advertiser Link Layer role.

An example of broadcaster is a public BLE thermometer [6], sending temperature readings (in the form of advertising packet rather than connection ones) to any interested device in the communication range.

Observer

Receiver side of a receive-only application, it listens for data embedded in advertising packets from broadcasting peers [35]. As said for the broadcaster, the radio part of an observer could be simplified and designed to be RX-only.

Central

It corresponds to the Link Layer master. A device in this role starts operations by listening to other peers advertising packets and then opens a connection with selected devices.

Peripheral

It is the LL slave. This actor advertises packets to allow a central to discover it and then establish a connection. The role is thought for devices with low hardware resources, in terms of computation and power supply.

An example of a couple central-peripheral could be a smart watch connected to a smart-phone⁵.

In the following there are reported two overviews about GAP operating modes and GAP procedures supported by BlueNRG-2 [28]. It can act on any modes implementing any procedure defined by Bluetooth SIG Core Specification, since it is a general purpose BLE application processor, respectively in Table 2.11 and 2.12.

⁵Also the "HRM" (in Appendix A) works as peripheral.

2.8.1 GAP Modes

Mode	GAP procedures	GAP role
Broadcast	Observation	Broadcaster
Non-discoverable	N/A	Peripheral
Limited discoverable	Limited and general discovery	Peripheral
General discoverable	General Discovery	Peripheral
Non connectable	N/A	Peripheral
Connectable (direct)	Direct advertising	Any
Connectable (undirect)	Undirect advertising	Any
Non bondable	N/A	Peripheral
Bondable	Bonding	Peripheral

Table 2.11: GAP modes and role applicability [28]

2.8.2 GAP Procedures

Procedure	GAP role	GAP peer mode
Observation	Observer	Central
Limited discovery	Central	Limited discoverable
General discovery	Central	Limited and general discoverable
Name discovery	Peripheral or central	Peripheral
Auto connection	Central	Connectable (in a <i>white list</i> [21])
General connection	Central	Connectable
Selective connection	Central	Connectable (in a white list)
Direct connection	Central	Connectable (direct)
Connection parameter update	Central	Any connectable
Bonding	Central	Bondable
Terminate	Central	Any connectable

Table 2.12: GAP procedures and mode applicability [28]

Chapter 3

BlueNRG-2 STMicroelectronics System-On-Chip

BlueNRG-2 is a STMicroelectronics BLE (5.0 compliant) application processor, embedding an ARM Cortex-M0 microcontroller.

It is a minor upgrade of its previous version, BlueNRG-1: as a matter of fact it has more Flash memory with respect to the latter, while the whole BLE stack architecture is the same. Its strength lies in the fact that BlueNRG-2 can natively run user application code, so it is more versatile than a network co-processor (like the BlueNRG-MS).

For this reason BlueNRG-2 embeds also all the most widespread useful peripherals in microcontroller based applications. In particular it is provided with [30]:

- High performance, ultra-low power Cortex-M0 32-bit based architecture core;
- Programmable 256 kB Flash;
- 24 kB RAM with retention (two 12 kB banks);
- 1 x UART interface;
- 1 x SPI interface;
- 2 x I²C interface;
- 14, 15 or 26 GPIOs (depends on the package);

- 2 x MFT (MultiFunction Timer);
- 10-bit ADC (for battery charge measurement);
- Watchdog and RTC;
- DMA controller;
- PDM (Pulse Density Modulation) stream processor for audio applications.

BlueNRG-2 pinout is reported in Figure 3.1 (QFN32 package is taken as example).

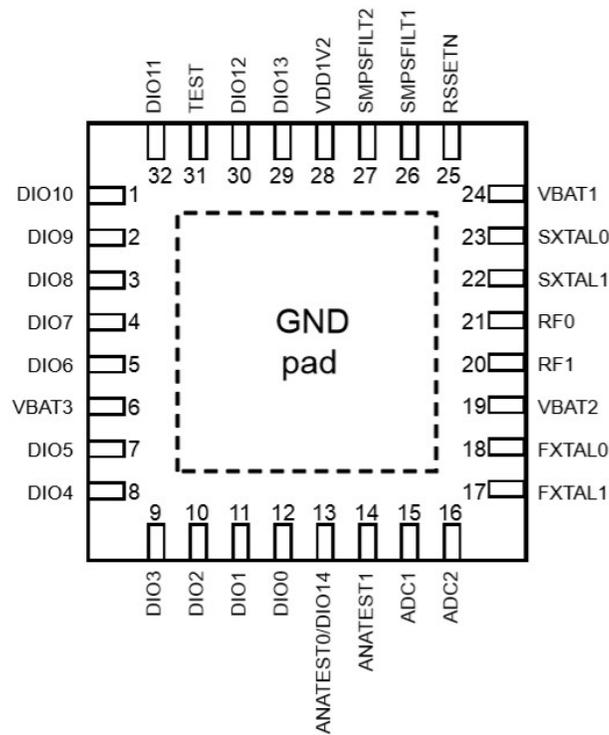


Figure 3.1: BlueNRG-2 pin out top view (QFN32) [30].

BlueNRG-2 architecture and bus interconnection topology is shown in Figure 3.2; the blue blocks represent the BLE Radio and RF front end, white ones the microcontroller architecture and bus interconnection, green is the BLE controller.

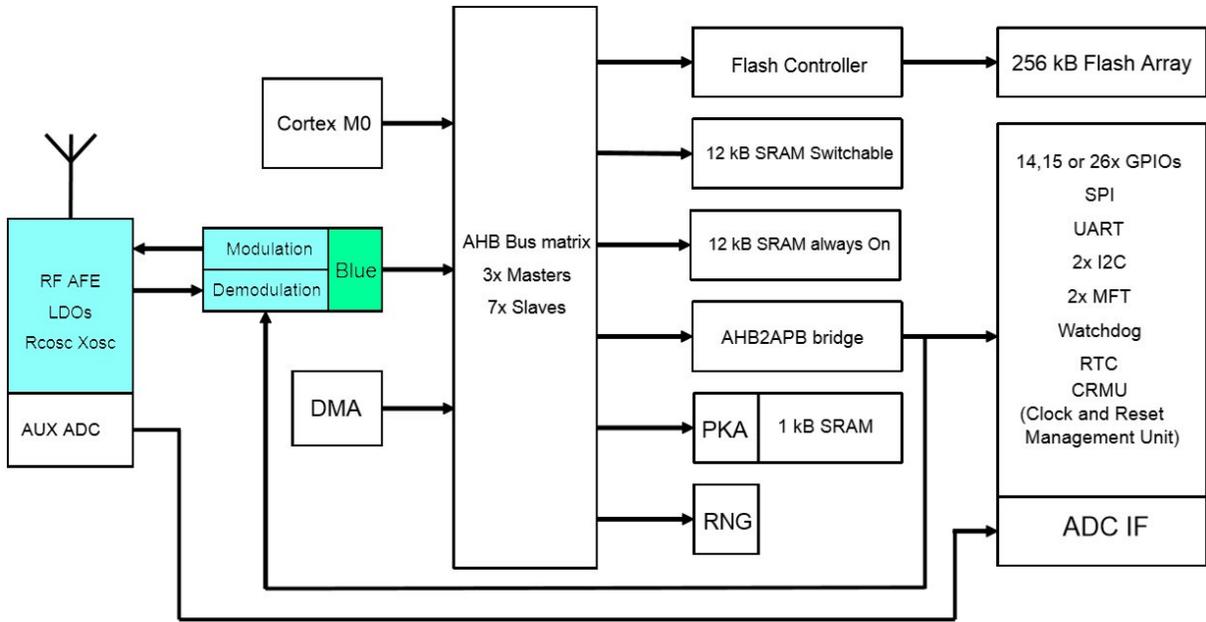


Figure 3.2: BlueNRG-2 datapath architecture (basic blocks). [30]

3.1 ARM Cortex-M0 Core Architecture

ARM Holdings is a British society whose core business is based on the *IP (Intellectual property)* selling of its architectural designs.

BlueNRG-2 is provided with an ARM Cortex-M0 microcontroller architecture. It is an ultra low power processor, with a very low gate count¹ and thus optimized for deeply embedded designs requiring area-optimized cores [5].

ISA - Instruction Set Architecture

ARM Cortex-M0 is a 32-bit *RISC (Reduced Instruction Set Computer)* architecture, based on a specification called *ARMv6-M Architecture*. The bus interface, peripherals interfaces, memory and registers access and internal data paths are 32-bit width [37].

In deep the Cortex-M0 core is designed with a three-stage pipeline architecture, whose basic mechanism is shown in Figure 3.3, keeping low the number of flip flops, hence reducing

¹This is the architectural concept, the final IC design depends on the silicon foundry. In 2009, when Cortex-M0 has been released, it was a demonstration on how to cram a 32-bit machine in a 8-bit processor IC footprint, specified in 12k gates. [37]

latency (jointly with a full pipe after three clock cycles), dynamic power and branch penalty², resulting in a good trade off with the “power efficiency” figure of merit.

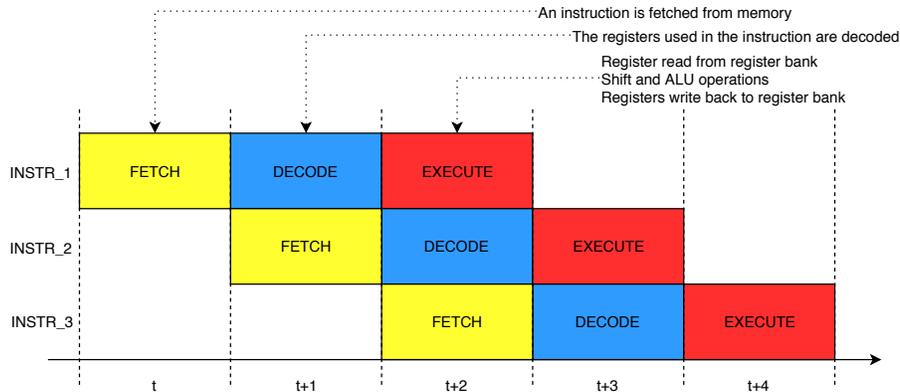


Figure 3.3: Cortex-M0 three-stage pipeline.

To further improve the power efficiency Cortex-M0 presents another microarchitectural-level optimization: it uses a subset (56 instructions) of the *Thumb* ISA with a subset inherited from the 32-bit full *Thumb* and the others are 16-bit in size to improve code density [37]. For this reason, even if this architecture is classified as RISC, this is not completely true since it has an hybrid (in instruction size) ISA. The result of this approach is shown in Figure 3.4.

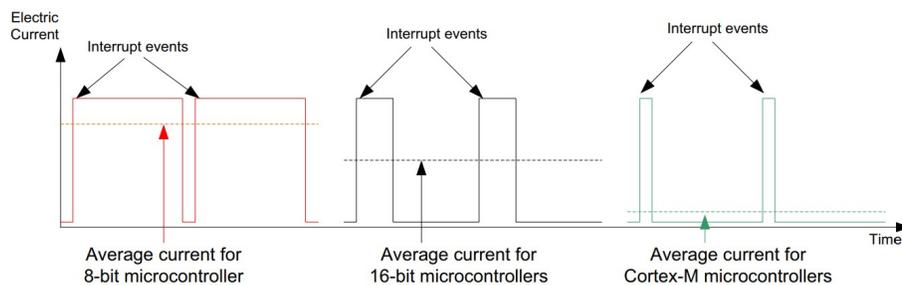


Figure 3.4: Average interrupt current consumption comparison between different architectures. [37]

The system bus interface has a pipelined design approach too and is based on a protocol called *AHB Lite* (*Advanced High performance Bus*). Figure 3.5 represents a Cortex-M0 based microcontroller (including bus and peripherals).

²Compared to the *classic RISC pipeline*.

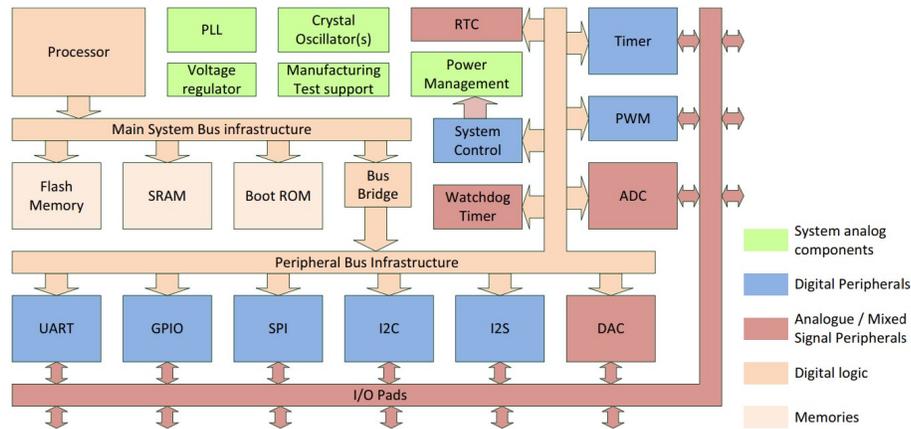


Figure 3.5: Cortex-M0 based microcontroller architecture. [37]

The AHB (System Bus) is a high-speed interconnection within microcontroller core and its basic peripherals (code memory, data memory, SRAM, boot ROM when available), resulting in an overall Von Neumann architecture. It supports 8-16-32-bit data transfers.

Other microcontroller peripherals are interconnected through the *APB (Advanced Peripheral Bus)*. It has a non pipelined architecture, allowing slower peripherals to work apart from the high speed core avoiding bottleneck risks.

The interconnection between System Bus and Peripheral Bus is provided by an AHB-APB bridge. In addition to that BlueNRG-2 has a *DMA (Direct Memory Access)* controller³, allowing DMA-capable peripheral to access the memory without any interrupt to the processing unit.

ARM Cortex-M0 supports hardware interrupt features (discussed at Section 3.2.3) designed in order to reduce ISR execution efficiency as well as software triggered exceptions with higher priority than user code. Moreover it carries also the possibility to selectively switch off the clock for those unused peripherals.

The mentioned features are extremely important in battery-powered wireless systems design, like BLE ones.

³On a Cortex-M0 based architecture this feature is optional. [37]

3.2 Peripherals

This section describes BlueNRG-2 hardware peripherals that has been ported on Mbed OS operating system, or in some way involved in the implementation of some Mbed OS features. Register addresses are reported for sake of completeness, however they are abstracted in STMicroelectronics BlueNRG-2 Development Kit [31] by *CMSIS - Cortex Microcontroller Software Interface Standard* abstraction layer through some intuitive macros (in Section 4.1.1).

Clock Gating

Clock peripheral base register is located at

$$\text{CKGEN_SOC_BASE_ADDR} = 0\text{x}40900000.$$

At the offset address

$$\text{CLOCK_EN} = \text{CKGEN_SOC_BASE_ADDR} + 0\text{x}20$$

there is a 32-bit register allowing unused peripheral clock gating (disable clock). Each bit of this register corresponds to enable/disable for a peripheral (except reserved bits); the bit mask is available in BlueNRG-2 reference manual [30].

Mbed OS high level API enable/disable clock peripherals results in a write on this register.

3.2.1 GPIO

The BlueNRG-2 offers 14 GPIOs (WCSP34 package), 15 GPIOs (QFN32 package) or 26 GPIOs (QFN48 package). The programmable I/O pin can be configured for operating as:

- programmable GPIOs;
- peripheral input or output line of standard communication interfaces;
- 2 PWM (Pulse Width Modulation) sources and 4 PWM output pins;
- 5 wakeup sources from standby and sleep mode;
- each IO pin can generate edge or level interrupts regardless of its mode configuration.

GPIO pin		GPIO mode 000		GPIO mode 001		GPIO mode 100		GPIO mode 101	
Name	Pull ⁴	Type	Signal	Type	Signal	Type	Signal	Type	Signal
IO0	DN	I/O	GPIO 0	I	UART_CTS	I/O	SPI_CLK	O	CPUCLK
IO1	DN	I/O	GPIO 1	O	UART_RTS	I/O	SPI_CS1	I	PDM_DATA
IO2	DN	I/O	GPIO 2	O	PWM0	O	SPI_OUT	O	PDM_CLK
IO3	DN	I/O	GPIO 3	O	PWM1	I	SPI_IN	-	-
IO4	DN	I/O	GPIO 4	I	UART_RXD	I/O	I2C2_CLK	O	PWM0
IO5	DN	I/O	GPIO 5	O	UART_TXD	I/O	I2C2_DAT	O	PWM1
IO6	DN	I/O	GPIO 6	O	UART_RTS	I/O	I2C2_CLK	I	PDM_DATA
IO7	DN	I/O	GPIO 7	I	UART_CTS	I/O	I2C2_DAT	O	PDM_CLK
IO8	DN	I/O	GPIO 8	O	UART_TXD	I/O	SPI_CLK	I	PDM_DATA
IO9	UP	I/O	GPIO 9	I	SWCLK	I	SPI_IN	O	XO16/32M
IO10	UP	I/O	GPIO 10	I	SWDIO	O	SPI_OUT	O	CLK_32K
IO11	UP	I/O	GPIO 11	I	UART_RXD	I/O	SPI_CS1	O	CLK_32K
IO12	NONE	OD	GPI 12	I	-	I/O	I2C1_CLK	-	-
IO13	NONE	OD	GPI 13	I	UART_CTS	I/O	I2C1_DAT	-	-
IO14	DN	I/O	GPIO 14	I/O	I2C1_CLK	I/O	SPI_CLK	-	-
IO15	DN	I/O	GPIO 15	I/O	I2C1_DAT	I/O	SPI_CS1	-	-
IO16	DN	I/O	GPIO 16	O	PWM0	I	SPI_IN	-	-
IO17	DN	I/O	GPIO 17	O	PWM1	O	SPI_OUT	-	-
IO18	DN	I/O	GPIO 18	O	SPI_CS2	O	UART_RTS	-	-
IO19	DN	I/O	GPIO 19	O	SPI_CS3	I	UART_CTS	-	-
IO20	DN	I/O	GPIO 20	I	UART_CTS	O	SPI_CS2	-	-
IO21	DN	I/O	GPIO 21	O	PWM1	I/O	SPI_CS1	-	-
IO22	DN	I/O	GPIO 22	O	PWM0	O	SPI_CS3	-	-
IO23	DN	I/O	GPIO 23	O	UART_TXD	O	SPI_OUT	O	PDM_CLK
IO24	DN	I/O	GPIO 24	I	UART_RXD	I	SPI_IN	I	PDM_DATA
IO25	DN	I/O	GPIO 25	O	UART_RTS	I/O	SPI_CLK	O	PDM_CLK

Table 3.1: BlueNRG-2 IO functional map [30].

By default each pin is configured as *input* with *pull enabled*. Every pin internally has only a pull type internally, according to the Table 3.1; for this reason in Mbed OS GPIO data structures (mentioned in 5.1.1) it has been necessary to replace `PullUp` and `PullDown`

⁴Whether specific pull mode is required, like in I2C, it shall be provided through external resistor.

with PullEnable and PullNone options.

Mbed OS GPIO map data structures has been designed according to Table 3.1.

IO9 and IO10 are *SWD (Serial Wire Debug)* pins. SWD is a subset of the JTAG interface used by ARM microcontrollers with small packages (as BlueNRG-2). These pins, consisting in clock and data lines, provide access to the BlueNRG-2 Cortex-M0 debug unit as a regular JTAG does (real-time access to system memory without halting the processor or requiring any target resident code [2] and regular usage of breakpoints and watchpoints). Its usage is shown in Appendix B.

GPIO base peripheral address is

$$\text{GPIO_BASE_ADDR} = 0x40000000$$

and peripheral registers are shown in Table 3.2 (further details are provided in BlueNRG-2 datasheet). STMicroelectronics releases a GPIO HAL in its BlueNRG DK, that simplify this peripheral usage and setup has been used in Mbed OS GPIO API implementation.

Address offset	Name	RW	Reset	Description
0x00	DATA	RW	0x00000000	IO0 to IO25 data value.
0x04	OEN	RW	0x00000000	GPIO output enable register (1 bit per GPIO).
0x08	PE	RW	0x03FFFFFF	Pull enable (1 bit per IO).
0x0C	DS	RW	0x00000000	IO driver strength (1 bit per IO).
0x10	IS	RW	0x00000000	Interrupt sense register (1 bit per IO).
0x14	IBE	RW	0x00000000	Interrupt edge register (1 bit per IO).
0x18	IEV	RW	0x00000000	Interrupt event register (1 bit per IO).
0x1C	IE	RW	0x00000000	Interrupt mask register (1 bit per IO).
0x20	RIS	R	0x00000000	Raw interrupt status register (1 bit per IO).
0x24	MIS	R	0x00000000	Masked interrupt status register (1 bit per IO).
0x28	IC	W	0x00000000	Interrupt clear register (1 bit per IO).
0x2C	MODE0	RW	0x00000000	Select mode for IO0 to IO7.
0x30	MODE1	RW	0x00000110	Select mode for IO8 to IO15.
0x34	MODE2	RW	0x00000000	Select mode for IO16 to IO23.
0x38	MODE3	RW	0x00000000	Select mode for IO24 to IO25.
0x3C	DATS	RW	0x00000000	Set some bits of DATA when in GPIO mode without affecting the others (1 bit per IO).

0x40	DATC	RW	0x00000000	Clear some bits of DATA when in GPIO mode without affecting the others (1 bit per IO).
0x44	MFTX	RW	0x00000000	Select the IO to be used as capture input for the MFTX timers.

Table 3.2: BlueNRG-2 GPIO registers [30].

3.2.2 Wake up Controller and Reset

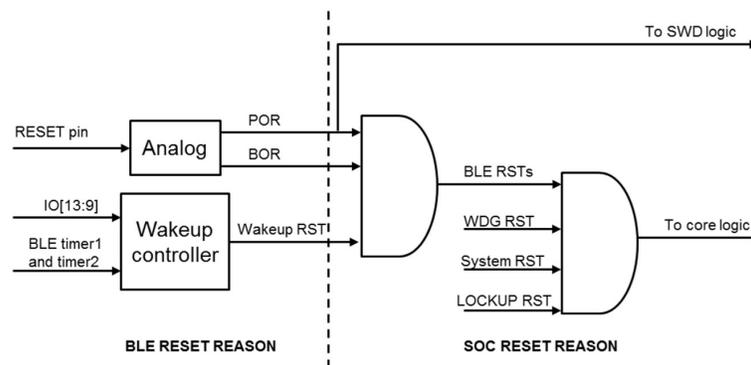


Figure 3.6: BlueNRG-2 wake up logic and reset generation [30].

General principle for wake up and reset logic is shown in Figure 3.6. Releasing the pin puts the chip out of shutdown state; at first the wake up logic is powered and receives the *POR* (*Power On Reset*). Whether the circuitry logic decides to wake up from *sleep* or *deep sleep* mode, it generates a *core logic reset*. The latter can also be triggered by watchdog expiry event, a *system reset* or a *lockup reset*. System reset does not affect the debugger connection, while lockup reset does not occur with debugger attached.

The reset procedure correct timing is shown in Figure 3.7.

3.2.3 NVIC

The *Nested Vectored Interrupt Controller* handles 48 exceptions divided in:

- 16 Cortex-M0 specific interrupts (0x00 - 0x3C offset address range);
- 32 device peripheral interrupts (0x40 - 0xBC offset address range).

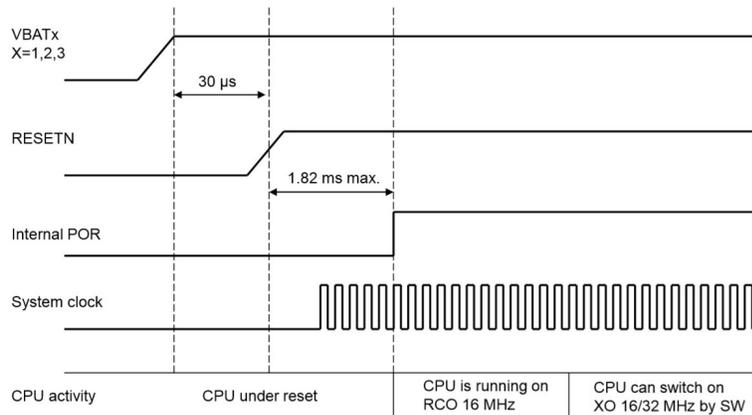


Figure 3.7: BlueNRG-2 power-up sequence [30].

On Cortex-M0 based devices *ISR vector table* `ISR_BASE_ADDR` can be only mapped on the Flash base address or the SRAM base address⁵. The ISR vector table is listed in Table 3.3; SRAM *remap* operation is in charge of the *Flash controller* (remap procedure is described in 3.2.6).

Position	Priority	Priority type	Description	Address
	-3	Fixed	Reset handler	0x00000004
	-2	Fixed	NMI handler	0x00000008
	-1	Fixed	HardFault handler	0x0000_000C
			RESERVED	0x00000010 - 0x00000028
	3	Settable	SVC handler	0x0000002C
			RESERVED	0x00000030 - 0x00000034
	5	Settable	PendSV handler	0x00000038
	6	Settable	SystemTick handler	0x0000003C
0	Init 0	Settable	GPIO interrupt	0x00000040
1	Init 0	Settable	FLASH controller interrupt	0x00000044
2	Init 0	Settable	RESERVED	0x00000048
3	Init 0	Settable	RESERVED	0x0000004C
4	Init 0	Settable	UART interrupt	0x00000050
5	Init 0	Settable	SPI interrupt	0x00000054
6	Init 0	CRITICAL	BLE controller interrupt	0x00000058
7	Init 0	Settable	Watchdog interrupt	0x0000005C

⁵There is no *VTOR - Vector Table Offset Register* defining a location, like in Cortex-M3 and M4 based devices.

8	Init 0	Settable	RESERVED	0x00000060
9	Init 0	Settable	RESERVED	0x00000064
10	Init 0	Settable	RESERVED	0x00000068
11	Init 0	Settable	RESERVED	0x0000006C
12	Init 0	Settable	RESERVED	0x00000070
13	Init 0	Settable	ADC interrupt	0x00000074
14	Init 0	Settable	I2C 2 interrupt	0x00000078
15	Init 0	Settable	I2C 1 interrupt	0x0000007C
16	Init 0	Settable	RESERVED	0x00000080
17	Init 0	Settable	MFT1 A interrupt	0x00000084
18	Init 0	Settable	MFT1 B interrupt	0x00000088
19	Init 0	Settable	MFT2 A interrupt	0x0000008C
20	Init 0	Settable	MFT2 B interrupt	0x00000090
21	Init 0	Settable	RTC interrupt	0x00000094
22	Init 0	Settable	PKA interrupt	0x00000098
23	Init 0	Settable	DMA interrupt	0x0000009C
24 – 31	Init 0	Settable	RESERVED	0x000000A0 – 0x000000BC

Table 3.3: BlueNRG-2 ISR vector table [30].

3.2.4 MFT

BlueNRG-2 has two *Multi Functions Timers*; main features are [30]:

- two 16-bit programmable timer counters;
- two 16-bit reload/capture registers (depending on the *mode* of operation);
- an 8-bit fully programmable prescaler (shared by both MFT);
- clock source selector in *pulse accumulate* mode, *external event* mode or *system clock* with settable prescaler;
- two I/O pins (TnA - TnB) with settable edge detection operating as *capture* inputs;

- two interrupts (one per timer) triggerable by *capture* event, timer *reload* or *underflow*⁶.

MFT can be used in 5 different modes and in each mode both MFT peripherals and registers have a certain configuration; the complete list of MFT peripherals mode operation is provided in the reference manual [30].

In this porting project MFT *Mode 3* has been used: timers have been configured as *dual independent timer/counter* peripherals. It allows to use MFT1 and MFT2 separately. The operation performed in *Mode 3* is shown in Figure 3.8.

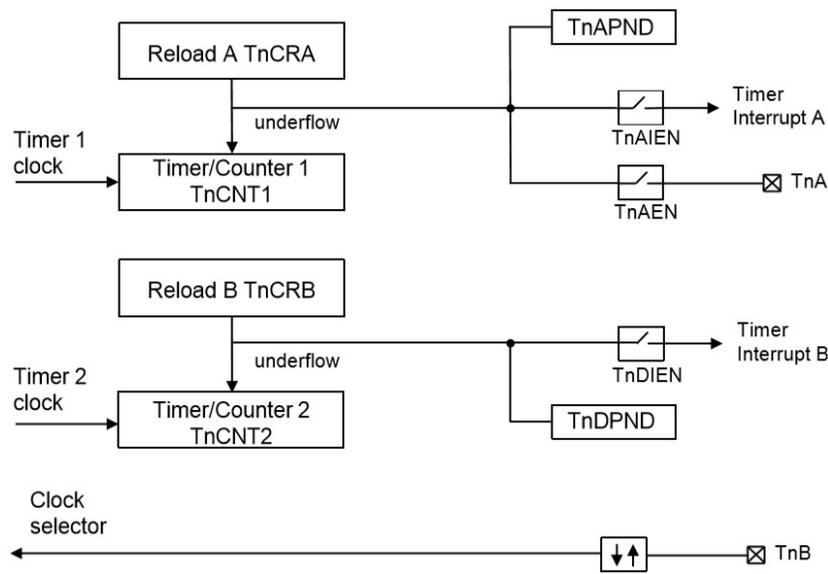


Figure 3.8: BlueNRG-2 MFT mode 3 block diagram [30].

More in general, *Mode 3* configures the peripheral to operate as dual independent *system timer* or external *event counter*. In addition, the Timer/Counter 1 can generate a 50% duty cycle PWM signal on the TnA pin. TnB pin can be used as *external-event* or *pulse-accumulate* input and provide clock either to MFT1 or MFT2. Both counters clock can be prescaled (but there is only a prescaler for both peripherals).

Timer/Counter 1 counts down at the selected clock speed. Upon underflow, TnCNT1 (count register) is reloaded at TnCRA (compare register) value and count proceeds. Whether enabled, TnA can toggle at each underflow event, generating the 50% PWM signal without any

⁶It is not allowed to use MFT as forward counters, but only backward: for this reason only overflow interrupt is not available. It has been taken into account and discussed in the microsecond ticker design, Section 5.2.5.

CPU interaction. In addition to that, underflow event sets the TnAPND (pending interrupt bit) value and an exception could be generated if TnAIEN (interrupt enable) is asserted.

Timer/Counter 2 works as same as Timer/Counter 1, with the only limitation in generating PWM output (not available).

Peripheral base addresses are located at

$$\text{MFT1_BASE_ADDR} = 0x40D00000,$$

$$\text{MFT2_BASE_ADDR} = 0x40E00000$$

and peripheral registers are shown in Table 3.4 (further details available in BlueNRG-2 reference manual [30]).

Address offset	Name	RW	Reset	Description
0x00	TnCNT1	RW	0x00000000	Timer/Counter1 register.
0x04	TnCRA	RW	0x00000000	Capture/Reload A register.
0x08	TnCRB	RW	0x00000000	Capture/Reload B register.
0x0C	TnCNT2	RW	0x00000000	Timer/Counter 2 register.
0x10	TnPRSC	RW	0x00000000	Clock prescaler register.
0x14	TnCKC	RW	0x00000000	Clock unit control register.
0x18	TnMCTRL	RW	0x00000000	Timer mode control register.
0x1C	TnICTRL	RW	0x00000000	Timer interrupt control register.
0x20	TnICLR	RW	0x00000000	Timer interrupt clear register.

Table 3.4: BlueNRG-2 MFTx registers [30].

3.2.5 UART

BlueNRG-2 integrates a *Universal Asynchronous Receiver/Transmitter* supporting much of the features of 16C650 UART [30] industry standard. These are:

- programmable baud rates up to 2 Mbps;
- programmable data frame of 5, 6, 7 or 8 bits of data;
- even, odd, stick or no-parity, 1 or 2 stop bit generation and detection;
- support hardware (RTS/CTS) and software (Xon/Xoff) flow control;

- false start bit detection;
- line break generation and detection;
- programmable 64 words FIFO, 8 bit + 4 status (optional) word length;
- DMA support.

Base peripheral address is

UART_BASE_ADDR = 0x40300000

and its registers are described in Table 3.5. Further details are provided in BlueNRG-2 datasheet [30].

Address offset	Name	RW	Reset	Description
0x00	DR	RW	0x00000000	Data register.
0x04	RSR	R	0x00000000	Receive status register.
0x08	ECR	W	0x00000000	Error clear register. Write to clear framing (FE), parity (PE), break (BE), and overrun (OE) errors.
0x0C	TIMEOUT	RW	0x000001FF	Timeout register.
0x18	FR	R	0x00001E90	Flag register.
0x1C	LCRH_RX	RW	0x00000000	Receive line control register.
0x24	IBRD	RW	0x00000000	Integer baud rate register.
0x28	FBRD	RW	0x00000000	Fractional baud rate register.
0x2C	LCRH_TX	RW	0x00000000	Transmit line control register.
0x30	CR	RW	0x00040300	Control register.
0x34	IFLS	RW	0x00000012	Interrupt FIFO level select register.
0x38	IMSC	RW	0x00000000	Interrupt mask set/clear register.
0x3C	RIS	R	0x00000000	Raw interrupt status register.
0x40	MIS	R	0x00000000	Masked interrupt status register.
0x44	ICR	W	0x00000000	Interrupt clear register.
0x48	DMACR	RW	0x00000000	DMA control register.
0x50	XFCR	RW	0x00000000	XON/XOFF control register.
0x54	XON1	RW	0x00000000	Xon1 character used for software flow control.
0x58	XON2	RW	0x00000000	Xon2 character used for software flow control.
0x5C	XOFF1	RW	0x00000000	Xoff1 character used for software flow control.

0x60	XOFF2	RW	0x00000000	Xoff2 character used for software flow control.
------	-------	----	------------	---

Table 3.5: BlueNRG-2 UART registers [30].

3.2.6 Memory

As said in the beginning of this Chapter, BlueNRG-2 is designed with a Von Neumann architecture approach. It means that CPU registers, code and data memory, peripherals and I/O are addressable through the whole machine memory space (4 GB for a 32-bit machine).

BlueNRG-2 is coded in little Endian format, the word least significant byte is the lowest numbered byte. The addressable memory space is divided into 16 main 256 MB blocks (not the whole space is addressed and it is marked as “RESERVED”).

All the peripherals are addressed by APB, except DMA, RNG and PKA peripherals that are addressed by AHB. These three hardware accelerators moreover shall be accessed only with 32-bit accesses: any 8-bit or 16-bit access generates an AHB error leading to a hard fault on Cortex-M0.

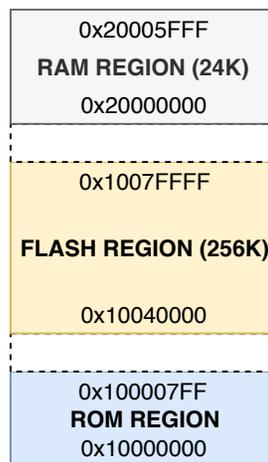


Figure 3.9: BlueNRG-2 memory address space.

The whole addressable space organization is provided in BlueNRG-2 datasheet [30]; the memory space (Flash and SRAM) is shown in Figure 3.9. The 2K ROM section stores, among the others, a pre-programmed (from STMicroelectronics at manufacturing) bootloader. It gives memory access whether the previously programmed user firmware locks the SWD interface

GPIO. This bootloader is executed (instead of Flash firmware) by forcing to GND the IO7 pin, and it is useful either to reprogram the Flash or to instruct the Flash controller (e.g. perform a *mass erase*).

Flash Controller

BlueNRG-2 integrates a Flash controller to interface its embedded 256 KB flash memory array. This array is composed by 128 pages containing 8 rows of 64 words ($128 \times 8 \times 64 = 65536$ words, 32-bit per word). Write operation consists in writing a '0' it means that write a logic '1' implies a previous erase. The ADDRESS inside its own register is built as

$$\text{ADDRESS}[15:0] = \text{XADR}[9:0] \ \& \ \text{YADR}[5:0]$$

with:

- $\text{XADR}[9:3]$ = page address;
- $\text{XADR}[2:0]$ = row address;
- $\text{YADR}[5:0]$ = word address (one word = four bytes);

Flash controller implements the necessary logic to carry out the Flash memory operations (program/erase) through an instruction and data access interface based on the AHB protocol. Its operations are controlled by peripheral registers starting from the base address

$$\text{FLASH_BASE_ADDR} = 0\text{X}40100000.$$

A detailed description of Flash controller registers is reported in the BlueNRG-2 datasheet [30]. For Mbed OS porting on BlueNRG-2 purpose it is proper to mention the FLASH - CONFIG register at

$$\text{FLASH - CONFIG} = \text{FLASH_BASE_ADDR} + 0\text{X}04.$$

It allows the ISR vector table *remap* operation (mandatory for Mbed OS) from Flash memory to SRAM base address, according to the description provided in Table 3.6.

Bit	Field name	Reset	RW	Description
0	RESERVED	0	RW	RESERVED
1	REMAP	0	RW	Remaps the interrupt vector table in RAM
2	RESERVED	0	RW	RESERVED
3	PREMAP	1	RW	Remaps the interrupt vector table in FLASH
31:4	RESERVED	0	RW	RESERVED

Table 3.6: BlueNRG-2 FLASH - CONFIG register description [30].

3.2.7 BLE

BlueNRG-2 integrates a BLE specification compliant RF transceiver, yielding the standard regulation in the unlicensed $2.4GHz$ ISM band. The RF transceiver requires very few external components, providing $96dB$ link budget with excellent link reliability, keeping the peak current below $15mA$.

In TX, the *PA (Power Amplifier)* drives the signal generated by the frequency synthesizer out to the antenna terminal through a simple matching network (delivered power as well as the harmonic content depends on the external seen impedance by PA).

Several operating modes are available for BlueNRG-2 radio: *Reset mode*, *Sleep mode*, *Active mode* and *Radio mode*.

Reset mode is entered asserting the active-low external reset signal; in this mode the SoC voltage regulators, clocks and RF front end are powered down.

In *Sleep mode* either the LS (*Low Speed*) crystal oscillator or the LS ring oscillator are running, whereas HS (*High Speed*) sources and RF circuitry is off; Link Layer FSMs and SRAM content are retained. In this mode BlueNRG-2 waits for a GPIO interrupt or an internal *Virtual Timer* peripheral expiration to wake up.

In *Active mode* everything is powered on: MCU, RF interface, power supplies and oscillators. *Radio mode* adds to active mode the TX and RX capability.

Link Layer setup

BlueNRG-2 Link Layer supports up to 8 simultaneous links; the number of active links is settable with a preprocessor directive. The usage of BlueNRG-2 in its “full” link layer configuration mode has a drawback: it requires both Flash and SRAM memory space. For this reason, in this experimental porting activity, BlueNRG-2 has been configured with the so-called `BLE_STACK_BASIC_CONFIGURATION`, i.e.:

- no Controller privacy;
- no LE secure connection;
- no Master GAP role;
- no Data Length Extension;
- 1 active Link Layer FSM.

In addition to that BlueNRG-2 is put in a special mode, the so-called *link-layer only*. It allows to remove from final build the STMicroelectronics host stack, resulting in more free-space for ARM Cordio host. This setup is done at runtime, during ARM Mbed OS BLE initialization (described in Section 5.3.1), by calling the procedure

```
aci_hal_write_config_data(0x2C, 1, &one);
```

where `&one` points to a user-defined memory location containing an unsigned integer equal to 1.

BlueNRG-2 *Link Layer only* is an experimental mode: it is not fully tested and supported by STMicroelectronics, nevertheless its current developed version has been resulted sufficiently mature to support this explorative porting activity.

Chapter 4

ARM Mbed OS 5

Mbed OS ecosystem comes out at the beginning of this decade with the idea (similar to the *Arduino One* project) of simplifying the design flow of embedded systems based on ARM architecture and encouraging the development of smart devices in the IoT era. It lies on the idea of *rapid prototyping*: providing a set of layers that “interprets” the user code, it allows the latter to work on different ARM microcontrollers in a uniform way [10]. Today ARM Mbed OS growth has been significant for the following reasons.

Despite code compilers relies on heuristic optimization engines (hence not always converging to the absolute minimum), they are becoming ever more efficient in terms of final binary code size; moreover a modern-day case of study are *machine learning compilers*, capable of reaching optimization levels¹ not allowed by heuristic methods [36].

In addition to that Mbed OS has been created with a modular design, providing an high level of flexibility in customization. As a matter of fact developers can easily include/exclude many features in their Mbed OS build, with regard of the final application.

Furthermore, starting from Mbed OS 5, it provides a complete IoT solution in terms of connectivity (i.e. *Pelion Device Management*): it enhances Mbed OS with cloud features, simplifying the trusted relationship establishment between a huge number of different devices and natively implementing the most widespread security protocols. This allows an high flexibility and a long product lifecycle due to the simplicity in distributing updates and extensions. An overview of an ARM Mbed OS based IoT infrastructure is shown in Figure 4.1

¹Not only on size or execution speed, but also on other new aspects, like *power efficiency*.

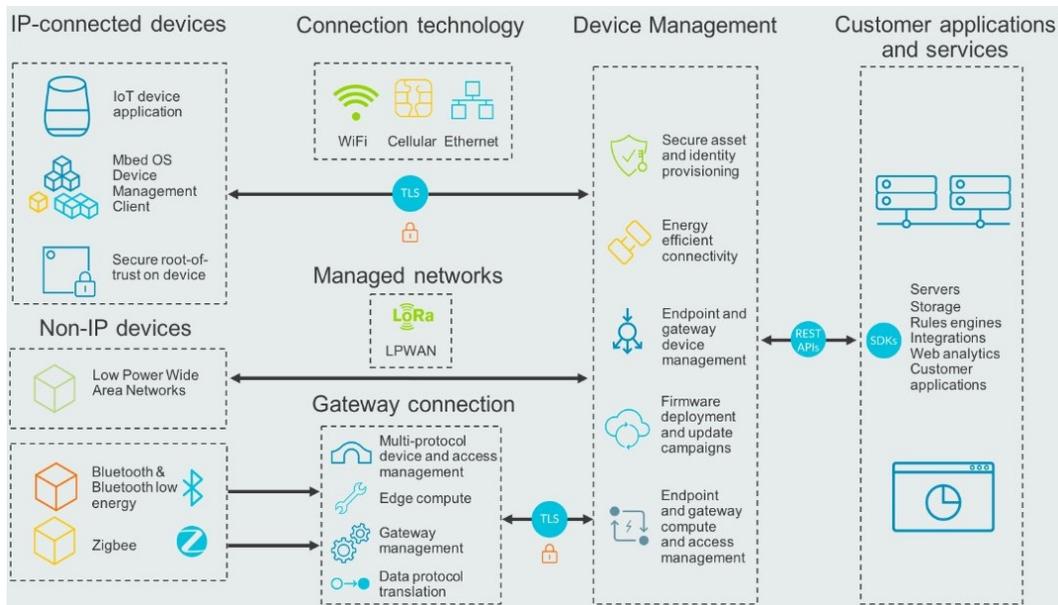


Figure 4.1: Mbed OS 5 IoT infrastructure (ARM Pelion-based). [10]

What has just been said means that today Mbed OS is not to be intended only as a *rapid prototyping* platform: it can be taken into account as complete solution and concrete option for new IoT designs.

4.1 HAL Architecture

Mbed OS provides an abstraction layer for those (ARM-based) microcontrollers it runs on, allowing designers to focus on writing C/C++ applications, in such a way it is possible to reuse the firmware on any Mbed-enabled platform [10].

Using this *HAL (Hardware Abstraction Layer)* design simplifies the integration (and hides the mechanisms) of the most widespread microcontroller peripherals, such as timers. As consequence, it is possible to selectively enable (or disable) a feature in a simple automatic way, by only including (or not) in the final project. The overall basic architecture of what runs on an Mbed-enabled board or module is shown in Figure 4.2.

Mbed OS supports an RTOS core (based on *CMSIS RTX RTOS*): it enables multithreading, real-time and deterministic firmware execution providing features like *mutexes*, *semaphores* and *threads* to the application level. RTOS feature is the most powerful of Mbed OS, but also

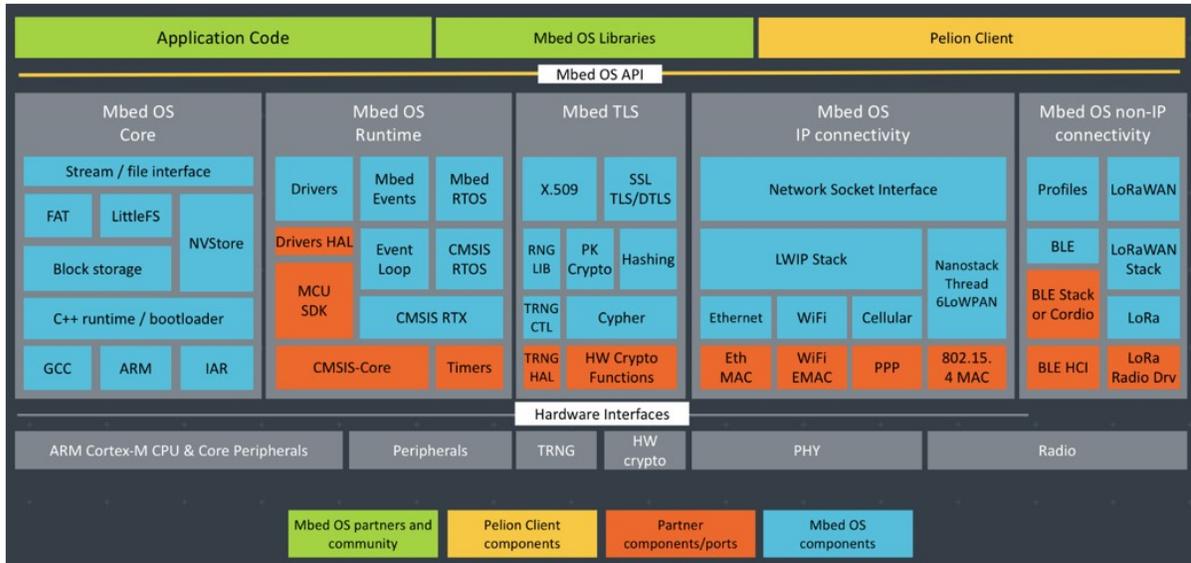


Figure 4.2: Mbed OS 5 architecture

the most resource-hungry; for this reason it has been ported and tested on BlueNRG-2, but not released, since it is not possible to use it when the Bluetooth Low Energy stack is running, even if in minimal configuration. Therefore, the RTOS porting is not mentioned in the *Porting Chapter (5)*.

Mbed OS is IoT-oriented, for this reason it comes out with a multilevel security model required by IoT products. It is able to exploit the low-level features and hardware accelerators provided by ARM silicon partners for data securing and identification [10].

The structure of ARM Mbed OS supports also *File System* technology: whereas an application requires a specific block level storage, it is feasible to choose the best file system fitting the IoT device. The *FAT* file system (e.g. in a datalog system backed by a micro SD card) allows the inter-operability between Mbed OS and the general purpose operating systems (Windows, MacOS, Linux), the whole is completed by a low-level support to the most common transport layers (like SPI).

On the bottom side Mbed OS integrates a retargeting layer, allowing a system-level abstraction of the bootstrap procedures and the integration with different toolchains [10]. Orange blocks in the bottom layer, shown in Figure 4.2, are the starting point building blocks for new target porting.

4.1.1 Layer description

More in details the Mbed OS project is split according to the following hierarchy (these are the sub-folder grouping the Mbed OS source code).

- *CMSIS Core*
- *Events*
- *RTOS*
- *Components*
- *Features*
- *Targets*
- *Drivers*
- *HAL*
- *Tools*

CMSIS Core

CMSIS Core (Cortex Microcontroller Software Interface Standard) implements the basic run-time system for a Cortex-M device and gives the user access to the processor core and the device peripherals [15]. It defines:

- *HAL* for Cortex-M processor registers with standardized definitions for the SysTick, NVIC, System Control Block registers, MPU registers, FPU registers, and core access functions.
- *System exception names* to interface to system exceptions without having compatibility issues.
- *Methods to organize header files* that makes it easy to learn new Cortex-M microcontroller products and improve software portability. This includes naming conventions for device-specific interrupts.
- *Methods for system initialization* to be used by each MCU vendor. For example, the standardized `SystemInit` function is essential for configuring the clock system of the device.
- *Intrinsic functions* used to generate CPU instructions that are not supported by standard C functions.
- A variable to determine the *system clock frequency* which simplifies the SysTick timer setup.

The detailed file structure of the CMSIS-Core device templates is shown in Figure 4.3.

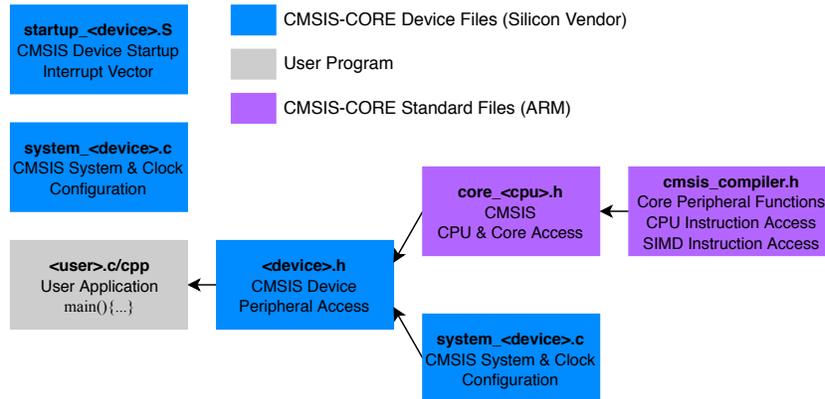


Figure 4.3: CMSIS Core File Structure. [15]

Components

It contains a collection of connectivity and inter-operability features. First of all it provides a security framework for IoT connectivity (*ARM PSA - Platform Security Architecture*); it provides also the HAL (up to Link Layer) for the *Personal Area Network* technology and *Mesh (802.15.4)*, as well as the enabling HAL for Wi-Fi and connectivity over IP. In terms of inter-operability it provides a *File System* feature abstraction layer.

Features

It contains additional (with respect to *components*) IoT connectivity technologies, like: BLE Host, Lightweight IP stack, cellular, TLS features, NFC, USB Host, etc.

BLE Host has been involved in this porting activity and it is discussed more in details at the end of this Section (*ARM Cordio BLE Host*).

Drivers

This part contains the user-level API, providing the access to the microcontroller peripherals, such as `Ticker`, `DigitalIn`, `DigitalOut`, `InterruptIn`, `Serial`, `Timer`, etc.

They are designed in C++, allowing the firmware designer to use the full C++ syntax and access the low level functionality by means of classes and objects relations, including the powerful features of *inheritance* and *polymorphism* of to the *OOP (Object Oriented Paradigm)*. For example, `DigitalOut` is a clever abstraction of the GPIO: it initializes the peripheral as GPIO output, capable of driving a LED, with only a simple code line

```
DigitalOut led(LED1);
```

where the constructor parameter is simple an enumeration literal defining the peripheral pin (further details in Section 5.1.1).

In addition to that, these drivers implements some useful *operators overloading*, in order to simplify the most widespread operations, like toggling a led in the following instruction.

```
led != led;
```

There are some peripherals, like GPIO, totally abstracted by its drivers (`DigitalOut`, `DigitalIn`), on the contrary other ones have a direct match, like `SPI.cpp`, `I2C.cpp`.

A full description of Mbed OS API is available in the Mbed OS reference book [10].

Events

The Event class provides APIs to configure events delay and period timings. It is feasible to execute operation like `post` an event to the underlying `EventQueue`, and `cancel` the most recently posted event [10].

The most powerful feature of an `EventQueue` is the background execution: it is capable to execute the dispatch loop in a transparent way, exploiting a ticker in RTOS-less design² and avoiding to the user every synchronization problems. An event *post* simply performs a *callback* registration (for one-shot or periodic execution); it is then automatically serialized and scheduled in the queue dispatch loop.

Serialization is the most intuitive way to deal with *Bluetooth Low Energy*: this technology is based on a commands and events flow, that can be easily handled by a queue of events (all Mbed OS BLE examples are based on this mechanism).

²A queue can run on a separate thread context in RTOS mode and multiple queues are admitted, each one with its own dispatch loop

At the end, another useful feature is given by the *deferred execution*: instead of executing complex tasks from ISR context, an interrupt signals events to the `EventQueue` and the latter executes ISRs from the dispatch loop context.

HAL

These are the Mbed OS API implementation and contains the access methods to the base peripherals. Each target shall implement its version of this HAL API and this implementation is required to be in C. Further details on the implemented API are available in Section 5.2.

Targets

It contains the CMSIS abstraction layer of each Mbed-enabled device, defines also the toolchain support, the startup code and linker files. *Targets* are organized in a hierarchical way described in 5.1.1.

RTOS

The Mbed OS RTOS capabilities include managing objects such as threads, synchronization objects and timers. It also provides interfaces for attaching an application-specific idle hook function, reads the OS tick count and implements functionality to report RTOS errors [10].

It provides a strong enhancement to Mbed OS in terms of synchronization and code safety, with the drawback of a huge increase in Flash and SRAM occupation, often unacceptable on limited-resources systems like BlueNRG-2.

Tools

Mbed OS includes many tools to simplify the development process, in terms of components configurations, toolchains integration and support. Besides it provides some testing tools, most of them are *python scripts* exploited by the Mbed CLI (4.2.2).

4.1.2 ARM Cordio BLE Host

ARM Cordio is a complete radio IP supporting Bluetooth 5 protocol, compatible with different ultra low power silicon technologies (from 55 to 40 nm). ARM has designed Link Layer and Physical Layer firmware, as well as a flexible and modular host. This host is open source and implemented by ARM Mbed OS; for this reason it has been ported on BlueNRG-2.

ARM Cordio Host stack consists of³:

- Attribute Protocol and Profile (ATT and GATT);
- Generic Access Profile (GAP);
- Security Manager Protocol (SMP);
- L2CAP;
- Wireless Software Foundation (WSF) portable OS services and wrappers (optimized for Cortex-M architecture);
- HCI (“thin” *ExactLE* HCI layer or full transport-based HCI).

This host is optimized for battery powered resource constrained devices, it has a low RAM footprint (10 kB), supporting master and slave roles and providing capability of acting as client or server. Besides it is extremely modular: mentioned features can be selectively disabled (through intuitive *macros* and *defines* automatically generated by the Mbed OS *library configurator*) to reduce the overall Flash memory footprint.

4.2 Design Tools

The ARM Mbed ecosystem provides many tools to enhance the development on Mbed OS platform and simplify the configuration procedures throughout the whole design process. Moreover it embeds a validation system and an off-the-shelf solution for code testing.

The two Mbed OS development tools are *Mbed Online Compiler* and *Mbed CLI (Command Line Interface)*. Both of them allow to perform two basic tasks [10]:

³Flyer available at: <https://www.arm.com/files/pdf/ARM-Cordio-Stack>.

- bring Mbed OS code from Mbed online repositories (with all dependencies);
- compile the code for a *target* and provide a single executable file to flash onto the Mbed device.

For non-conventional design (like porting of new devices) Mbed OS ecosystem provides also *exporters* capable of project automatic creation for the most widespread IDEs.

4.2.1 Mbed Online Compiler

The Mbed Online Compiler provides a lightweight C/C++ IDE, preconfigured to quickly write code, compile and flash on the target device. Any install neither any preliminary setup is required to use the online compiler: it is a web app, which is why only a browser (it is a platform-independent solution, available on every operating system) and a login are required. A screenshot of the Mbed Online Compiler is provided in Figure 4.4.

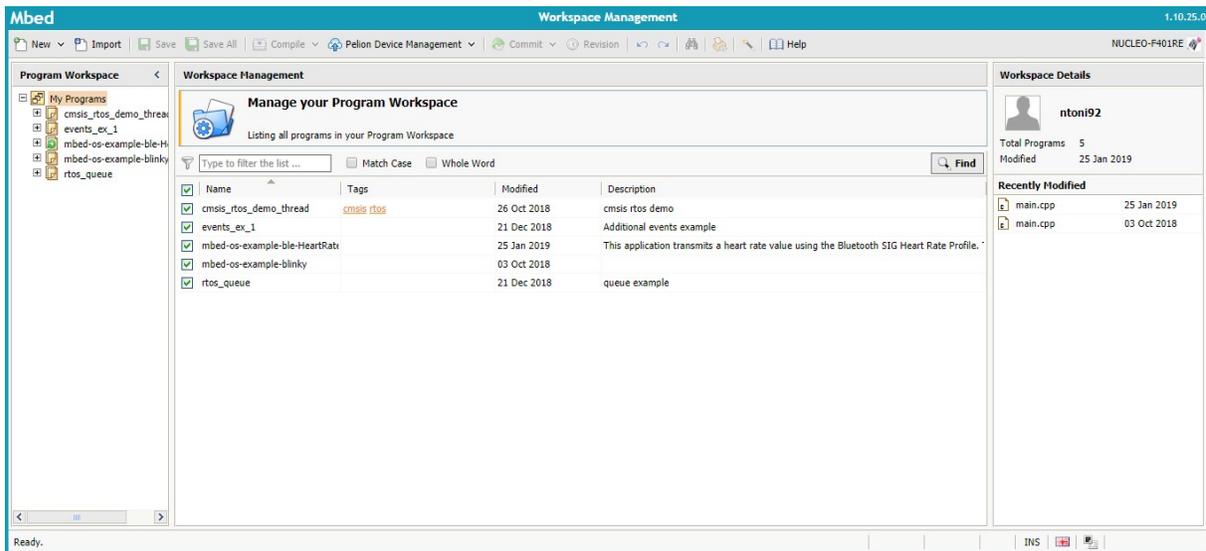


Figure 4.4: Mbed Online Compiler.

Despite it is a web app, it is extremely powerful and versatile: it includes code formatting and auto-generation of documentation features, syntax highlighting, undo/redo and cut/copy/paste keyboard shortcuts and even a code formatter. Its compilation engine is based on ARMC5 or ARMC6 (depending on the device HAL architecture).

Another important feature is the integrated *version control*: it allows project collaboration, branch and merge of firmware code, through a simplified interface and a *Git*-like model [11]. At the end it can be used free-of-charge even in commercial applications.

Its limitation is represented by *debug* because it does not provide debugging tools; for this reason, for those project where it is required, there's the need to *export* to third party complete tools (as described, for this porting project, in Appendix B).

4.2.2 Mbed CLI

ARM Mbed CLI is the command line tool packaged as `mbed-cli`, based on Python and supporting a BASH-like syntax. It enables all the Online Compiler features, like Git- and Mercurial-based version control, dependencies management, code publishing, support for remotely hosted repositories (GitHub, GitLab and mbed.org), use of the Arm Mbed OS build system and export functions and other operations. A complete user guide, as well as installation requirements and procedure of the Mbed CLI is provided in the Mbed OS Reference Guide [10].

.mbedignore

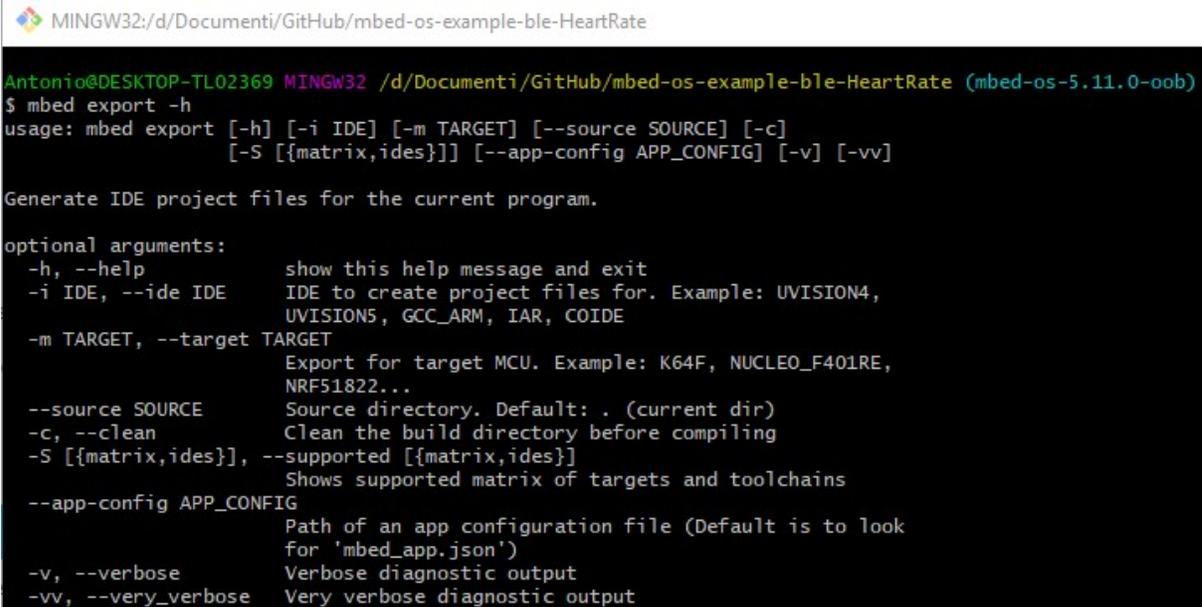
The *.mbedignore* file can be placed in any directory where the Mbed build system is going to search for source files and dependencies. Paths (defined in a BASH-like style) inside this file are ignored and excluded from the Mbed build. This kind of approach is useful, since it avoids developers to writes the definitions of a huge quantity of preprocessor macros deciding whether or not include files. A convenient place for the *.mbedignore* file is the project root directory.

4.2.3 Exporting

The Mbed ecosystem exporters allow to develop Mbed OS applications on third party IDEs and toolchains (among the others *Keil uVision*, *Eclipse CDT*, *IAR EWARM*, etc.). Moreover it is feasible to migrate an existing Mbed OS project into another IDE project, either from the Mbed Online Compiler or Mbed CLI.

With respect to the Online Compiler exporter, it allows to download a project into a *.zip* archive containing a generated project in the specified toolchain format.

Concerning a local project Mbed CLI generates the third party project files in the target exported project through the `mbed export` command (its guide is reported in Figure 4.5). At the end it is sufficient to import the project into the selected IDE (or open the just created project file).



```
MINGW32:/d/Documenti/GitHub/mbed-os-example-ble-HeartRate
Antonio@DESKTOP-TL02369 MINGW32 /d/Documenti/GitHub/mbed-os-example-ble-HeartRate (mbed-os-5.11.0-oob)
$ mbed export -h
usage: mbed export [-h] [-i IDE] [-m TARGET] [--source SOURCE] [-c]
                  [-S [{matrix,ides}]] [--app-config APP_CONFIG] [-v] [-vv]

Generate IDE project files for the current program.

optional arguments:
  -h, --help                show this help message and exit
  -i IDE, --ide IDE         IDE to create project files for. Example: UVISION4,
                           UVISION5, GCC_ARM, IAR, COIDE
  -m TARGET, --target TARGET
                           Export for target MCU. Example: K64F, NUCLEO_F401RE,
                           NRF51822...
  --source SOURCE           Source directory. Default: . (current dir)
  -c, --clean               Clean the build directory before compiling
  -S [{matrix,ides}], --supported [{matrix,ides}]
                           Shows supported matrix of targets and toolchains
  --app-config APP_CONFIG  Path of an app configuration file (Default is to look
                           for 'mbed_app.json')
  -v, --verbose             Verbose diagnostic output
  -vv, --very_verbose      Very verbose diagnostic output
```

Figure 4.5: Mbed CLI project exporter help.

Additional information about exporters, up-to-date compatibility issues between IDEs and toolchains and required configurations are available in the Mbed OS Reference Guide [10].

Chapter 5

Porting

Mbed OS is the ARM ecosystem for the Internet of Things. Since it is an open source platform, each user can contribute to its growth in many different ways: by fixing some bugs or defects in the current implementation, refactoring some functionality in a more efficient way, updating documentation and updating targets.

These activities can be done internally, in a direct contact with ARM supervisors, or externally by opening a *Pull Request* on GitHub ARM Mbed OS repository [7] from a previously created *fork* on ARM Mbed OS *master* repository (i.e. a clone on the own GitHub repository of the ARM Mbed OS develop branch [11]). Exploiting the pull request procedures is absolutely the most convenient option: ARM provides some automatic tests for the *CI* (*Continuous Integration*), license check and the compatibility with the most widespread toolchains, allowing a developer to avoid common initial errors at the beginning of the activity.

The porting of a new target (STMicroelectronics BlueNRG2) in the ARM Mbed ecosystem is classified as “target update”. There was a preliminary study by the company itself consisting in the porting of some basic peripheral and a reduced set of BLE features on BlueNRG-1, but there was any attempt of porting the full BLE stack on a STMicroelectronics BLE SoC with the proposed approach. Since it has had a highly experimental connotation, an hybrid develop approach has been exploited:

- direct contact with ARM Cordio Host Stack supervisors to propose and discuss some guidelines, concerning BLE, to be adopted without exposing to the whole Mbed OS team

some implementation-specific details;

- pull request to correct some issues and refine some details mandatory to CI, integration with the Online Compiler, Mbed CLI compliance and other things potentially useful for Mbed Community and for porting this kind of target SoC devices.

Since, as previously said, this activity has been designed as experimental for evaluating the feasibility and give a feedback about supporting ARM Mbed OS on BLE application processors, the porting has not been done in a full way, but some important points have been identified and developed.

The full porting targets list is defined by ARM [9] and divided into the following tasks.

- | | |
|---------------------------------------|----------------------------------|
| 1. Setting up (HW and SW) | 9. RTC |
| 2. Bootstrap and reset handler | 10. SPI |
| 3. IRQ | 11. TRNG |
| 4. GPIO | 12. Connectivity |
| 5. Serial port | 13. Flash |
| 6. Low power ticker | 14. Bootloader |
| 7. Microsecond ticker | 15. Pelion Client (optional) |
| 8. Tickless | 16. Other HAL modules (optional) |

The implemented basic set of tasks in this porting are the highlighted ones. Source code developed during this activity is quite huge, for this reason it is not entirely reported in this document; nevertheless it is fully available on the GitHub repository:

```
https://github.com/ntoni92/mbed-os-BlueNRG2/tree/master/targets/TARGET\_STMBLUE.
```

Details about porting are described and discussed more in detail in the following sections. Unless otherwise specified, mentioned code is available under this root (relevant excerpts are directly reported, where needed for reasons of clarity, inside each section or in Appendix A).

5.1 Setting up (Hardware and Software)

The work carried out during this period starts from choosing a suitable setup for the development environment. This phase, that consists in the choose of an *IDE (Integrated Development Environment)*, a toolchain and a development board, can be considered a preliminary, and is described apart in Appendix B.

As prerequisite of a generic Cortex-M based device porting on Mbed OS there is also the implementation of the *CMSIS core*.

CMSIS Core

CMSIS-Core (Cortex Microcontroller Software Interface Standard) implements the basic runtime system for a Cortex-M device and gives the user access to the processor core and the device peripherals [15]. More in detail it contains the *HAL* (low level map of addresses into more user-friendly names) for processor registers, Interrupt Controller and peripherals.

CMSIS support is provided by the file

```
https://github.com/ntoni92/mbed-os-BlueNRG2/blob/master/  
    targets/TARGET_STMBLUE/TARGET_BLUENRG2/  
    TARGET_STEVAL_IDB008V2/device/BlueNRG2.h
```

and included in the STMicroelectronics BlueNRG-2 DK [31].

Peripheral HAL is available in the directory

```
https://github.com/ntoni92/mbed-os-BlueNRG2/tree/master/  
    targets/TARGET_STMBLUE/Periph_Driver
```

and provided in a couple of “*.c” and “*.h” files for each BlueNRG-2 peripheral.

5.1.1 Target Description

Mbed OS integrates information about supported devices in a “*json*” file. It is located at *targets/targets.json* under the Mbed OS root directory [7]. This file organizes the supported devices in a hierarchical way shown in Figure 5.1, and each supported device shall respect one of these hierarchies.

```

MCU -> Board
MCU -> Module -> Board
Family -> MCU -> Board
Family -> MCU -> Module -> Board
Family -> Subfamily -> MCU -> Board
Family -> Subfamily -> MCU -> Module -> Board

```

Figure 5.1: Mbed OS target hierarchical organization [10]

The support of BlueNRG-2 has been added by following the first one in Figure 5.1; the code added in *targets.json* file for the BlueNRG-2 support in Mbed OS ecosystem is available and described in the following.

```

"BLUENRG2": {
  "inherits": ["Target"],
  "core": "Cortex-M0",
  "supported_toolchains": ["GCC_ARM"],
  "release_versions": ["5"],
  "default_lib": ["std"],
  "public": false,
  "extra_labels": ["STMBLUE", "CORDIO"],
  "device_has": ["SERIAL", "SERIAL_ASYNC", "SERIAL_FC", "INTERRUPTIN", "
    SLEEP"],
  "macros": ["BLUENRG2_DEVICE", "LS_SOURCE=LS_SOURCE_EXTERNAL_32KHZ", "
    SMPS_INDUCTOR=SMPS_INDUCTOR_10uH", "BLE_STACK_CONFIGURATION=
    BLE_STACK_BASIC_CONFIGURATION", "CMSIS_VECTAB_VIRTUAL", "
    CMSIS_VECTAB_VIRTUAL_HEADER_FILE=\"cmsis_nvic.h\""],
  "features": ["BLE"]
},
"STEVAL_IDB008V2": {
  "inherits": ["BLUENRG2"],
  "extra_labels_add": ["STMBLUE"],
  "device_has_add": ["STDIO_MESSAGES", "USTICKER"],
  "config": {
    "clock_source": {
      "help": "Crystal oscillator frequency",
      "value": "HS_SPEED_XTAL_32MHZ",
      "macro_name": "HS_SPEED_XTAL"
    },
    "clock_source": {
      "help": "Controller tick time (in milliseconds)",
      "value": "200",
      "macro_name": "TICK_MS"
    }
  }
}
}
}

```

First of all, STMicroelectronics already has its hierarchy inside Mbed OS because of its support to STM32 ARM Cortex-M based MCUs. However, even if BlueNRG-2 is a Cortex-M0 based device, its hardware peripherals and therefore HAL is much different from the STM32; for this reason a new hierarchy have been defined for the specified STMicroelectronics BLE processor.

MCU - BLUENRG2

The entry BLUENRG2 describes the microcontroller: it *inherits* properties from the basic Mbed Target, defining its Cortex-M0 core base and the support for GCC_ARM toolchain. This entry is not `public`, it means that it cannot be directly exported but is only defined as parent. `macros-add` creates a preprocessor directive for each listed macro, at the end `features` contains directives for the build system where it has to scan for resources [10].

Board - STEVAL_IDB008V2

With respect to MCU section it adds some macros containing information about some components connected to the SoC, the high-speed crystal oscillator frequency and the stack library configuration. In addition to that it provides information about available HAL; to include them in the building process, the Mbed OS build system defines a macro composed by `DEVICE_` followed by the string defining the HAL in the `device_has` list.

The STEVAL_IDB008V2 is shown in Figure 5.2.

Together with the target definition, Mbed OS defines a method to map buttons and LEDs connected to the target on a specific board¹. This is done through the definition of the `PinName` enumeration inside the header

```
https://github.com/ntoni92/mbed-os-BlueNRG2/blob/master/  
targets/TARGET_STMBLUE/TARGET_BLUENRG2/  
TARGET_STEVAL_IDB008V2/PinNames.h
```

and allows, for instance, the use of the literal LED1 without knowing on which BlueNRG-2 pin it is connected.

¹More in general, Mbed OS allows to redefine the name for the GPIO connected to.

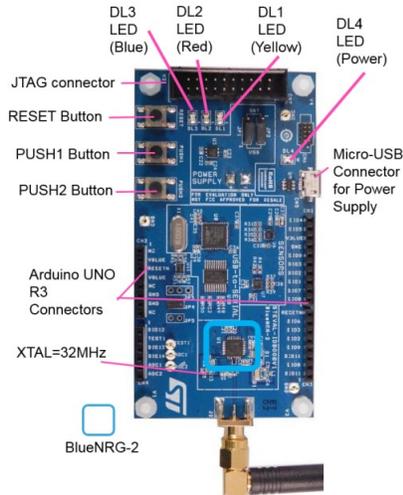


Figure 5.2: STEVAL_IDB008Vx [29]

5.2 Hardware API and Peripheral Drivers

Once terminated the preliminary setup required by Mbed build system, the porting activity has moved on the startup procedure design, the correct memory map and then the HAL implementation.

5.2.1 Startup Routine and Linker Script

Bootstrap operation is defined mainly by two phases: the design of a startup routine and locating it at the address where the control unit starts to execute the program after the reset. This second point is part of a more general memory operation that consists in creating a correct mapping of the whole BlueNRG-2 memory.

Reset Handler

The RESET_HANDLER is located at

https://github.com/ntoni92/mbed-os-BlueNRG2/blob/master/targets/TARGET_STMBLUE/hal/src/system_bluenrg1.c

named *system_bluenrg1.c*². A reset routine has been provided for the *ARMC5*, *ARMC6 IAR*,

²The *bluenrg1* statement has been kept to be consistent with the BlueNRG DK nomenclature.

`GCC_ARM` toolchain; in the following lines the reference one is the `GCC_ARM RESET_HANDLER` (identified by a `#ifdef __GNUC__`) directive. This code is written in a mixed C and inline assembly way, the reset execution flow is provided in Figure 5.3.

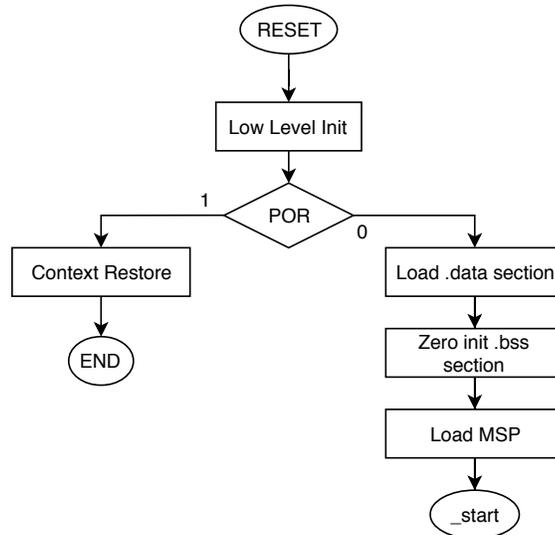


Figure 5.3: `RESET_HANDLER` flowchart (startup part 1).

First of all this startup routine performs a `low_level_init`, that checks a status flag to identify the reset reason, returning 1 if the reason is a *POR* (*Power-On Reset*) or performing a `CS_contextRestore` when the reason is a wake up from *sleep*, then resuming the execution from the point it has been halted.

In case of *POR*, the routine proceeds by loading the *.data* segment initializers from Flash memory to SRAM and by filling the *.bss* section with zero values (additional details about the meanings of this memory sections are provided in the *Linker Script* paragraph). At the end, the entry point for the application is loaded into the *MSP* (*Master Stack Pointer*) and the Mbed OS start function is called.

Even if the reset procedure is terminated, the startup procedure is not completed: inside the Mbed OS context there is a `mbed_sdk_init` procedure (in `mbed_overrides.c` file) that performs other low level initialization, whose behavior is defined in Figure 5.4.

This routine performs an initialization for the analog circuitry (SMPS - Switched Mode Power Supply and LDO - Low DropOut regulators configuration, oscillators setup and calibration, BOR - BrownOut Reset detection setup), then the controller-side stack initialization.

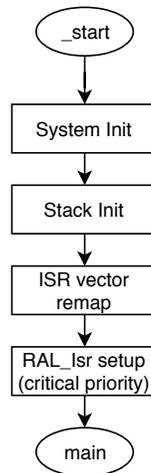


Figure 5.4: MBED_SDK_INIT flowchart (startup part 2).

Startup is completed by setting `RAL_ISR` interrupt service routine to the `BLUE_CTRL_IRQn` (i.e. the BlueNRG-2 radio controller) and the NVIC priority of this interrupt to the highest (`0 - CRITICAL`, mandatory for correct radio events handling).

Linker Script

Where the source code of a program is split into multiple files (Mbed OS is coded in more than 16000 files), each file is processed on its own. The building process is divided into four steps [16], an overview is given in Figure 5.5.

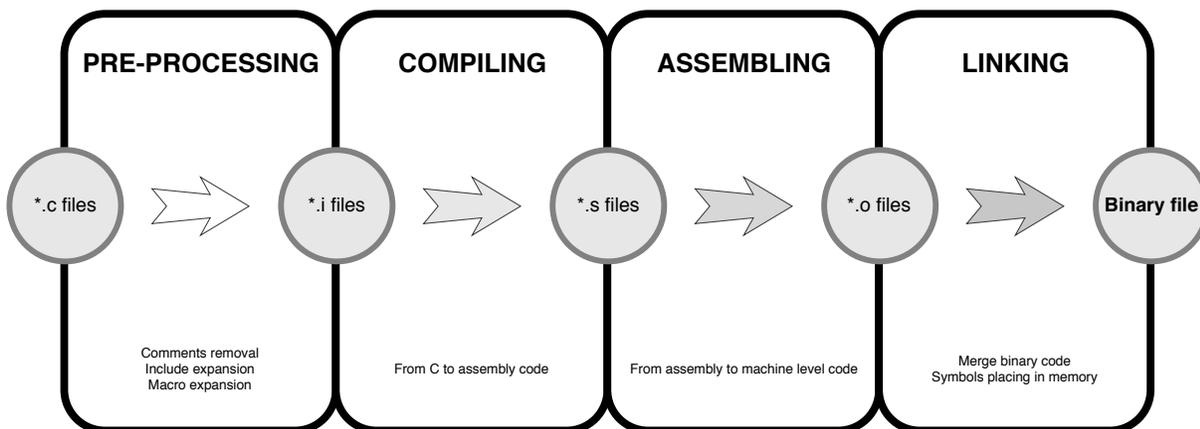


Figure 5.5: Building process flow.

The final merging of intermediate created files is in charge of the *linker*. Linking process

on a *bare metal* system³ consists in a *static linking*, it means that the produced binary file shall include all the required library (after all there is any possibility of dynamic runtime library load on bare metal). Static linking does:

- *symbol resolution*, i.e. univocally mapping of each symbol reference into a symbol definition, in a multi-file design at this stage symbols are marked with E (right now defined and relocatable), U (undefined), D (already defined) and at the end of this phase any U symbol shall exists, otherwise a link error is raised;
- *relocation*, i.e. the map of each symbol onto a specific code memory location.

The latter determines the final location of all the memory sections, defined in the following.

.text section is the section of program's virtual memory space containing executable instructions; it is mapped in ROM, i.e. on the Flash memory on BlueNRG-2. At the beginning of this section (placed at the Flash base address, 0x10040000 on BlueNRG-2) of this section there is located the startup routine, followed by the ISR vector table.

Along with BlueNRG-2 stack definitions, it contains virtual methods table, C++ constructors and destructors required by Mbed OS framework.

.bss section contains static uninitialized variables. This section shall be minimized because it is entirely located in RAM (i.e. on the BlueNRG-2 SRAM). There are other sections, with different names, that can be logically considered part of .bss: they have been defined apart from .bss for a dedicated configuration of some BlueNRG-2 stack library data structures (i.e. `.bss.__blue_RAM`).

.data section contains initialized data, that is initialized variables and static global variables. On this BlueNRG-2 porting it contains some compile-time configuration of the ST stack library (i.e. BLE_BASIC_CONFIGURATION, 3.2.7) and Mbed OS default configuration. This section is located in BlueNRG-2 Flash memory but after reset is copied into SRAM.

³*Bare Metal* indicates a computing system in which application are not built on an operating system.

.heap region is a BlueNRG-2 SRAM region with variable size, dedicated to *dynamic allocation* operations (`malloc`, `free`, etc.); it is a shared region, accessible from all Mbed OS tasks. Its beginning is located from contiguously to the end of the `.bss` section and grows towards the `stack` region.

stack region is the classical program stack. In particular it is a *LIFO (Last In First Out)* memory segment that starts at the end of the SRAM space and grows backwards the `.heap` region. When the latter two regions overlaps, an error is raised (*stack overflow* error).

The linker script file has been developed only for the GCC_ARM toolchain, and it is provided at

```
https://github.com/ntoni92/mbed-os-BlueNRG2/blob/master/
    targets/TARGET_STMBLUE/TARGET_BLUENRG2/
TARGET_STEVAL_IDB008V2/device/TOOLCHAIN_GCC_ARM/BlueNRG2.ld.
```

5.2.2 IRQ and NVIC

Mbed OS requires the implementation of a mechanism to change ISRs at runtime. Taking into account limitations of ARM Cortex-M0 core, detailed in 3.1, this is done by adding to the CMSIS-Core HAL the following files:

```
https://github.com/ntoni92/mbed-os-BlueNRG2/blob/master/
    targets/TARGET_STMBLUE/TARGET_BLUENRG2/
TARGET_STEVAL_IDB008V2/device/cmsis_nvic.c
```

containing the BlueNRG-2 ported code and, in the same directory, `cmsis_nvic.h` header containing definitions. The CMSIS-style function that allows to change the ISR behavior are `NVIC_SetVector` and `NVIC_GetVector`.

The latter only returns the handler address of the IRQ passed as input parameter; the first has been implemented on BlueNRG-2 Mbed OS porting in order to change the ISR vector location from Flash to SRAM base address (by asserting the proper bit in the `FLASH->CONFIG` register), copy the entire ISR vector on its first call (i.e. in `RESET_HANDLER`), and on successive calls map only the desired ISR for the desired IRQ. Signature and implementation of

NVIC_SetVector are reported in the following listing.

```
void NVIC_SetVector(IRQn_Type IRQn, uint32_t vector) {
    uint32_t *old_vectors = (uint32_t *)NVIC_FLASH_VECTOR_ADDRESS;
    // Copy from flash and switch to dynamic vectors if first time called
    if (FLASH->CONFIG == FLASH_PREMAP_MAIN) {
        for (int i = 0; i < NVIC_NUM_VECTORS; i++) {
            *((uint32_t *) (NVIC_RAM_VECTOR_ADDRESS + (i*4))) =
                old_vectors[i];
        }
        FLASH->CONFIG = FLASH_REMAP_RAM;
    }
    // Set the vector
    *((uint32_t *) (NVIC_RAM_VECTOR_ADDRESS + (IRQn*4) + (
        NVIC_USER_IRQ_OFFSET*4))) = vector;
}
```

5.2.3 GPIO

This API provides access to the General Purpose Input/Output digital lines of BlueNRG-2 and its implementation provides to the STMicroelectronics SoC capability to:

- setup a pin to drive a logic value (`DigitalOut`);
- acquire a logic value from a pin (`DigitalIn`);
- use a pin in a bidirectional way implementing the latter two features at the same time (`DigitalInOut`);
- allow a pin to interrupt the BlueNRG-2 logic core when an event of rise or falling edge (or both) occurs and execute a user-defined attached ISR (`InterruptIn`).

In particular, the first three features are enable by implementing the Mbed OS API `gpio_api.h`, the latter through `gpio_irq_api.h` in the directory <https://github.com/ntoni92/mbed-os-BlueNRG2/tree/master/hal>.

Implementation are provided in files (following the same nomenclature for the API headers)

https://github.com/ntoni92/mbed-os-BlueNRG2/blob/master/targets/TARGET_STMBLUE/gpio_api.c

and

https://github.com/ntoni92/mbed-os-BlueNRG2/blob/master/targets/TARGET_STMBLUE/gpio_irq_api.c.

To implement these two files, it is mandatory to define two GPIO data structures, and Mbed OS guidelines suggests to do this in a file *objects.h* within the `mbed-os/targets/TARGET_VENDOR/TARGET_MCU_FAMILY/TARGET_MCUNAME` directory. Further details are provided into the following paragraphs.

gpio_api.c

First of all `gpio_t` data structure has been implemented and imported. BlueNRG-2 HAL already provided a suitable GPIO data structure in its driver set (*BlueNRG1_gpio.h*), so only a redefinition has been sufficient.

```
#define gpio_t GPIO_InitType
```

The primary functions of this driver are `gpio_init` and `gpio_dir`. Mbed OS classes defining digital ports (`DigitalIn`, `DigitalOut`, `DigitalInOut` and `InterruptIn`) calls them to define the behavior of the initialized pin. The implementation is provided in the following listing.

```
//initialize gpio pin
void gpio_init(gpio_t *obj, PinName pin) {
    obj->GPIO_Pin = (uint32_t)pin;
    // preset to output by default
    obj->GPIO_Mode = GPIO_Output;
    obj->GPIO_Pull = ENABLE;
    obj->GPIO_HighPwr = ENABLE;
    // Enable the GPIO Clock
    SysCtrl_PeripheralClockCmd(CLOCK_PERIPH_GPIO, ENABLE);

    GPIO_Init(obj);
    obj->GPIO_Mode = GPIO_Output;
}

//set gpio pin direction
inline void gpio_dir(gpio_t *obj, PinDirection direction) {
    obj->GPIO_Mode = (uint8_t)direction;
    GPIO_Init(obj);
    obj->GPIO_Mode = direction;
}
```

gpio_irq_api.c

To enable IRQ from GPIO peripheral, the `gpio_irq_s` data structure has been defined. It contains the EXTI (EXternal Interrupt) structure defined in the BlueNRG-2 peripheral HAL GPIO file and an `id` field; it is shown in the following listing.

```
struct gpio_irq_s{
    GPIO_EXTIConfigType exti;
    uint32_t id
};
```

This driver provides an initialization routine `gpio_init`, that basically sets interrupt to be edge-triggered by writing a zero in the `exti.GPIO_IrqSense` register (Mbed OS I/O high level class does not allow level sensitive interrupts, even if BlueNRG-2 has this feature it is not available to user level). In addition to that it sets `GPIO_Handler` as GPIO IRQ handler; the latter clears the pending IRQ bit in the NVIC and executes the Mbed OS callback `irq_handler`.

Mbed OS high level interrupt API are capable of configuring a GPIO interrupt in four different modes: `IRQ_NONE`, `RISE_ENABLE`, `FALL_ENABLE`, `BOTH_ENABLE`. On BlueNRG-2, GPIO IRQ configuration is slightly different: this SoC has a register (`exti.GPIO_event`) containing only `rise`, `fall` or `both`. The `none` interrupt status is obtained by writing the IE (Interrupt Enable) register. For this reason, a map between Mbed OS API and BlueNRG-2 EXTI controller has been obtained by implementing the adaption depicted in Figure 5.6 and the code in the following listing.

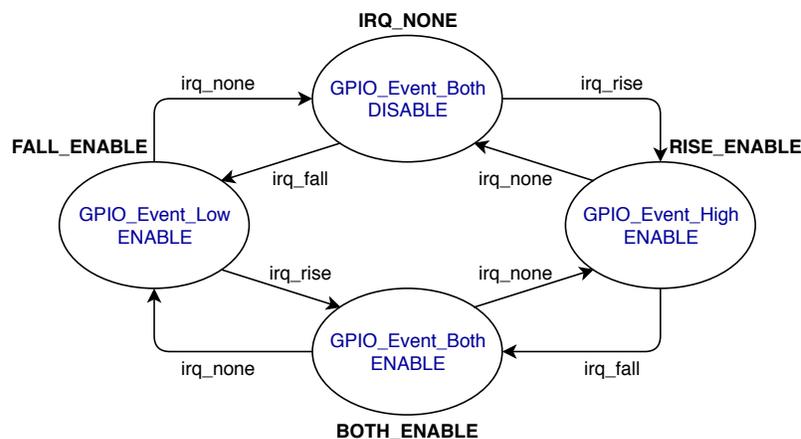


Figure 5.6: BlueNRG-2 GPIO driver adaption layer to Mbed OS interrupt HAL.

```

void gpio_irq_set(gpio_irq_t *obj, gpio_irq_event event, uint32_t enable)
{
    gpio_irq_event event_new;
    FunctionalState enable_new;
    switch(event) {
        case IRQ_NONE:
            //its initialize value
            event_new = (gpio_irq_event) GPIO_Event_Both;
            enable_new = DISABLE;
            break;
        case IRQ_RISE:
            event_new = (gpio_irq_event) GPIO_Event_High;
            enable_new = ENABLE;
            break;
        case IRQ_FALL:
            event_new = (gpio_irq_event) GPIO_Event_Low;
            enable_new = ENABLE;
            break;
    }
    if(obj->exti.GPIO_Event != GPIO_Event_Both)
        event_new = GPIO_Event_Both;
    //end of lut
    obj->exti.GPIO_Event = event_new;
    GPIO_EXTICfg(&obj->exti);
    //to avoid the IRQ_FALL to be mapped into IRQnone
    obj->exti.GPIO_Event = event;
    GPIO_EXTICmd(obj->exti.GPIO_Pin, (FunctionalState) enable_new);
}

```

5.2.4 Serial

This API provides to Mbed OS access to UART peripheral. This serial link consists in two unidirectional asynchronous channels (one for TX and one for RX). Since a UART connection is asynchronous, receiver and transmitter must be configured with the same speed settings.

On BlueNRG-2 there are two pins, IO_8 (USBTX) and IO_11 (USBRX), defined in the *PinNames.h* `PinName` enumeration, that are connected to the Mbed OS USB port, allowing a user to easily communicate with a PC terminal emulator through a USB connection [10].

Mbed OS UART HAL porting requires to provide the definition of the data structure `serial_s` in the header file *objects.h*. STMicroelectronics *BlueNRG1_uart.h* DK peripheral driver provides a data structure `UART_InitType` for the serial port configuration, nevertheless it is not sufficient to provide complete Mbed OS support, since the latter requires additional information regarding the IRQ index (for SoC where multiple serial interfaces are present, but

it's not BlueNRG-2 case) and TX/RX pins. The implementation of `serial_s` is provided in the following listing.

```
struct serial_s{
    UARTName uart;
    uint32_t index_irq; // Used by irq
    UART_InitType init; //bluenrg struct
    PinName pin_tx;
    PinName pin_rx;
};
```

On the user-level API side, TX and RX pin shall be defined as parameters of the class `Serial` invocation, the *baud rate* can be defined in the same declaration as optional parameter (if not declared, Mbed OS uses as baud rate the value defined by the macro `MBED_CONF_PLATFORM_STDIO_BAUD_RATE`, defined in the `mbed_config.h` file auto-generated by the Mbed CLI project exporter). UART initialization relies on a routine that provides a default peripheral configuration, in particular it performs:

1. clock UART and GPIO initialization;
2. GPIO TX and RX pin setup, raising an error if the selected pin is not connected to the UART controller;
3. 8-bit data length, 1 stop bit, no parity and no hardware flow control;
4. interrupt capability on data reception.

Parameters at point 3 can be modified from the user-level `Serial` base class access methods at a second time. In addition to that the serial initialization sets `stdio_uart_inited`, an Mbed OS internal flag, to 1, when the UART I/O is mapped on the BlueNRG-2 GPIO connected to the USB interface. This provides to the user direct access to the Mbed OS `STDIO_MESSAGES` feature: it allows the user to direct access (and call) C standard I/O function (for instance `printf`) in a transparent way, i.e. without configuring any `Serial` object. This has been, during the porting activity, a very powerful aid for debugging purpose.

The UART initialization code is reported in the following listing.

```
void serial_init(serial_t *obj, PinName tx, PinName rx){
    //GPIO and UART Peripherals clock enable
    SysCtrl_PeripheralClockCmd(CLOCK_PERIPH_UART | CLOCK_PERIPH_GPIO,
        ENABLE);
```

```

//This statement is valid for both BlueNRG1-2,
//developed under DK 3.0.0
//GPIO TX config
switch(tx){
    case USBTX:
        GPIO_InitUartTxPin8();
        break;
    case IO_5:
        GPIO_InitUartTxPin5();
        break;
    default:
        error("The selected is not UARTTX capable.
              Choose the correct pin.");
        break;
}
//GPIO RX config
switch(rx){
    case USBRX:
        GPIO_InitUartRxPin11();
        break;
    case IO_4:
        GPIO_InitUartRxPin4();
        break;
    default:
        error("The selected is not UARTRX capable.
              Choose the correct pin.");
        break;
}

/*
----- UART configuration -----
- BaudRate = 115200 baud
- Word Length = 8 Bits
- One Stop Bit
- No parity
- Hardware flow control disabled (RTS and CTS signals)
- Receive and transmit enabled
*/
UART_StructInit(&obj->serial.init);
UART_Init(&obj->serial.init);
obj->serial.uart = UART_1;
obj->serial.pin_tx = tx;
obj->serial.pin_rx = rx;
/* Interrupt as soon as data is received. */
UART_RxFifoIrqLevelConfig(FIFO_LEV_1_64);
/* Enable UART */
UART_Cmd(ENABLE);
// For stdio management in platform/mbedboard.c
//and platform/mbedretarget.cpp
if (tx==USBTX && rx==USBRX) {
    stdio_uart_init = 1;
    memcpy(&stdio_uart, obj, sizeof(serial_t));
}
}

```

5.2.5 Microsecond Ticker

The porting of this API allows Mbed OS to perform activities that requires an accurate timing (for instance `wait` or periodic routines execution from interrupt context). In addition to that it permits to the Mbed OS scheduler to correctly define its *system time*.

The Mbed OS porting guide provides a detailed description concerning the implementation of this API [9], defining some constraints for some functions to avoid *race conditions* and problems related to *thread* and *interrupt safety*.

On the hardware side, it requires a hardware counter with the following features:

- reported frequency between 250KHz and 8MHz;
- 16 bit (at least);
- ticker rolls over at $(1 \ll \text{bits})$ and continues counting starting from 0;
- increment by 1 each tick.
- interrupt generation capability on a compare event.

In addition to that, the last API requirement is to provide in this API information about the *tick frequency* and *bit number* of the hardware timer in the `ticker_info_t` by using the following function.

```
const ticker_info_t *us_ticker_get_info()
{
    static const ticker_info_t info = {
        FREQ_TICK, // 1 MHz
        NUMBITS    // 16
    };
    return &info;
}
```

On the ISR side, Mbed OS higher level API requires that, when the compare condition is verified and the IRQ is generated, the ISR calls the `us_ticker_irq_handler` Mbed OS API callback.

BlueNRG-2 has two hardware peripherals, *MFT* (*Multi Functional Timer*, described at 3.2.4) clocked at *System Clock* frequency (32 MHz), suitable for this API implementation; *MFT2B* has been chosen for this purpose.

Nevertheless BlueNRG-2 MFT2B (but also MFT1) is not fully compliant with Mbed OS requirements, since it is capable only of backwards counting: this has been handled by exploiting the *one's complement* arithmetic, providing the bitwise complement timer value for timestamp *read* operation and computing the value for interrupts *fire* at specific timestamp by difference instead of sum. For the microsecond ticker is relevant to provide full API ported, it is available in Appendix A.

5.3 Connectivity

BlueNRG-2 is a Bluetooth Low Energy application processor, capable of running a full BLE stack (host and controller on the same chip) as described in Chapter 2. On the other hand, Mbed OS is an embedded operating system oriented to IoT, for this reason it includes all the necessary features to develop ARM Cortex-M based devices in terms of connectivity; in spite of this the juxtaposition Mbed OS - BlueNRG-2 is not so immediate.

Mbed OS is designed for “pure” microcontrollers, providing detailed information about porting on this kind of devices and focusing on the most widespread microcontroller peripherals, such as UART, GPIO, SPI, I2C, DMA, ADC, etc. However, BlueNRG-2 is a different product, porting Mbed OS by means of microcontroller is feasible, but useless. On the contrary, what has been identified as “mandatory” from STMicroelectronics for BlueNRG-2 is to port the Mbed OS BLE architecture, in such a way to exploit the SoC for its original purpose.

In addition to that, at the beginning of the thesis project, there was any consolidated procedure to port Mbed OS BLE architecture directly on a BLE application processor due to the lack of an existing Mbed-enabled BLE SoC⁴: this makes the study of this part of the project the most experimental and the most interesting at all.

Mbed OS BLE stack implementation

As previously mentioned, Mbed OS provides its BLE connectivity API [8], described in Chapter 4, providing a simplified access to BLE technology at user application level.

⁴More precisely, not by using the stack configuration developed in this work.

Provide functionality to these BLE API goes through the implementation of the underlying BLE stack architecture; for this purpose two ideas have been evaluated:

1. design an adaption layer between Mbed OS BLE API and STMicroelectronics BlueNRG stack library;
2. exploit the ARM Cordio BLE host architecture and design the adaption layer at HCI level.

Advantage of the first approach has been identified with *flash* occupation: BlueNRG stack would be highly optimized in terms of code size, however this approach implies a continuous support from STMicroelectronics for the integration of any change from ARM to its BLE API in the adaption layer.

In addition to that, a comparison between the two approaches has been directly discussed with ARM Mbed OS team: ARM has provided some information (details are classified) about an internal test similar to this case, showing a comparison between solution 1 and 2 and quantifying the flash occupation difference in less than 5 kB (more in solution 2) and furthermore showing an unexpected SRAM save of 4 kB (less in solution 2, due to the deep integration of ARM Cordio BLE Host in Mbed OS that allows heavy runtime optimization).

For this reasons solution 2 has been identified as the most convenient, and one has to notice also another important advantage for the approach 2: the adaption layer on the HCI layer would require less support and would be valid for longer time, since it leans on an interface that is standardized by Bluetooth SIG [21], allowing this solution to work mainly in compliance with Bluetooth standard and providing the highest level of abstraction regarding both host and controller implementation.

5.3.1 BLE API

Once opted for a BLE stack porting over HCI layer, another solution space analysis has been performed (even with the precious support from ARM) to determine the most convenient implementation, resulting in these two options:

1. thin HCI porting design, using ARM Cordio HCI commands on the HCI host-side to

wrap ST controller procedure, and ST events callback procedures to wrap ARM Cordio Host event handlers (HCI controller side);

2. design a full transport based HCI layer between ARM Cordio host and ST controller.

First option implementation relies on the *exactLE* thin HCI of ARM Cordio host; as the same as the preface of this section this first option presents more disadvantages than benefits. *ExactLE* implementation requires a controller implementation-specific synchronization task to handle the host-controller communication in a correct way and to provide the correct link-layer FSM timing and processing. This solution is good, however is again too sensitive to the host implementation, so it has been discarded.

Therefore the second option has been deeply investigated and chosen. The last issue to solve is related to the absence of a real serial transport: it has been overcome by exploiting *DTM* (*Direct Test Mode*). The latter is a Bluetooth SIG standard, used for automated conformance testing and allowing direct access to LL and PHY features of the *DUT* (*Device Under Test*) [21] by enabling the exchange of packets compliant to the HCI standard format (described in Chapter 2, Section 2.3) over a serial interface, like UART or SPI. The mechanism exploited by DTM in this project is only for parsing (there is no serial interface involved between host and controller) and is presented in Figure 5.7.

Once defined the overall porting architecture, the next step is to identify ARM Cordio host requirements and implement them. HCI driver shall be split in two entities.

HCI Transport - *TransportDriver*

This driver inherits from the ARM Cordio *CordioHCITransportDriver* class. It must implement:

- `initialize` and `terminate` methods. This porting on SoC design has not a real transport, so any initialization/termination is required, nevertheless transport initialization is exploited for the periodic Link Layer FSM tick (calling `BTLE_StackTick`) invocation.
- `write` method sends data to the BlueNRG-2 controller. It performs a parse of the packet coming from the ARM Cordio host and a linear search in a *command table* based on the

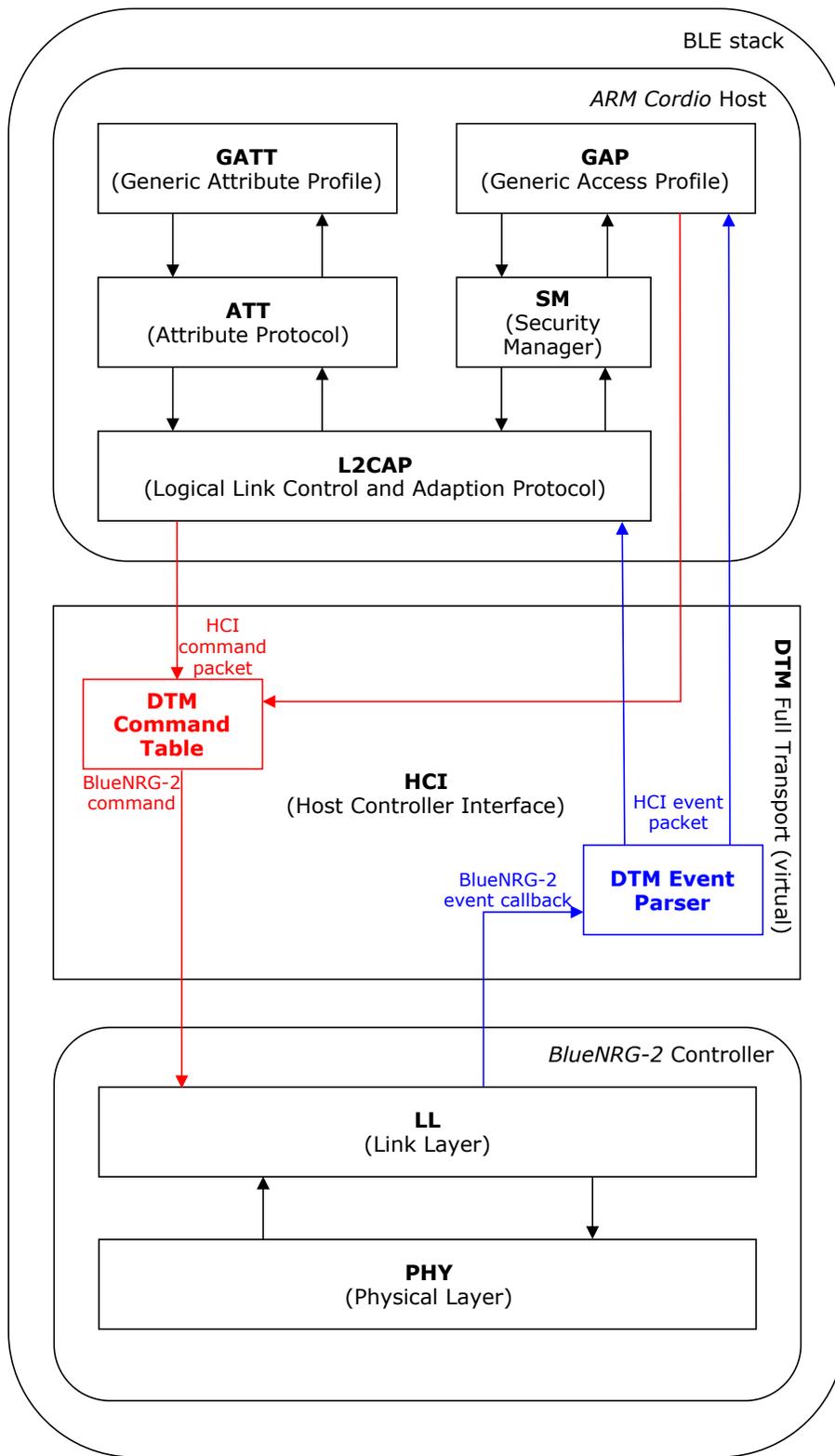


Figure 5.7: BLE stack - DTM adaption layer (at HCI)

HCI command opcodes. Once resolved the opcode, the HCI command is executed by pushing the HCI routine address in a *function pointer*, the rest of the host HCI packet is parsed into PACKED⁵ structures organized according to the BlueNRG-2 required parameters for the command to be executed. The *command table* is provided in Appendix A, while the data structure implementing the *function pointer* mechanism is presented in the next listing.

- `on_data_received` is a method whose purpose is to forward a packet coming from the controller to the host, it is also “externalized” to be available to the C code of the DTM parser (with name `rcv_callback`).

```

/**** IN DTM HEADER ****/
typedef uint16_t (*hci_command_process_and_response_type)(uint8_t *
    buffer_in, uint16_t buffer_in_length, uint8_t *buffer_out, uint16_t
    buffer_out_max_length) ;
typedef struct hci_command_table_type_s {
    uint16_t opcode;
    hci_command_process_and_response_type execute;
} hci_command_table_type;

```

An example of command parsing through the DTM adaption layer is provided in the following listing. Concerning BLE events, the parsing procedure is the opposite.

```

typedef PACKED(struct) hci_ble_set_random_address_cp0_s {
    uint8_t Random_Address[6];
} hci_ble_set_random_address_cp0;

uint16_t hci_ble_set_random_address_process(uint8_t *buffer_in, uint16_t
    buffer_in_length, uint8_t *buffer_out, uint16_t buffer_out_max_length)
{
    /* Input params */
    hci_ble_set_random_address_cp0 *cp0 = (hci_ble_set_random_address_cp0 *) (
        buffer_in + 1 + (0));

    int output_size = 1;
    /* Output params */
    uint8_t *status = (uint8_t *) (buffer_out + 6);

    if (buffer_out_max_length < (1 + 6)) { return 0; }
    *status = hci_ble_set_random_address(cp0->Random_Address /* 6 */);
    buffer_out[0] = 0x04;
    buffer_out[1] = 0x0E;
    buffer_out[2] = output_size + 3;
    buffer_out[3] = 0x01;
}

```

⁵PACKED structures are data structures compiled without padding (to save memory).

```
buffer_out[4] = 0x05;
buffer_out[5] = 0x20;
return (output_size + 6);
}
```

HCI Driver - *HCIDriver*

This class inherits from the base class `CordioHCIDriver`. It shall provide:

- `do_initialize` and `do_terminate`, in this BlueNRG-2 driver they are not implemented since controller initialization is performed at startup;
- `get_buffer_pool_description` returns a 1430 bytes buffer for the ARM Cordio host to use;
- HCI reset sequence is started by `start_reset_sequence`, then the handler `handle_reset_sequence` is performed until the reset procedure end is signaled by the signal `signal_reset_sequence_done`.

More in detail, the `handle_reset_sequence` provides initialization for BlueNRG-2 controller parameters, i.e. the *LE event mask*, that is a bit mask which events the controller signals to the host and those to flush. It provides also initialization for ARM Cordio runtime parameters, such as *Bluetooth address*, *controller buffer size*, *supported state*, *whitelist size*, *LE (Length Extension) features supported*, *resolving list size* and *maximum data length* [21] [6].

`TransportDriver` and `HCIDriver` are provided in the file `BlueNrgHCIDriver.cpp`, while DTM adaption layer, due to huge amount of code, is not directly reported (as previously said only the DTM command table is) in this document and provided at

```
https://github.com/ntoni92/mbed-os-BlueNRG2/blob/master/  
targets/TARGET\_STMBLUE/DTM/src/DTM\_cmd\_db.c
```

5.4 Low power mode

As last step of the described porting activity there is the implementation of mechanisms that allows BlueNRG-2 to save power in idle state through Mbed OS API. As explained in Chapter 1, power saving is mandatory to make a BLE product attractive in an aggressive market context.

BlueNRG-2 natively provides three different low power modes [27]:

- *CPU-Halt mode*
 - The least aggressive low power modes. CPU is stopped, but all device peripherals are active and able to wake the CPU through interrupt.
- *Sleep mode*
 - CPU and peripherals stopped, with the exception of the low speed oscillator and external wake up sources (wake up *virtual timers* and GPIO pin 9, 10, 11, 12, 13).
 - When wake up is triggered BlueNRG-2 reverts to its previous running mode after the high speed oscillator stabilization.
- *Standby mode*
 - CPU and all peripherals disabled, wake up sources are only GPIO pin 9, 10, 11, 12, 13.
 - When wake up is triggered BlueNRG-2 reverts to its previous running mode after the high speed oscillator stabilization.

On the Mbed OS side, the low power features are enabled by adding the “SLEEP” label in the “`device_has`” option of target’s section in the `targets.json` file, described in Section 5.1.1 [9]. After that, low power feature are enabled by providing a target-specific implementation of the API `mbed-os/hal/sleep_api.h`. The latter defines two low power modes, `hal_sleep` and `hal_deepsleep` whose study is provided in Section 5.4.1 and 5.4.2 respectively and implementation is provided in Appendix A.

5.4.1 Sleep

In Mbed OS *sleep* mode the system core clock is disabled and RAM is put into data retention state. In addition to that:

- any peripheral capable of generating interrupts must wake up the device;

- the device shall wake up within $10\mu s$.

In addition to that, BlueNRG-2 controller requires to be queried and responds itself, according to the state of the LL FSM, the sleep mode depth it can achieve.

Taking into account these design information, the only BlueNRG-2 sleep mode compatible with Mbed OS is the *CPU-Halt mode*. For this reason the *sleep manager* is designed to query the controller stack and if this check returns a deep sleep mode equal or greater than *CPU-Halt* the CPU is halted (if this check fails the controller is not ready to sleep and thus the whole system can't sleep).

If the preliminary sleep check is passed there is no need to save peripheral registers (*CPU-Halt mode* maintains configurations for all peripherals). The only exception is the *watchdog*: before sleep its control register value is saved (it will be restored right after wake up) and the peripheral is disabled, to avoid a reset while CPU is halted.

5.4.2 Deep Sleep

In Mbed OS *deep sleep* mode the system core clock is disabled and RAM is put into data retention state. In addition to that:

- GPIO, RTC (Real Time Clock) or Low Power Ticker must wake up the CPU;
- the device shall wake up within $10ms$.

The same BlueNRG-2 stack controller query mechanism described for *sleep* mode in Section 5.4.1 is exploited to validate deep sleep request, but with respect to the latter, for this mode a return value higher than *CPU-Halt* is required.

However BlueNRG-2 hardware architecture presents a huge limitation in implementing Mbed OS deep sleep mode: there is neither RTC nor hardware timer connected to the wake up controller⁶. More precisely, there are two timers connected to the wake up controller, nevertheless they are used from STMicroelectronics stack library, they are not directly accessible as hardware peripherals but only through an interface provided in the header *bluenrg1_stack.h*

⁶An hardware timer connected to the wake up controller is required for implementing the Low Power Ticker, that is in turn a wake up from deep sleep source.

(called *Virtual Timers*), and so quite useless for implementing an Mbed OS Low Power Ticker (at least in a traditional way).

The only available wake up peripheral is GPIO. This represents an enormous limitation in using deep sleep mode, since there is no way to schedule a wake up event at a certain timestamp; this confines deep sleep to be used only in applications where wake up is required to occur in a totally non-deterministic way and where there is an external access to the system (resulting in a GPIO interrupt). One can see this deep sleep implementation only as a porting exercise and something to evaluate benefits of BlueNRG-2 advanced sleep modes.

Concerning the implementation, when BlueNRG-2 goes in deep sleep, the peripheral registers values are lost. For this reason there are more operation to do before going into deep sleep:

- save peripheral registers values into SRAM;
- save status into SRAM and disable watchdog;
- save application context (processor registers contents into SRAM);
- save MSP, PSP (Main and Process Stack Pointers) and PC (Program Counter) values into SRAM.

This setup completes the preliminary operations before going in deep sleep, and it has been the last step in this experimental porting of ARM Mbed OS on BlueNRG-2 SoC.

5.5 Results and further developments

In this section there are presented some significant results concerning the utilization of Mbed OS operating system on BlueNRG-2. First of all there are reported information about compiled binary size and memory (Flash memory and SRAM occupation), after that there is a functional verification of an application (designed by ARM Mbed team and suggested as BLE “Hello World”) and at the end there are some measurements concerning power consumption.

HRM - Heart Rate Monitor

The presented HRM application has been taken as reference application while developing the BlueNRG-2 Mbed OS porting, due to the fact that HRM profile requires the functionality of almost all the BLE features (advertising, connection, service discovery, notification) and so it is a complete test for all the BLE stack layers. In addition to that, this approach has been discussed and used in accordance with an internal ST team (located in France) working on the Mbed OS porting on another BLE device, in order to exchange useful feedback about reciprocal issues and progresses.

HRM has been useful also for another reason: it has been compiled and executed on an already working and consolidated STMicroelectronics configuration, a dual-chip over HCI (NUCLEO-F401RE + X-NUCLEO-IDB05A1 BlueNRG-MS network coprocessor expansion, already mentioned in Chapter 1). Since the HCI is implemented through SPI, it has been possible to extract the HCI trace resulting by correct execution of HRM, and use it as benchmark and debug during BlueNRG-2 porting activity.

HRM code is reported in Appendix A.

5.5.1 Code size

The following report tables refers to HRM code compiled with:

- GCC_ARM toolchain 6.3.1 - 20170620 (release) [ARM/embedded-6-branch revision 249437]
- Mbed OS 5.11 [ARMmbed/mbed-os master branch, commit ID df9ac85cbb38d8f06380b339398e73f968e3ba0a]

Considering that BlueNRG-2 controller is configured at its minimum (single link mode) one can immediately notice that Mbed OS, also configured at its minimum (no RTOS), keeps BlueNRG-2 memory to its limit: the free space for application is very tiny.

In *debug mode* moreover the *printf* (more in general the *STDIO*) cannot be called and has to be disabled, otherwise the final code size exceeds Flash capacity.

In *develop mode* there is an improvement in terms of occupation, however in this configuration debug symbols are removed from the executable (it means no live variables information

and no stepping mode execution) resulting in an increased difficulty in application develop.

Optimization level	.text [B]	.data [B]	.bss [B]
-Os -g1 (optimize size)	213032	10600	7700

Table 5.1: Mbed OS HRM compiler report (develop configuration)

Optimization level	.text [B]	.data [B]	.bss [B]
-Og -g3 (optimize debug)	241600	10600	7828

Table 5.2: Mbed OS HRM compiler report (debug configuration)

	Flash (.text + .data) [B]	SRAM (.data + .bss) [B]
<i>HRM (develop profile)</i>	223632	18300
<i>HRM (debug profile)</i>	251600	18428

Table 5.3: HRM code size and SRAM occupation

A possible trade off could be to compile Mbed OS code with optimize-size options and the user code with debug symbols enabled. A more aggressive solution could be (as ARM does with Mbed OS 2 and OS 5 in its *online compiler*) to compile and provide Mbed OS 5 as static binary library.

5.5.2 Power performances

The last Mbed HRM testing on BlueNRG-2 concerns the power consumption of the system. Considering the thesis work purpose, it is not interesting to characterize precisely the power behavior of BlueNRG-2, but rather one could consider this part as an additional Mbed OS HRM functional verification on BlueNRG-2 (an more in general provide food for though regarding what said on Bluetooth Low Energy technology in Chapter 1).

The measurement setup is reported in Figure 5.8 and consists of:

- X-NUCLEO-LPM01A board for Nucleo expansion consumption measurement [33] (the blue one);

- a X-NUCLEO expansion shield (the green top one) embedding a complete BlueNRG-2 *System-on-Module*, with RF frontend and ceramic antenna (the one in the metallic case marked with “ES”).

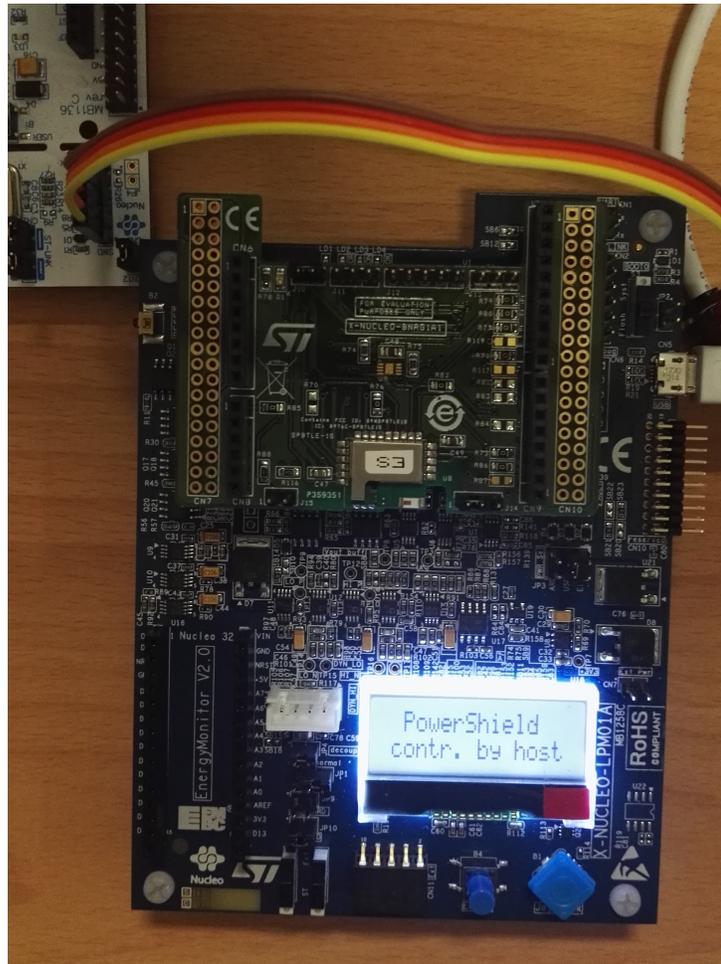


Figure 5.8: STMicroelectronics “PowerShield” board and target BlueNRG-2 module expansion power measurement setup

The following is performed on this setup, instead of using the STEVAL-IDB008V2 board, because it has only the BlueNRG-2 SoC (and not the accessory STEVAL ICs like, for instance, the UART to USB interface).

Since the interest is in a qualitative power behavior, details about the measure itself are omitted. Moreover the BlueNRG-2 *System-On-Module* is not commercial at the time when this study has been carried on, additional information (like schematics and meaning of connectors)

cannot be provided since they are STMicroelectronics-restricted. Finally, one has to notice that the proposed Mbed OS BLE API implementation on BlueNRG-2 does not support sleep modes because of the lack of a wake up source (discussed at 5.3.1).

A dynamic current measure plot, with a sampling frequency of $10kHz$, is provided in Figure 5.9, generated by a software GUI (*STM32CubeMonitor-Power* [34]) controlling the X-NUCLEO-LPM01A through UART.



Figure 5.9: ARM Mbed OS HRM current trend during execution

Figure 5.9 shows BlueNRG-2 startup (POR) after about $2s$, then it starts advertising with $1s$ interval (advertising events corresponds to $14mA$ peaks). At about $18s$ there is a connection from a central device and a service discovery request, the activity increases and average current rises up, the central device enables notifications on the BlueNRG-2 HRM peripheral and at about $23s$ it starts to notify the heart rate value (performed each second, but covered from other device activity). At the end, after $35s$, central device disconnects from HRM and it starts advertising again.

During connection anything can be improved, however, as proposal of future search (since at the time of this porting design is not feasible) it could be ideal to put the device in low power mode during idle intervals occurring between advertising events (the latter proposed

implementation puts BlueNRG-2 in *sleep mode* -5.4.1 - during these idle intervals). A simple example in next section shows the potentially power saving with this strategy.

Low Power modes (further development)

Since BLE API is not compatible with *deep sleep mode*, this simple test has been designed to show the current consumption trend of an application capable to go into the maximum achievable power saving mode.

It is an application that blinks a LED every 2s. This first plot (Figure 5.10) represents the case in which BlueNRG-2 is fully active while the LED is on (the LED absorbed current contributes in this phase) and is in *sleep mode* (*CPU-Halt*) when the LED is off.



Figure 5.10: Sample LED Blink application with *sleep mode*

It presents a $2.5mA$ current consumption in idle phase, the same as the HRM example idle phase between advertising events.

In the next example, shown in Figure 5.11, idle phase (LED off) is spent in *deep sleep mode* with a BlueNRG-2 *Virtual Timer* sets to wake up the device and turn on the LED every 2s. The code implementation of this last example is not reported, since it is coded in an hybrid Mbed OS and bare direct calls to the STMicroelectronics BlueNRG-2 controller HAL, so not

compliant with Mbed OS guidelines.



Figure 5.11: Sample LED Blink application with *deep sleep mode*

One can notice that, out of idle phase, the consumption has not a regular trend. This is due to the fact that the wake up from deep sleep requires more operations with respect to sleep case, involving oscillator re-calibration and a *context restore*. But above all, the most noticeable information is the current consumption in idle *deep sleep mode*: it is less than $5\mu A$.

5.5.3 Final considerations

Strictly concerning BlueNRG-2, collected data demonstrate not only that the coexistence of ARM Mbed OS (with his BLE host stack ARM Cordio) and BlueNRG-2 HAL controller architecture is feasible, but that it makes sense to support this porting to reach the Mbed-enabled status. The only feature completely cut off in this development is *OTA - Over The Air* firmware update and programming: it is one of the most interesting features of BLE, but it requires a huge amount of free memory to be implemented (this is not the case of this porting). Nevertheless, for those designs not requiring OTA, Mbed OS on BlueNRG-2 could be an interesting solution. For this reason a merge pull-request has been opened with ARM, but there are some features that ARM should introduce to support Cordio-based SoC design. The most important

among the others is the capability of processing also the controller FSM at the same time (not provided because, up to now, ARM Cordio host has been intended to work in a dual-chip configuration when host and controller run on different timing domains and communicate over an asynchronous serial interface), so the possibility to register a callback that runs in the same host tick context has been required to ARM. This would allow to hide the stack synchronization mechanism at the user level BLE API.

The pull-request is available at the following URL:

<https://github.com/ARMmbed/mbed-os/pull/9491>.

More generally, this porting activity has been started with experimental and research purposes. First and foremost this experience shows that, in embedded system design, good software design strategies become every day more and more important and by now it is mandatory to match a good electronic architecture to an efficient firmware construction. Let's concentrate on power efficiency discussed in the last activity of this chapter: consumption has a strong cross-correlation with the code running on it, power has been reduced of many order of magnitude only by changing firmware design strategy. Translating these concepts into real world, BLE technology is thought for ultra low power applications, mainly powered by coin-cell batteries: it means that a firmware has the "authority" to decide if battery charge lasts for six days or for six months. Whether the first option occurs while the second is expected brings to destroy all the effort done for an excellent electronic/microelectronic design and above all for the improvement of fabrication and miniaturization device technology, which in the last few years is the primary R&D item of expenditure for a silicon foundry because of the increasing cost-per-transistor trend.

Besides, experimenting this porting opens to the possibility in terms of firmware reuse: in the past a firmware was more "bare metal" than today and thus strictly related to its hardware and a change in the hardware architecture of the same product (for instance a new version of the product) usually required a new firmware. For this reason the concept of HAL - Hardware Abstraction Layer has been introduced: a good co-design of the HAL-based firmware avoids a new bare-metal design of the latter and allows to perform only a less invasive code refactoring (for instance, on STM32 family microcontrollers, CubeMX HAL permits to easily migrate the

code between different Cortex-M microcontrollers). Mbed OS is breaking again this paradigm allowing to run the same code on different architectures without changing any code line: for instance it is possible the same identical HRM code either on a SoC (as BlueNRG-2 application processor is) or on a dual-chip over HCI architecture (for instance NUCLEO + BlueNRG-MS BLE co-processor).

For the discussed reasons STMicroelectronics, an excellence in semiconductor industry, has began a reinforcing policy in its application design sector and to increase its asset also in software/firmware solutions quality too. Moreover it has decided to reinforce its partnership with ARM Mbed OS by starting to develop the support on other BLE devices. STMicroelectronics considers Mbed OS support an important asset in its portfolio, and even if this project has not the ambition to say if it absolutely correct or not, it adds a good reason to think that is a reasonable choice.

Appendix A - Source Code

Microsecond Ticker - us_ticker_api.c

```
#include "us_ticker_api.h"
#include "PeripheralNames.h"
#include "hal_types.h"
#include "BlueNRG1_mft.h"
#include "BlueNRG1_sysCtrl.h"
#include "misc.h"

#define FREQ_TICK 1000000
#define NUMBITS 16

enum InitStatus{
    noinit = 0,
    init = 1,
}status;

const ticker_info_t *us_ticker_get_info()
{
    static const ticker_info_t info = {
        FREQ_TICK,
        NUMBITS
    };
    return &info;
}

void us_ticker_isr(void){
    //clear interrupt flag and call the us ticker handler
    MFT_ClearIT(MFT2, MFT_IT_TND);
    us_ticker_irq_handler();
}

void us_ticker_init(void){
    if(status == init) return;

    status = init;

    MFT_InitType timer_init;
    NVIC_InitType NVIC_InitStructure;
```

```

//tickcount = 0;

NVIC_SetVector(MFT2B_IRQn, (uint32_t)&us_ticker_isr);
SysCtrl_PeripheralClockCmd(CLOCK_PERIPH_MTFX2, ENABLE);
MFT_StructInit(&timer_init);

timer_init.MFT_Mode = MFT_MODE_3;
timer_init.MFT_Prescaler = SYST_CLOCK/FREQ_TICK - 1;
timer_init.MFT_Clock1 = MFT_NO_CLK;
timer_init.MFT_Clock2 = MFT_PRESCALED_CLK;
timer_init.MFT_CRA = 0x0000;
timer_init.MFT_CRB = 0xFFFF;

MFT_Init(MFT2, &timer_init);

//Set counter for timer2
MFT_SetCounter2(MFT2, 0xFFFF);

//Enable MFT2B Interrupt
NVIC_InitStructure.NVIC_IRQChannel = MFT2B_IRQn;
NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority =
    HIGH_PRIORITY;
NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
NVIC_Init(&NVIC_InitStructure);

//Enable the MFT2 interrupt
MFT_EnableIT(MFT2, MFT_IT_TND, ENABLE);

//Start MTF2B
MFT_Cmd(MFT2, ENABLE);
}

void us_ticker_free(){
    if(status == init){
        MFT_Cmd(MFT2, DISABLE); //Disable MTF2B
        MFT_DeInit(MFT2); //Deinit peripheral
        NVIC_DisableIRQ(MFT2B_IRQn); //Disable IRQ
        SysCtrl_PeripheralClockCmd(CLOCK_PERIPH_MTFX2, DISABLE);
        //Disable Clock Gate peripheral
        status = noint;
    }
}

uint32_t us_ticker_read(){
    if(!status)
        us_ticker_init();
    //bitwise complement, MFT2 is a backward counter
    return ~MFT_GetCounter2(MFT2);
}

/* We get here absolute interrupt time which takes into account counter
    overflow.
    * Since we use additional count-DOWN timer to generate interrupt we need
    to calculate load value based on timestamp.

```

```

* If the timestamp value is higher than the actual count, then the
interrupt will be triggered after "timestamp - actualticks".
* If the timestamp value is lower than the actual count, then the
interrupt is triggered at the next count cycle after the overflow,
i.e. "timestamp + 1 x overflowcount (0xFFFF) - actualcount".
*/
void us_ticker_set_interrupt(timestamp_t timestamp){
    if(!status)
        us_ticker_init();

    uint16_t delta_ticks = (uint16_t)(timestamp - us_ticker_read());
    //back counter
    if (delta_ticks == 0) {
        // The requested delay is less than the minimum resolution of this
        //counter.
        delta_ticks = 1;
    }
    //Clock Gate peripheral, safe since IRQ is disabled by high level API
    SysCtrl_PeripheralClockCmd(CLOCK_PERIPH_MTFX2, DISABLE);
    MFT2->TNCRB = delta_ticks;
    MFT_EnableIT(MFT2, MFT_IT_TND, ENABLE);
    //Disable Clock Gate peripheral
    SysCtrl_PeripheralClockCmd(CLOCK_PERIPH_MTFX2, ENABLE);
}

/* NOTE: following API implementations must be called with interrupts
disabled! */
void us_ticker_disable_interrupt(void){
    if(!status)
        us_ticker_init();
    MFT_EnableIT(MFT2, MFT_IT_TND, DISABLE);
}

void us_ticker_clear_interrupt(void){
    MFT_ClearIT(MFT2, MFT_IT_TND);
}

void us_ticker_fire_interrupt(void){
    NVIC_SetPendingIRQ(MFT2B_IRQn);
}

```

DTM Command Parsing - Command Table

```

/**** IN DTM IMPLEMENTATION ****/
const hci_command_table_type hci_command_table[57] = {
    /* hcidisconnect */
    {0x0406, hci_disconnect_process},
    /* hcireadremoteversioninformation */
    {0x041d, hci_read_remote_version_information_process},

```

```

/* hciseteventmask */
{0x0c01, hci_set_event_mask_process},
/* hcireset */
{0x0c03, hci_reset_process},
/* hcireadtransmitpowerlevel */
{0x0c2d, hci_read_transmit_power_level_process},
/* hcireadlocalversioninformation */
{0x1001, hci_read_local_version_information_process},
/* hcireadlocalsupportedcommands */
{0x1002, hci_read_local_supported_commands_process},
/* hcireadlocalsupportedfeatures */
{0x1003, hci_read_local_supported_features_process},
/* hcireadbdbaddr */
{0x1009, hci_read_bd_addr_process},
/* hcireadrssi */
{0x1405, hci_read_rssi_process},
/* hcileseteventmask */
{0x2001, hci_le_set_event_mask_process},
/* hcilereadbufferize */
{0x2002, hci_le_read_buffer_size_process},
/* hcilereadlocalsupportedfeatures */
{0x2003, hci_le_read_local_supported_features_process},
/* hcilesetrandomaddress */
{0x2005, hci_le_set_random_address_process},
/* hcilesetadvertisingparameters */
{0x2006, hci_le_set_advertising_parameters_process},
/* hcilereadadvertisingchanneltxpower */
{0x2007, hci_le_read_advertising_channel_tx_power_process},
/* hcilesetadvertisingdata */
{0x2008, hci_le_set_advertising_data_process},
/* hcilesetscanrespondedata */
{0x2009, hci_le_set_scan_response_data_process},
/* hcilesetadvertiseenable */
{0x200a, hci_le_set_advertise_enable_process},
/* hcilesetscanparameters */
{0x200b, hci_le_set_scan_parameters_process},
/* hcilesetscanenable */
{0x200c, hci_le_set_scan_enable_process},
/* hcilecreateconnection */
{0x200d, hci_le_create_connection_process},
/* hcilecreateconnectioncancel */
{0x200e, hci_le_create_connection_cancel_process},
/* hcilereadwhitelistize */
{0x200f, hci_le_read_white_list_size_process},
/* hcileclearwhitelist */
{0x2010, hci_le_clear_white_list_process},
/* hcileadddevicetowhitelist */
{0x2011, hci_le_add_device_to_white_list_process},
/* hcileremovedevicefromwhitelist */
{0x2012, hci_le_remove_device_from_white_list_process},
/* hcileconnectionupdate */
{0x2013, hci_le_connection_update_process},
/* hcilesethostchannelclassification */
{0x2014, hci_le_set_host_channel_classification_process},

```

```

/* hcilereadchannelmap */
{0x2015, hci.le.read.channel.map.process},
/* hcilereadremoteusedfeatures */
{0x2016, hci.le.read.remote.used.features.process},
/* hcileencrypt */
{0x2017, hci.le.encrypt.process},
/* hcilerand */
{0x2018, hci.le.rand.process},
/* hcilestartencryption */
{0x2019, hci.le.start.encryption.process},
/* hcilelongtermkeyrequestreply */
{0x201a, hci.le.long.term.key.request.reply.process},
/* hcilelongtermkeyrequestednegativereply */
{0x201b, hci.le.long.term.key.requested.negative.reply.process},
/* hcilereadsupportedstates */
{0x201c, hci.le.read.supported.states.process},
/* hcilereceivertest */
{0x201d, hci.le.receiver.test.process},
/* hciletransmittertest */
{0x201e, hci.le.transmitter.test.process},
/* hciletetestend */
{0x201f, hci.le.test.end.process},
/* hcilesetdatalength */
{0x2022, hci.le.set.data.length.process},
/* hcilereadsuggesteddefaultdatalength */
{0x2023, hci.le.read.suggested.default.data.length.process},
/* hcilewritesuggesteddefaultdatalength */
{0x2024, hci.le.write.suggested.default.data.length.process},
/* hcilereadlocalp256publickey */
{0x2025, hci.le.read.local.p256.public.key.process},
/* hcilegeneratedhkey */
{0x2026, hci.le.generate.dhkey.process},
/* hcileadddevicetoresolvinglist */
{0x2027, hci.le.add.device.to.resolving.list.process},
/* hcileremovedevicefromresolvinglist */
{0x2028, hci.le.remove.device.from.resolving.list.process},
/* hcileclearresolvinglist */
{0x2029, hci.le.clear.resolving.list.process},
/* hcilereadresolvinglistsize */
{0x202a, hci.le.read.resolving.list.size.process},
/* hcilereadpeerresolvableaddress */
{0x202b, hci.le.read.peer.resolvable.address.process},
/* hcilereadlocalresolvableaddress */
{0x202c, hci.le.read.local.resolvable.address.process},
/* hcilesetaddressresolutionenable */
{0x202d, hci.le.set.address.resolution.enable.process},
/* hcilesetresolvableprivateaddresstimeout */
{0x202e, hci.le.set.resolvable.private.address.timeout.process},
/* hcilereadmaximumdatalength */
{0x202f, hci.le.read.maximum.data.length.process},
/* acihalwriteconfigdata */
{0xfc0c, aci.hal.write.config.data.process},
/* acihalreadconfigdata */
{0xfc0d, aci.hal.read.config.data.process}

```

```
};
```

HCI Driver - BlueNrgHCIDriver.cpp

```
/**
*****
* @file    BlueNrgHCIDriver.c
* @author  Antonio O.
* @date    21 Dec. 2018
*****
*/
#include <mbed.h>
#include "CordioHCIDriver.h"
#include "CordioHCITransportDriver.h"
#include "hci_api.h"
#include "hci_cmd.h"
#include "hci_core.h"
#include "dm_api.h"
#include "bstream.h"
//
#include "DTM_boot.h"
#include "DTM_cmd_db.h"
#include "osal.h"
#include "bluenrg1_api.h"
#include "bluenrg1_stack.h"
#include "hci_defs.h"
#include "BLEInstanceBase.h"

//STACKTICK CODE is used for emulating BTLE StackTick on an HCI vendor
//specific command.
//This value is not used and can be assigned to this functionality.
//The wrapper for BTLE StackTick is provided in DTM command db header
#define STACKTICK_CODE 0xFCFF
#define TICK_US        TICK_MS*1000

extern "C" void rcv_callback(uint8_t *data, uint16_t len){
    ble::vendor::cordio::CordioHCITransportDriver::on_data_received(
        data, len);
}

#define HCI_RESET_RAND_CNT        4
#define LL_WITHOUT_HOST_OFFSET 0x2C

namespace ble {
namespace vendor {
namespace bluenrg {

/**
* BlueNRG HCI driver implementation.
* @see cordio::CordioHCIDriver
*/
```

```

*/
class HCIDriver : public cordio::CordioHCIDriver
{
public:
    /**
     * Construction of the BlueNRG HCIDriver.
     * @param transport: Transport of the HCI commands.
     */
    HCIDriver(cordio::CordioHCITransportDriver& transport_driver) :
        cordio::CordioHCIDriver(transport_driver) { }

    virtual ~HCIDriver() {};
    /**
     * @see CordioHCIDriver::doinitialize
     */
    virtual void do_initialize() {

    }

    /**
     * @see CordioHCIDriver::startresetsequence
     */
    virtual void start_reset_sequence() {
        /* send an HCI Reset command to start the sequence */
        HciResetCmd();
    }

    /**
     * @see CordioHCIDriver::doterminate
     */
    virtual void do_terminate() {

    }

    /**
     * @see CordioHCIDriver::handlerresetsequence
     */
    virtual void handle_reset_sequence(uint8_t *pMsg) {
        // only accept command complete event:
        if (*pMsg != HCI_CMD_CMPL_EVT) {
            return;
        }

        uint8_t Value_LL = 0x01;
        uint16_t opcode;
        // static uint8_t randCnt;

        /* parse parameters */
        pMsg += HCI_EVT_HDR_LEN;
        pMsg++;
        BSTREAM_TO_UINT16(opcode, pMsg); // copy opcode
        pMsg++;
        /* skip status */
    }
};

```

```

    /* decode opcode */
    switch (opcode)
    {
        case HCI_OPCODE_RESET:
            /* initialize rand command count */
            randCnt = 0;

// manually initialization of random address
//(because there is no GAP init)

//8 bytes allocated because of ST random generation
            uint8_t Random_Address[8];
            hci_le_rand(Random_Address);
//bitwise or for 2 MSB (required by core 5.0 for Random Static Address)
            Random_Address[5] = Random_Address[5] | 0
                xC0;

//Set the controller in Link Layer Only mode
            aci_hal_write_config_data(LL_WITHOUT_HOST_OFFSET, 1, &
                Value_LL);
            hci_le_set_random_address(Random_Address);

//DO NOT SET ANY EVENT MASK, BY DEFAULT ALL EVENTS ENABLED

// Ask the Bluetooth address of the controller.
            hci_read_bd_addr(hciCoreCb.bdAddr);

// Read the size of the buffer of the controller and store the buffer
//parameters in the stack Cordio runtime parameters.
            hci_le_read_buffer_size(&hciCoreCb.bufSize, &hciCoreCb.
                numBufs);
/* initialize ACL buffer accounting */
            hciCoreCb.availBufs = hciCoreCb.numBufs;

// Read the states and state combinations supported by the link layer
// of the controller and store supported state and combination in the
//runtime parameters of the stack Cordio.
            hci_le_read_supported_states(hciCoreCb.leStates);

// Read the total of whitelist entries that can be stored in the
//controller and store the number of whitelist entries in the stack
//Cordio runtime parameters.
            hci_le_read_white_list_size(&hciCoreCb.whiteListSize);

// Read the LE features supported by the controller and store the set
// of LE features supported by the controller in the Cordio Stack
//runtime parameters.
            uint8_t features[2];
//it is a 64 bit number, but only 16 MSB are significant (future use)
            hci_le_read_local_supported_features(features);
            hciCoreCb.leSupFeat = features[1] << 8 | features[0];

//reset sequence could terminate here depending on controller configuration

```

```

        hciCoreReadResolvingListSize();

        break;

        case HCI_OPCODE_LE_READ_RES_LIST_SIZE:
// Store the number of address translation entries in the stack
// runtime parameter.
            BSTREAM_TO_UINT8(hciCoreCb.resListSize, pMsg);

// Read the Controller maximum supported payload octets and packet
// duration times for transmission and reception.
            hciCoreReadMaxDataLen();
            break;

        case HCI_OPCODE_LE_READ_MAX_DATA_LEN:
        {
// Store payload definition in the runtime stack parameters.
            uint16_t maxTxOctets;
            uint16_t maxTxTime;

            BSTREAM_TO_UINT16(maxTxOctets, pMsg);
            BSTREAM_TO_UINT16(maxTxTime, pMsg);

/* use Controller's maximum supported payload octets and packet
* duration times for transmission as Host's suggested values for
* maximum transmission number of payload octets and maximum packet
* transmission time for new connections.
*/
            HciLeWriteDefDataLen(maxTxOctets, maxTxTime);
        }
        break;

        case HCI_OPCODE_LE_WRITE_DEF_DATA_LEN:
            if (hciCoreCb.extResetSeq)
            {
/* send first extended command */
                (*hciCoreCb.extResetSeq)(pMsg, opcode);
            }
            else
            {
/* initialize extended parameters */
                hciCoreCb.maxAdvDataLen = 0;
                hciCoreCb.numSupAdvSets = 0;
                hciCoreCb.perAdvListSize = 0;

/* send next command in sequence */
                HciLeRandCmd();
            }
            break;

        case HCI_OPCODE_LE_READ_MAX_ADV_DATA_LEN:
        case HCI_OPCODE_LE_READ_NUM_SUP_ADV_SETS:
        case HCI_OPCODE_LE_READ_PER_ADV_LIST_SIZE:
// handle extended command

```

```

        if (hciCoreCb.extResetSeq)
        {
/* send next extended command in sequence */
            (*hciCoreCb.extResetSeq) (pMsg, opcode);
        }
        break;

        case HCI_OPCODE_LE_RAND:
/* last command in sequence set resetting state
* and call callback */
            signal_reset_sequence_done();
            break;

        default:
            break;
    }
}

virtual ble::vendor::cordio::buf_pool_desc_t
get_buffer_pool_description() {
    uint8_t buffer[1430];
    static const wsfBufPoolDesc_t pool_desc[] = {
        { 16, 14 },
        { 32, 10 },
        { 64, 4 },
        { 128, 2 },
        { 272, 1 }
    };

    return ble::vendor::cordio::buf_pool_desc_t(buffer, pool_desc);
}

private:

    void hciCoreReadMaxDataLen(void)
    {
/* if LE Data Packet Length Extensions is supported by Controller
* and included */
        if ((hciCoreCb.leSupFeat & HCI_LE_SUP_FEAT_DATA_LEN_EXT) &&
            (hciLeSupFeatCfg & HCI_LE_SUP_FEAT_DATA_LEN_EXT))
        {
/* send next command in sequence */
            HciLeReadMaxDataLen();
        }
        else
        {
/* send next command in sequence */
            HciLeRandCmd();
        }
    }

    void hciCoreReadResolvingListSize(void)
    {
/* if LL Privacy is supported by Controller and included */

```

```

        if ((hciCoreCb.leSupFeat & HCI_LE_SUP_FEAT_PRIVACY) &&
            (hciLeSupFeatCfg & HCI_LE_SUP_FEAT_PRIVACY))
        {
/* send next command in sequence */
            HciLeReadResolvingListSize();
        }
        else
        {
            hciCoreCb.resListSize = 0;
/* send next command in sequence */
            hciCoreReadMaxDataLen();
        }
    }
};

/**
 * Virtual Transport driver, used to exchange packet between host and
 * controller.
 */
class TransportDriver : public cordio::CordioHCITransportDriver {
public:

    TransportDriver() : _ble_base(BLE::DEFAULT_INSTANCE) {}

    virtual ~TransportDriver() { }

    /**
     * @see CordioHCITransportDriver::initialize
     */
    virtual void initialize() {
        /* Stack Initialization */
        //DTMStackInit();
        /* Periodic signal for BTLEStackTick initialization */
        //tick.attachus(callback(this, &StackTick), TICKMS*1000);
        tick.attach_us(mbed::callback(this, &ble::vendor::bluenrg::
            TransportDriver::StackTick), TICK_US);
    }

    /**
     * @see CordioHCITransportDriver::terminate
     */
    virtual void terminate() { }

    /**
     * @see CordioHCITransportDriver::write
     */
    virtual uint16_t write(uint8_t type, uint16_t len, uint8_t *pData) {
        if(type== HCI_CMD_TYPE){
            uint8_t resp_len;
            resp_len = process_command(pData, len, buffer_out, 255);
            rcv_callback(buffer_out, resp_len);
        }
        else if(type==HCI_ACL_TYPE){

```

```

        uint16_t connHandle;
        uint16_t dataLen;
        uint8_t* pduData;
        uint8_t pb_flag;
        uint8_t bc_flag;

        connHandle = ((pData[1] & 0x0F) << 8) + pData[0];
        dataLen = (pData[3] << 8) + pData[2];
        pduData = pData+4;
        pb_flag = (pData[1] >> 4) & 0x3;
        bc_flag = (pData[1] >> 6) & 0x3;
        hci_tx_acl_data(connHandle, pb_flag, bc_flag, dataLen,
            pduData);
    }

    return len;
}

private:
    uint16_t process_command(uint8_t *buffer_in, uint16_t buffer_in_length,
        uint8_t *buffer_out, uint16_t buffer_out_max_length){
        uint16_t ret_val, opCode;

        Osal_MemCpy(&opCode, buffer_in, 2);
        for (uint i = 0; i < (sizeof(hci_command_table)/sizeof(
            hci_command_table_type)); i++) {
            if (opCode == hci_command_table[i].opcode) {
                ret_val = hci_command_table[i].execute(buffer_in+2,
                    buffer_in_length-2, buffer_out, buffer_out_max_length);
                return ret_val;
            }
        }
        // Unknown command length
        buffer_out[0] = 0x04;
        buffer_out[1] = 0x0F;
        buffer_out[2] = 0x04;
        buffer_out[3] = 0x01;        ///01 unknown command ;-
        buffer_out[4] = 0x01;
        Osal_MemCpy(&buffer_out[5], &opCode, 2);
        return 7;
    }

    void StackTick(){
        _ble_base->signalEventsToProcess(BLE::DEFAULT_INSTANCE);
    }

    //buffer to store hci event packets generated after an hci command
    uint8_t buffer_out[258];
    Ticker tick;
    BLEInstanceBase* _ble_base;
};

} // namespace bluenrg
} // namespace vendor

```

```

} // namespace ble

/**
 * Cordio HCI driver factory
 */
ble::vendor::cordio::CordioHCIDriver& ble_cordio_get_hci_driver() {
    static ble::vendor::bluenrg::TransportDriver transport_driver;
    static ble::vendor::bluenrg::HCIDriver hci_driver(transport_driver);
    return hci_driver;
}

```

Low Power Mode - sleep_api.c

```

/*
 * sleepapi.h
 *
 * Created on: 25 jan 2019
 * Author: Antonio O.
 */

#ifdef DEVICE_SLEEP
    // IO13 - IO12 - IO11 - IO10 - IO9
#define GPIO_WAKE_BIT_MASK    31
// asserted pin in GPIO_WAKE_BIT_MASK are asserted at the value in the same
// position of this mask
#define GPIO_WAKE_LEVEL_MASK 0

#define SHPR3_REG 0xE000ED20

#define WAKENED_FROM_IO9      0x09
#define WAKENED_FROM_IO10     0x11
#define WAKENED_FROM_IO11     0x21
#define WAKENED_FROM_IO12     0x41
#define WAKENED_FROM_IO13     0x81
#define WAKENED_FROM_BLUE_TIMER1 0x101
#define WAKENED_FROM_BLUE_TIMER2 0x401

#define LOW_POWER_STANDBY    0x03

#define BLUE_CURRENT_TIME_REG 0x48000010

#include "sleep_api.h"
#include "bluenrg1_stack.h"
#include "misc.h"
#include "miscutil.h"

//BlueNRG2 sleepmodes types
typedef enum {
    SLEEPMODE_RUNNING        = 0,

```

```

    SLEEPMODE_CPU_HALT      = 1,
    SLEEPMODE_WAKETIMER    = 2,
    SLEEPMODE_NOTIMER      = 3,
} SleepModes;

uint32_t cStackPreamble[CSTACK_PREAMBLE_NUMBER];
volatile uint32_t* ptr ;

static void BlueNRG_HaltCPU(void) {
// Store the watchdog configuration and the disable it to avoid reset
//during CPU halt.
    uint32_t WDG_CR_saved = WDG->CR;
    WDG->CR = 0;

// Wait for interrupt is called: the core execution is halted until an
//event occurs.
    __WFI();

    // Restore the watchdog functionality.
    WDG->CR= WDG_CR_saved;
}

/** STMICROELECTRONICS DEEP SLEEP IMPLEMENTATION */
static void BlueNRG_DeepSleep(SleepModes sleepMode, uint8_t gpioWakeBitMask
)
{
    uint32_t savedCurrentTime, nvicPendingMask;
    PartInfoType partInfo;
    uint8_t i;
    /* System Control saved */
    uint32_t SYS_Ctrl_saved;
    /* NVIC Information Saved */
    uint32_t NVIC_ISER_saved, NVIC_IPR_saved[8], PENDSV_SYSTICK_IPR_saved;
    /* CKGEN SOC Enabled */
    uint32_t CLOCK_EN_saved;
    /* GPIO Information saved */
    uint32_t GPIO_DATA_saved, GPIO_OEN_saved, GPIO_PE_saved, GPIO_DS_saved,
        GPIO_IS_saved, GPIO_IBE_saved;
    uint32_t GPIO_IEV_saved, GPIO_IE_saved, GPIO_MODE0_saved,
        GPIO_MODE1_saved, GPIO_IOSEL_MFTX_saved;
#ifdef BLUENRG2_DEVICE
    uint32_t GPIO_MODE2_saved, GPIO_MODE3_saved;
#endif
    /* UART Information saved */
    uint32_t UART_TIMEOUT_saved, UART_LCRH_RX_saved, UART_IBRD_saved,
        UART_FBRD_saved;
    uint32_t UART_LCRH_TX_saved, UART_CR_saved, UART_IFLS_saved,
        UART_IMSC_saved;
    uint32_t UART_DMACR_saved, UART_XFCR_saved, UART_XON1_saved,
        UART_XON2_saved;
    uint32_t UART_XOFF1_saved, UART_XOFF2_saved;
    /* SPI Information saved */
    uint32_t SPI_CR0_saved, SPI_CR1_saved, SPI_CPSR_saved, SPI_IMSC_saved,
        SPI_DMACR_saved;

```

```

uint32_t SPI_RXFRM_saved, SPI_CHN_saved, SPI_WDTXF_saved;
/* I2C Information saved */
uint32_t I2C_CR_saved[2], I2C_SCR_saved[2], I2C_TFTR_saved[2],
        I2C_RFTR_saved[2];
uint32_t I2C_DMAR_saved[2], I2C_BRCCR_saved[2], I2C_IMSCR_saved[2],
        I2C_THDDAT_saved[2];
uint32_t I2C_THDSTA_FST_STD_saved[2], I2C_TSUSTA_FST_STD_saved[2];
/* RNG Information saved */
uint32_t RNG_CR_saved;
/* SysTick Information saved */
uint32_t SYST_CSR_saved, SYST_RVR_saved;
/* RTC Information saved */
uint32_t RTC_CWDMR_saved, RTC_CWDLR_saved, RTC_CWYMR_saved,
        RTC_CWYLR_saved, RTC_CTCR_saved;
uint32_t RTC_IMSC_saved, RTC_TCR_saved, RTC_TLR1_saved, RTC_TLR2_saved,
        RTC_TPR1_saved;
uint32_t RTC_TPR2_saved, RTC_TPR3_saved, RTC_TPR4_saved;
/* MFTX Information saved */
uint32_t T1CRA_saved, T1CRB_saved, T1PRSC_saved, T1CKC_saved,
        T1MCTRL_saved, T1ICTRL_saved;
uint32_t T2CRA_saved, T2CRB_saved, T2PRSC_saved, T2CKC_saved,
        T2MCTRL_saved, T2ICTRL_saved;
/* WDT Information saved */
uint32_t WDG_LR_saved, WDG_CR_saved, WDG_LOCK_saved;
/* DMA channel [0..7] Information saved */
uint32_t DMA_CCR_saved[8], DMA_CNDTR_saved[8], DMA_CPAR_saved[8],
        DMA_CMAR[8];
/* ADC Information saved */
uint32_t ADC_CTRL_saved, ADC_CONF_saved, ADC_IRQMASK_saved,
        ADC_OFFSET_LSB_saved, ADC_OFFSET_MSB_saved;
uint32_t ADC_THRESHOLD_HI_saved, ADC_THRESHOLD_LO_saved;
/* FLASH Config saved */
uint32_t FLASH_CONFIG_saved;
/* PKA Information saved */
uint32_t PKA_IEN_saved;

/* Get partInfo */
HAL_GetPartInfo(&partInfo);

/* Save the peripherals configuration */
/* System Control */
SYS_Ctrl_saved = SYSTEM_CTRL->CTRL;
/* FLASH CONFIG */
FLASH_CONFIG_saved = FLASH->CONFIG;
/* NVIC */
NVIC_IUSER_saved = NVIC->IUSER[0];

// Issue with Atollic compiler
// memcpy(NVIC_IPR_saved, (void const *)NVIC->IP, sizeof(NVIC_IPR_saved));
for (i=0; i<8; i++) {
    NVIC_IPR_saved[i] = NVIC->IP[i];
}

```

```

PENDSV_SYSTICK_IPR_saved = *(volatile uint32_t *) SHPR3_REG;
/* CKGEN SOC Enabled */
CLOCK_EN_saved = CKGEN_SOC->CLOCK_EN;
/* GPIO */
GPIO_DATA_saved = GPIO->DATA;
GPIO_OEN_saved = GPIO->OEN;
GPIO_PE_saved = GPIO->PE;
GPIO_DS_saved = GPIO->DS;
GPIO_IS_saved = GPIO->IS;
GPIO_IBE_saved = GPIO->IBE;
GPIO_IEV_saved = GPIO->IEV;
GPIO_IE_saved = GPIO->IE;
GPIO_MODE0_saved = GPIO->MODE0;
GPIO_MODE1_saved = GPIO->MODE1;
#ifdef BLUENRG2_DEVICE
GPIO_MODE2_saved = GPIO->MODE2;
GPIO_MODE3_saved = GPIO->MODE3;
#endif
GPIO_IOSEL_MFTX_saved = GPIO->MFTX;
/* UART */
UART_TIMEOUT_saved = UART->TIMEOUT;
UART_LCRH_RX_saved = UART->LCRH_RX;
UART_IBRD_saved = UART->IBRD;
UART_FBRD_saved = UART->FBRD;
UART_LCRH_TX_saved = UART->LCRH_TX;
UART_CR_saved = UART->CR;
UART_IFLS_saved = UART->IFLS;
UART_IMSC_saved = UART->IMSC;
UART_DMACR_saved = UART->DMACR;
UART_XFCR_saved = UART->XFCR;
UART_XON1_saved = UART->XON1;
UART_XON2_saved = UART->XON2;
UART_XOFF1_saved = UART->XOFF1;
UART_XOFF2_saved = UART->XOFF2;
/* SPI */
SPI_CR0_saved = SPI1->CR0;
SPI_CR1_saved = SPI1->CR1;
SPI_CPSR_saved = SPI1->CPSR;
SPI_IMSC_saved = SPI1->IMSC;
SPI_DMACR_saved = SPI1->DMACR;
SPI_RXFRM_saved = SPI1->RXFRM;
SPI_CHN_saved = SPI1->CHN;
SPI_WDTXF_saved = SPI1->WDTXF;
/* I2C */
for (i=0; i<2; i++) {
    I2C_Type *I2Cx = (I2C_Type*)(I2C2_BASE+ 0x100000*i);
    I2C_CR_saved[i] = I2Cx->CR;
    I2C_SCR_saved[i] = I2Cx->SCR;
    I2C_TFTR_saved[i] = I2Cx->TFTR;
    I2C_RFTR_saved[i] = I2Cx->RFTR;
    I2C_DMAR_saved[i] = I2Cx->DMAR;
    I2C_BRCCR_saved[i] = I2Cx->BRCCR;
    I2C_IMSCR_saved[i] = I2Cx->IMSCR;
    I2C_THDDAT_saved[i] = I2Cx->THDDAT;
}

```

```

    I2C_THDSTA_FST_STD_saved[i] = I2Cx->THDSTA_FST_STD;
    I2C_TSUSTA_FST_STD_saved[i] = I2Cx->TSUSTA_FST_STD;
}
/* RNG */
RNG_CR_saved = RNG->CR;
/* RTC */
RTC_CWDMR_saved = RTC->CWDMR;
RTC_CWDLR_saved = RTC->CWDLR;
RTC_CWYMR_saved = RTC->CWYMR;
RTC_CWYLR_saved = RTC->CWYLR;
RTC_CTCR_saved = RTC->CTCR;
RTC_IMSC_saved = RTC->IMSC;
RTC_TCR_saved = RTC->TCR;
RTC_TLR1_saved = RTC->TLR1;
RTC_TLR2_saved = RTC->TLR2;
RTC_TPR1_saved = RTC->TPR1;
RTC_TPR2_saved = RTC->TPR2;
RTC_TPR3_saved = RTC->TPR3;
RTC_TPR4_saved = RTC->TPR4;
/* MFTX */
T1CRA_saved = MFT1->TNCRA;
T1CRB_saved = MFT1->TNCRB;
T1PRSC_saved = MFT1->TNPRSC;
T1CKC_saved = MFT1->TNCKC;
T1MCTRL_saved = MFT1->TNMCTRL;
T1ICTRL_saved = MFT1->TNICTRL;
T2CRA_saved = MFT2->TNCRA;
T2CRB_saved = MFT2->TNCRB;
T2PRSC_saved = MFT2->TNPRSC;
T2CKC_saved = MFT2->TNCKC;
T2MCTRL_saved = MFT2->TNMCTRL;
T2ICTRL_saved = MFT2->TNICTRL;
/* SysTick */
SYST_CSR_saved = SysTick->CTRL;
SYST_RVR_saved = SysTick->LOAD;
/* WDT */
WDG_LR_saved = WDG->LR;
WDG_CR_saved = WDG->CR;
if (WDG->LOCK == 0) {
    WDG_LOCK_saved = 0x1ACCE551;
} else {
    WDG_LOCK_saved = 0;
}
/* DMA */
for (i=0; i<8; i++) {
    DMA_CH_Type *DMAx = (DMA_CH_Type*) (DMA_CH0_BASE+ 0x14*i);
    DMA_CNDTR_saved[i] = DMAx->CNDTR;
    DMA_CCR_saved[i] = DMAx->CCR;
    DMA_CPAR_saved[i] = DMAx->CPAR;
    DMA_CMAR[i] = DMAx->CMAR;
}
/* ADC */
ADC_CONF_saved = ADC->CONF;
ADC_IRQMASK_saved = ADC->IRQMASK;

```

```

ADC_OFFSET_MSB_saved = ADC->OFFSET_MSB;
ADC_OFFSET_LSB_saved = ADC->OFFSET_LSB;
ADC_THRESHOLD_HI_saved = ADC->THRESHOLD_HI;
ADC_THRESHOLD_LO_saved = ADC->THRESHOLD_LO;
ADC_CTRL_saved = ADC->CTRL;

/* PKA */
PKA_IEN_saved = PKA->IEN;

// Enable the STANDBY mode
if (sleepMode == SLEEPMODE_NOTIMER) {
    BLUE_CTRL->TIMEOUT |= LOW_POWER_STANDBY<<28;
}

//Save the CSTACK number of words that will be restored at wakeup reset
i = 0;
ptr = __vector_table[0]...ptr ;
ptr -= CSTACK_PREAMBLE_NUMBER;
do {
    cStackPreamble[i] = *ptr;
    i++;
    ptr++;
} while (i < CSTACK_PREAMBLE_NUMBER);

if (((partInfo.die_major<<4)|(partInfo.die_cut)) >= WA_DEVICE_VERSION) {
    /* Lock the flash */
    flash_sw_lock = FLASH_LOCK_WORD;
    /* Disable BOR */
    SET_BORconfigStatus(FALSE);
}

//Enable deep sleep
SystemSleepCmd(ENABLE);
//The disableirq() used at the beginning of the BlueNRGSleep()
//function
//masks all the interrupts. The interrupts will be enabled at the end of
//the context restore. Now induce a context save.
void CS_contextSave(void);
CS_contextSave();

//Disable deep sleep, because if no reset occurs for an interrupt
//pending, the register value remain set and if a simple CPUHALT command
//is called from the application the BlueNRG-1 enters in deep sleep
//without make a context save.
//So, exiting from the deep sleep the context is restored with
//wrong random value.
SystemSleepCmd(DISABLE);

if (!wakeupFromSleepFlag) {
    if ((NVIC->ISPR[0]&(1<<BLUE_CTRL_IRQn)) == 0) {
//At this stage the Blue Control Interrupt shall not be pending.
//So, if this happens means that the application has called the
//BlueNRGSleep() API with the wakeup source already acrive.
//In this scenario we don't need to wait the 91 us, otherwise

```

```

//the radio activity will be compromised.
    nvicPendingMask = savedNVIC_ISPR ^ NVIC->ISPR[0];
    if ((savedSHCSR != SCB->SHCSR) ||
//Verified if a SVCcall Interrupt is pending
        ((savedNVIC_ISPR != NVIC->ISPR[0]) && (nvicPendingMask & NVIC->
            ISER[0]))
//Verified if a NVIC Interrupt is pending
        ((savedICSR & 0x10000000) != (SCB->ICSR & 0x10000000)) ||
// Verified if a PendSV interrupt is pending
        (((savedICSR & 0x40000000) != (SCB->ICSR & 0x40000000)) && (SysTick
            ->CTRL & 0x02))) {
// Verified if a SysTick interrupt is pending
    savedCurrentTime = (*(volatile uint32_t *)BLUE_CURRENT_TIME_REG) >>
        4;
    if (0xFFFFF >= (savedCurrentTime+3)) {
//Check if the counter are wrapping
        while ((savedCurrentTime+3) > ((* (volatile uint32_t *)
            BLUE_CURRENT_TIME_REG) >> 4)); //Not Wrap
        } else {
            while (((*(volatile uint32_t *)BLUE_CURRENT_TIME_REG) >> 4) != (
                savedCurrentTime + 3 - 0xFFFFF)); //Wrap
        }
    }
}

if (((partInfo.die_major<<4)|(partInfo.die_cut)) >= WA_DEVICE_VERSION)
{
    /* Restore BOR configuration */
    SET_BORconfigStatus(TRUE);
    /* Unlock the flash */
    flash_sw_lock = FLASH_UNLOCK_WORD;
}

// Disable the STANDBY mode
if (sleepMode == SLEEPMODE_NOTIMER) {
    BLUE_CTRL->TIMEOUT &= ~(LOW_POWER_STANDBY<<28);
}

} else {

    /* Start a new calibration, needed to signal if the HS is ready */
    CKGEN_BLE->CLK32K_IT = 1;
    CKGEN_BLE->CLK32K_COUNT = 0;
    CKGEN_BLE->CLK32K_PERIOD = 0;

// Restore the CSTACK number of words that will be saved before the sleep
    i = 0;
    ptr = __vector_table[0]...ptr ;
    ptr -= CSTACK_PREAMBLE_NUMBER;
    do {
        *ptr = cStackPreamble[i];
        i++;
        ptr++;
    } while (i < CSTACK_PREAMBLE_NUMBER);
}

```

```

/* Restore the peripherals configuration */
/* FLASH CONFIG */
FLASH->CONFIG = FLASH_CONFIG_saved;
/* NVIC */
NVIC->ISER[0] = NVIC_ISER_saved;

for (i=0; i<8; i++) {
    NVIC->IP[i] = NVIC_IPR_saved[i];
}

*(volatile uint32_t *)SHPR3_REG = PENDSV_SYSTICK_IPR_saved;
/* CKGEN SOC Enabled */
CKGEN_SOC->CLOCK_EN = CLOCK_EN_saved;
/* GPIO */
GPIO->DATA = GPIO_DATA_saved;
GPIO->OEN = GPIO_OEN_saved;
GPIO->PE = GPIO_PE_saved;
GPIO->DS = GPIO_DS_saved;
GPIO->IEV = GPIO_IEV_saved;
GPIO->IBE = GPIO_IBE_saved;
GPIO->IS = GPIO_IS_saved;
GPIO->IC = GPIO_IE_saved;
GPIO->IE = GPIO_IE_saved;
GPIO->MODE0 = GPIO_MODE0_saved;
GPIO->MODE1 = GPIO_MODE1_saved;
#ifdef BLUENRG2_DEVICE
GPIO->MODE2 = GPIO_MODE2_saved;
GPIO->MODE3 = GPIO_MODE3_saved;
#endif
GPIO->MFTX = GPIO_IOSEL_MFTX_saved;
/* UART */
UART->TIMEOUT = UART_TIMEOUT_saved;
UART->LCRH_RX = UART_LCRH_RX_saved;
UART->IBRD = UART_IBRD_saved;
UART->FBRD = UART_FBRD_saved;
UART->LCRH_TX = UART_LCRH_TX_saved;
UART->CR = UART_CR_saved;
UART->IFLS = UART_IFLS_saved;
UART->IMSC = UART_IMSC_saved;
UART->DMACR = UART_DMACR_saved;
UART->XFCR = UART_XFCR_saved;
UART->XON1 = UART_XON1_saved;
UART->XON2 = UART_XON2_saved;
UART->XOFF1 = UART_XOFF1_saved;
UART->XOFF2 = UART_XOFF2_saved;
/* SPI */
SPI1->CR0 = SPI_CR0_saved;
SPI1->CR1 = SPI_CR1_saved;
SPI1->CPSR = SPI_CPSR_saved;
SPI1->IMSC = SPI_IMSC_saved;
SPI1->DMACR = SPI_DMACR_saved;
SPI1->RXFRM = SPI_RXFRM_saved;

```

```

SPI1->CHN = SPI_CHN_saved;
SPI1->WDTXF = SPI_WDTXF_saved;
/* I2C */
for (i=0; i<2; i++) {
    I2C_Type *I2Cx = (I2C_Type*)(I2C2_BASE+ 0x100000*i);
    I2Cx->CR = I2C_CR_saved[i];
    I2Cx->SCR = I2C_SCR_saved[i];
    I2Cx->TFTR = I2C_TFTR_saved[i];
    I2Cx->RFTR = I2C_RFTR_saved[i];
    I2Cx->DMAR = I2C_DMAR_saved[i];
    I2Cx->BRCCR = I2C_BRCCR_saved[i];
    I2Cx->IMSCR = I2C_IMSCR_saved[i];
    I2Cx->THDDAT = I2C_THDDAT_saved[i];
    I2Cx->THDSTA_FST_STD = I2C_THDSTA_FST_STD_saved[i];
    I2Cx->TSUSTA_FST_STD = I2C_TSUSTA_FST_STD_saved[i];
}
/* RNG */
RNG->CR = RNG_CR_saved;
/* SysTick */
SysTick->LOAD = SYST_RVR_saved;
SysTick->VAL = 0;
SysTick->CTRL = SYST_CSR_saved;
/* RTC */
RTC->CWDMMR = RTC_CWDMMR_saved;
RTC->CWDLR = RTC_CWDLR_saved;
RTC->CWYMR = RTC_CWYMR_saved;
RTC->CWYLR = RTC_CWYLR_saved;
RTC->CTCR = RTC_CTCR_saved;
RTC->IMSC = RTC_IMSC_saved;
RTC->TLR1 = RTC_TLR1_saved;
RTC->TLR2 = RTC_TLR2_saved;
RTC->TPR1 = RTC_TPR1_saved;
RTC->TPR2 = RTC_TPR2_saved;
RTC->TPR3 = RTC_TPR3_saved;
RTC->TPR4 = RTC_TPR4_saved;
/* Enable moved at the end of RTC configuration */
RTC->TCR = RTC_TCR_saved;
/* MFTX */
MFT1->TNCRA = T1CRA_saved;
MFT1->TNCRB = T1CRB_saved;
MFT1->TNPRSC = T1PRSC_saved;
MFT1->TNCKC = T1CKC_saved;
MFT1->TNMCTRL = T1MCTRL_saved;
MFT1->TNICTRL = T1ICTRL_saved;
MFT2->TNCRA = T2CRA_saved;
MFT2->TNCRB = T2CRB_saved;
MFT2->TNPRSC = T2PRSC_saved;
MFT2->TNCKC = T2CKC_saved;
MFT2->TNMCTRL = T2MCTRL_saved;
MFT2->TNICTRL = T2ICTRL_saved;
/* WDT */
WDG->LR = WDG_LR_saved;
WDG->CR = WDG_CR_saved;
WDG->LOCK = WDG_LOCK_saved;

```

```

/* DMA */
for (i=0; i<8; i++) {
    DMA_CH_Type *DMAx = (DMA_CH_Type*)(DMA_CH0_BASE+ 0x14*i);
    DMAx->CNDTR = DMA_CNDTR.saved[i];
    DMAx->CCR = DMA_CCR.saved[i] ;
    DMAx->CPAR = DMA_CPAR.saved[i];
    DMAx->CMAR = DMA_CMAR[i];
}
/* ADC */
ADC->CONF = ADC_CONF.saved;
ADC->IRQMASK = ADC_IRQMASK.saved;
ADC->OFFSET_MSB = ADC_OFFSET_MSB.saved;
ADC->OFFSET_LSB = ADC_OFFSET_LSB.saved;
ADC->THRESHOLD_HI = ADC_THRESHOLD_HI.saved;
ADC->THRESHOLD_LO = ADC_THRESHOLD_LO.saved;
ADC->CTRL = ADC_CTRL.saved;

/* PKA */
PKA->IEN = PKA_IEN.saved;
//The five IRQs are linked to a real ISR. If any of the five IRQs
//triggered, then pend their ISR
//Capture the wake source from the BLEREASONRESET register
if ((CKGEN_SOC->REASON_RST == 0) &&
    (CKGEN_BLE->REASON_RST >= WAKENED_FROM_IO9) &&
    (CKGEN_BLE->REASON_RST <= WAKENED_FROM_IO13) &&
    gpioWakeBitMask) {
    if (((CKGEN_BLE->REASON_RST & WAKENED_FROM_IO9) ==
        WAKENED_FROM_IO9) && (GPIO->IE & GPIO_Pin_9)) ||
        ((CKGEN_BLE->REASON_RST & WAKENED_FROM_IO10) ==
        WAKENED_FROM_IO10) && (GPIO->IE & GPIO_Pin_10)) ||
        ((CKGEN_BLE->REASON_RST & WAKENED_FROM_IO11) ==
        WAKENED_FROM_IO11) && (GPIO->IE & GPIO_Pin_11)) ||
        ((CKGEN_BLE->REASON_RST & WAKENED_FROM_IO12) ==
        WAKENED_FROM_IO12) && (GPIO->IE & GPIO_Pin_12)) ||
        ((CKGEN_BLE->REASON_RST & WAKENED_FROM_IO13) ==
        WAKENED_FROM_IO13) && (GPIO->IE & GPIO_Pin_13)))
        {
            NVIC->ISPR[0] = 1<<GPIO_IRQn;
        }
}

// Disable the STANDBY mode
if (sleepMode == SLEEPMODE_NOTIMER) {
    BLUE_CTRL->TIMEOUT &= ~(LOW_POWER_STANDBY<<28);
}

//Restore the System Control register to indicate which HS crystal is used
SYSTEM_CTRL->CTRL = SYS_Ctrl.saved;

// Wait until the HS clock is ready.
// If SLEEPMODENOTIMER is set, wait the LS clock is ready.
if (sleepMode == SLEEPMODE_NOTIMER) {
    DeviceConfiguration(FALSE, TRUE);
} else {

```

```

    DeviceConfiguration(FALSE, FALSE);
}

/* If the HS is a 32 MHz */
if (SYS_Ctrl_saved & 1) {
#if (FORCE_CORE_TO_16MHZ == 1)
    /* AHB up converter command register write*/
    AHBUPCONV->COMMAND = 0x14;
#else
    /* AHB up converter command register write*/
    AHBUPCONV->COMMAND = 0x15;
#endif
}

//We can clear PRIMASK to reenale global interrupt operation.
//enableirq(); //done in haldeepsleep
}

void hal_sleep(void) {
    //only CPU halt, wakeup from any interrupt source
    //wakeup timer is not available

    // Disable IRQs
    core_util_critical_section_enter();

    //Flag to signal if a wakeup from standby or sleep occurred
    wakeupFromSleepFlag = 0;

    //ask the BLE controller if link layer termination is ongoing,
    //to go sleep at least it shall return at least SLEEPMODECPUHALT
    SleepModes sleepMode_allowed = (SleepModes)
        BlueNRG_Stack_Perform_Deep_Sleep_Check();

    if(sleepMode_allowed >= SLEEPMODE_CPU_HALT) {
        BlueNRG_HaltCPU();
    }

    // Unmask all the interrupt
    core_util_critical_section_exit();
}

void hal_deepsleep(void) {
    //check no active UART RX - when tx ongoing fifo empty flag is 0 (RESET)
    #if DEVICE_SERIAL
        serialTxActive();
    #endif

    // Disable IRQs
    core_util_critical_section_enter();

    //Flag to signal if a wakeup from standby or sleep occurred
    wakeupFromSleepFlag = 0;
}

```

```

//ask the BLE controller if link layer termination is ongoing,
//to go sleep at least it shall return at least SLEEPMODEWAKETIMER - only
//GPIO wakeup available
    volatile SleepModes sleepMode_allowed = (SleepModes)
        BlueNRG_Stack_Perform_Deep_Sleep_Check();

    switch(sleepMode_allowed) {
    case SLEEPMODE_CPU_HALT:
        BlueNRG_HaltCPU();
        break;
    case SLEEPMODE_WAKETIMER:
    case SLEEPMODE_NOTIMER:
        //Setup the GPIO Wakeup Source
        //sleepMode_allowed = SLEEPMODEWAKETIMER;
        SYSTEM_CTRL->WKP_IO_IS = GPIO_WAKE_LEVEL_MASK;
        SYSTEM_CTRL->WKP_IO_IE = GPIO_WAKE_BIT_MASK;
        BlueNRG_DeepSleep(sleepMode_allowed, GPIO_WAKE_BIT_MASK);
        break;
    default:
        break;
    }

    // Unmask all the interrupt
    core_util_critical_section_exit();
}

#endif //DEVICE_SLEEP

```

ARM Mbed OS HRM (Heart Rate Monitor) - main.cpp

```

/* mbed Microcontroller Library
 * Copyright (c) 2006-2015 ARM Limited
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *     http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

#include <events/mbed_events.h>
#include <mbed.h>
#include "ble/BLE.h"
#include "ble/gap/Gap.h"
#include "ble/services/HeartRateService.h"

```

```

#include "pretty_printer.h"

const static char DEVICE_NAME[] = "ORLHRM";

static events::EventQueue event_queue(/* event count */ 16 *
EVENTS_EVENT_SIZE);

class HeartrateDemo : ble::Gap::EventHandler {
public:
    HeartrateDemo(BLE &ble, events::EventQueue &event_queue) :
        _ble(ble),
        _event_queue(event_queue),
        _led1(LED1, 1),
        _connected(false),
        _hr_uuid(GattService::UUID_HEART_RATE_SERVICE),
        _hr_counter(100),
        _hr_service(ble, _hr_counter, HeartRateService::LOCATION_FINGER),
        _adv_data_builder(_adv_buffer) { }

    void start() {
        _ble.gap().setEventHandler(this);

        _ble.init(this, &HeartrateDemo::on_init_complete);

        _event_queue.call_every(500, this, &HeartrateDemo::blink);
        _event_queue.call_every(1000, this, &HeartrateDemo::
            update_sensor_value);

        _event_queue.dispatch_forever();
    }

private:
    //Callback triggered when the ble initialization process has finished
    void on_init_complete(BLE::InitializationCompleteCallbackContext *
        params) {
        if (params->error != BLE_ERROR_NONE) {
            printf("Ble initialization failed.");
            return;
        }

        print_mac_address();

        start_advertising();
    }

    void start_advertising() {
        /* Create advertising parameters and payload */

        ble::AdvertisingParameters adv_parameters(
            ble::advertising_type_t::CONNECTABLE_UNDIRECTED,
            ble::adv_interval_t(ble::millisecond_t(1000))
        );

        _adv_data_builder.setFlags();
    }

```

```

_adv_data_builder.setAppearance(ble::adv_data_appearance_t::
    GENERIC_HEART_RATE_SENSOR);
_adv_data_builder.setLocalServiceList(mbed::make_Span(&_hr_uuid, 1)
);
_adv_data_builder.setName(DEVICE_NAME);

/* Setup advertising */

ble_error_t error = _ble.gap().setAdvertisingParameters(
    ble::LEGACY_ADVERTISING_HANDLE,
    adv_parameters
);

if (error) {
    printf("ble.gap().setAdvertisingParameters() failed\r\n");
    return;
}

error = _ble.gap().setAdvertisingPayload(
    ble::LEGACY_ADVERTISING_HANDLE,
    _adv_data_builder.getAdvertisingData()
);

if (error) {
    printf("ble.gap().setAdvertisingPayload() failed\r\n");
    return;
}

/* Start advertising */

error = _ble.gap().startAdvertising(ble::LEGACY_ADVERTISING_HANDLE)
;

if (error) {
    printf("ble.gap().startAdvertising() failed\r\n");
    return;
}
}

void update_sensor_value() {
    if (!_connected) {
// Do blocking calls or whatever is necessary for sensor polling.
// In our case, we simply update the HRM measurement.
        _hr_counter++;

        // 100 i= HRM bps i=175
        if (_hr_counter == 175) {
            _hr_counter = 100;
        }

        _hr_service.updateHeartRate(_hr_counter);
    }
}
}

```

```

    void blink(void) {
        _led1 = !_led1;
    }

private:
    /* Event handler */

    void onDisconnectionComplete(const ble::DisconnectionCompleteEvent&) {
        _ble.gap().startAdvertising(ble::LEGACY_ADVERTISING_HANDLE);
        _connected = false;
    }

    virtual void onConnectionComplete(const ble::ConnectionCompleteEvent &
        event) {
        if (event.getStatus() == BLE_ERROR_NONE) {
            _connected = true;
        }
    }

private:
    BLE &_ble;
    events::EventQueue &_event_queue;
    DigitalOut _led1;

    bool _connected;

    UUID _hr_uuid;

    uint8_t _hr_counter;
    HeartRateService _hr_service;

    uint8_t _adv_buffer[ble::LEGACY_ADVERTISING_MAX_SIZE];
    ble::AdvertisingDataBuilder _adv_data_builder;
};

//Schedule processing of events from the BLE middleware in the event queue.
void schedule_ble_events(BLE::OnEventsToProcessCallbackContext *context) {
    event_queue.call(Callback<void()>(&context->ble, &BLE::processEvents));
}

int main()
{
    BLE &ble = BLE::Instance();
    ble.onEventsToProcess(schedule_ble_events);

    HeartrateDemo demo(ble, event_queue);
    demo.start();

    return 0;
}

```


Appendix B - Toolchain Setup

In this Appendix there is explained how to configure an Eclipse CDT-based free development environment for mbed-OS, as an alternative to the priced counterparts (IAR EWARM, Keil μ Vision) and the ST supported toolchain Atollic TrueSTUDIO (that is currently under development).

The given configuration has been used during the BlueNRG-2 Mbed OS porting activity.

Prerequisites

To ensure a correct management of binary installation (and future updates of them), it is useful to use `xpm`, a package manager built on a javascript runtime (Node.js), that allows an high portability on different development platform and a correct organization of the folder hierarchy (in such a way Eclipse is able to resolve all its path dependencies) [13]. The following description relies on the use of `xpm`.

First of all, the following elements must be installed:

1. ARM toolchain, by using the command:

```
$ xpm install --global @gnu-mcu-eclipse/arm-none-eabi-gcc
```

2. This step is *Windows specific*, it provides, among the others, *make.exe*:

```
$ xpm install --global @gnu-mcu-eclipse/windows-build-tools
```

3. OpenOCD, a software interface for the STM ST-Link debugger:

```
$ xpm install --global @gnu-mcu-eclipse/openocd
```

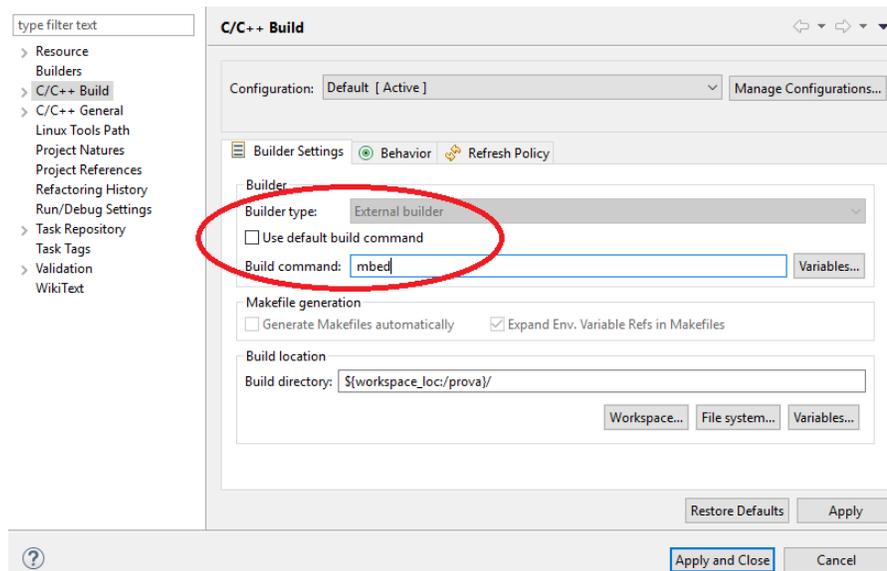
4. Java Development Kit (through its installation wizard) [18].

5. Eclipse-CDT with MCU plugins. The simplest way to get it is to download a ready-to-go version from the GNU MCU Eclipse webpage. [13].
6. mbed-CLI (through its installation wizard) [10].

C/C++ Build

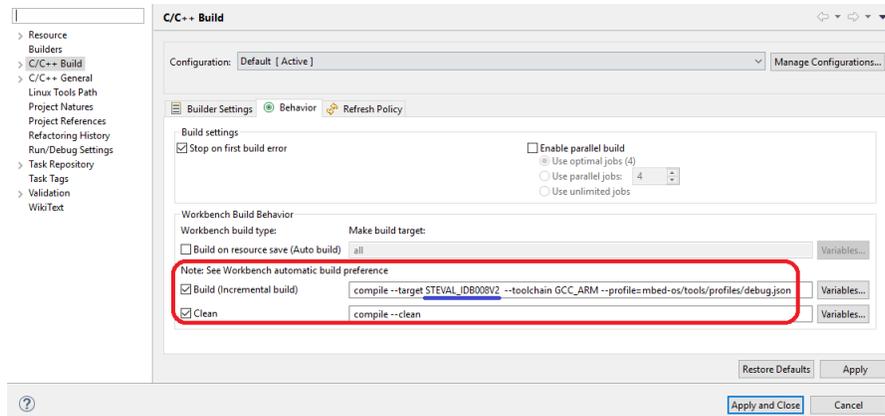
Using the classic Eclipse C/C++ toolchain (GCC_ARM, based on the autogeneration of makefile provided by Mbed-OS python exporters and Eclipse CDT) is feasible, but the ARM Mbed OS compliance is obtained when the source code passes some tests provided in the Mbed-OS distribution, compiling using the Mbed-CLI. For this reason Mbed-CLI compiler support has been introduced in Eclipse too.

To perform this migration one has to open the *Project - Properties*, then choose *External builder* in the *Builder type:* field, deselect the item *Use default build command* and type “mbed” into the *Build command:* label.



Builder Settings

After that, in the *Behavior* tab one has to configure options for build and clean, choosing the target name identifying the development board in the *targets.json* file in the mbed-OS project folder (in this case, as example, is reported the name of the evaluation board of BlueNRG-2, underlined in blue).



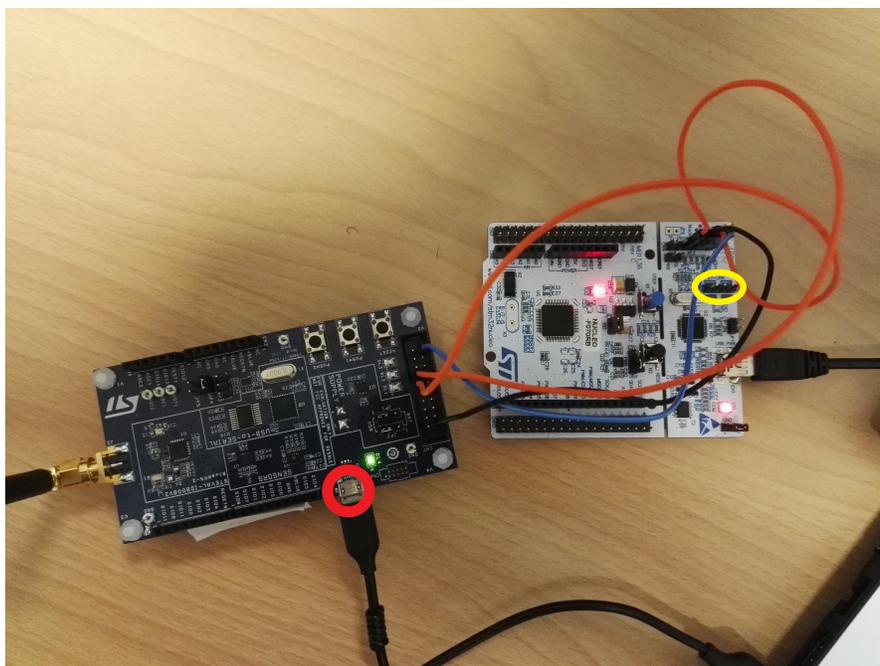
Behavior

Debug and Run configuration

The debug configuration relies on:

- ST-Link v2.1 hardware adapter;
- GDB + OpenOCD software interface. [25]

In particular, the ST-Link hardware adapter is not the standalone one, but the one embedded on a STM32 NUCLEO-F070RB. While using this kind of hardware it is mandatory to configure the SWD *without hardware reset pin*, due to the design of the Nucleo board. Moreover, it is impossible to use the ST-Link in JTAG mode or SWIM (Single Wire Interface Module): the only admitted mode is the 2 wires SWD plus VCC detection and ground, so a 4 wires connection. On the NUCLEO ST-Link board it is noticeable the removal of the two yellow-highlighted jumpers, essential for the external programming usage. One has also to notice that no UART connection is provided to the ST-Link (it is provided to the PC through the connector on the target board - red).



ST-Link connection to STEVAL-IDB008V2

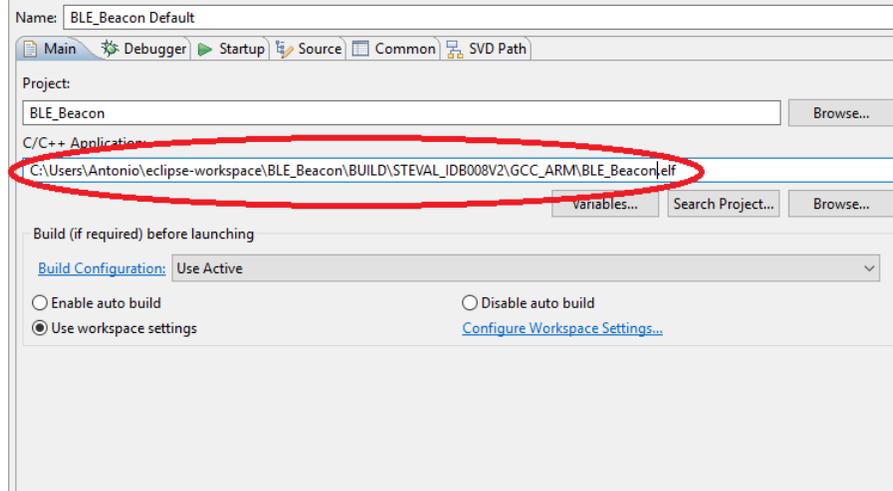
To handle those problems, the board (in this case the STEVAL-IDB008V2 [29], used as development platform) OpenOCD TCL script files have been modified, adding the instructions in the highlighted lines in the following figure, in such a way as to ensure:

- definition of *HLA-SWD* transports (the default one is JTAG);
- BlueNRG-2 software reset before and after flash programming.

```
# This is an evaluation board with a single BlueNRG-2 chip.  
# http://www.st.com/content/st_com/en/products/evaluation-to  
set CHIPNAME bluenrg-2  
source [find interface/stlink.cfg]  
transport select hla_swd  
source [find target/bluenrg-x.cfg]  
reset_config none separate
```

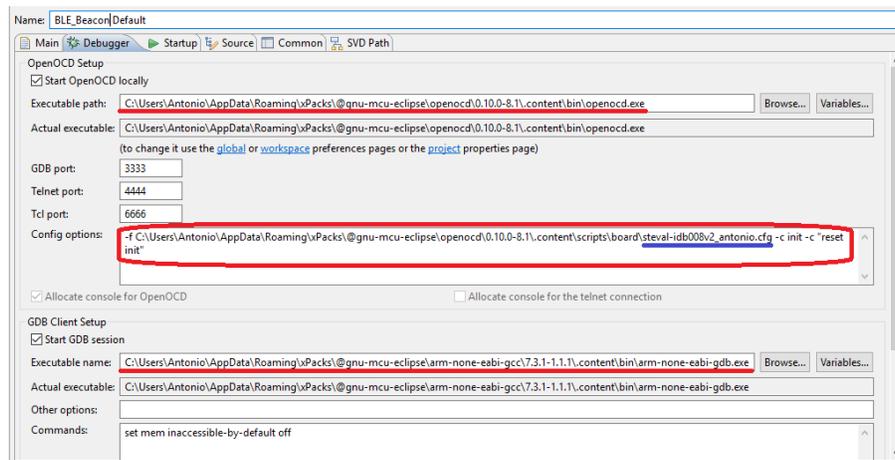
Modified OpenOCD script

On Eclipse, one has to add a new OpenOCD from the menu *Debug - Debug Configurations*, setting up the path to the *.elf* executable file.



Debug Configuration - Main

At the end, one has to configure the path for the OpenOCD executable, board script files, init options, and the path to GDB executable.



Debug Configuration - Debugger

Bibliography

- [1] Apple. *Getting Started with iBeacon*, 2014. Published at <https://developer.apple.com/ibeacon/Getting-Started-with-iBeacon.pdf>.
- [2] ARM. *Serial Wire Debug - 2-Pin Debug Port*. Published at <https://developer.arm.com/products/architecture/cpu-architecture/debug-visibility-and-trace/coresight-architecture/serial-wire-debug> - last visit: Oct. 2018.
- [3] ARM. *Cortex™-M0 Devices - Generic User Guide*, 2009. [DUI0497A] - Published at http://infocenter.arm.com/help/topic/com.arm.doc.dui0497a/DUI0497A_cortex.m0_r0p0_generic_ug.pdf.
- [4] ARM. *Cortex™-M0 Revision r0p0 - Technical Reference Manual*, 2009. [DDI0432C] - Published at http://infocenter.arm.com/help/topic/com.arm.doc.ddi0432c/DDI0432C_cortex.m0_r0p0_trm.pdf.
- [5] ARM. *ARM®v6-M Architecture - Reference Manual*, 2017. [DDI0419D] - Published at https://static.docs.arm.com/ddi0419/d/DDI0419D_armv6m_arm.pdf.
- [6] ARM. *ARM Mbed GitHub repository - BLE Examples*, 2018. Available (with examples descriptions) at <https://github.com/ARMmbed/mbed-os-example-ble> - last visit: Feb. 2019.
- [7] ARM. *ARM Mbed GitHub repository - Mbed OS 5*, 2018. Available at <https://github.com/ARMmbed> - last visit: Feb. 2019.

- [8] ARM. *ARM Mbed OS 5 - Bluetooth overview*, 2018. Published at <https://os.mbed.com/docs/v5.11/apis/bluetooth.html> - last visit: Feb. 2019.
- [9] ARM. *ARM Mbed OS 5 - Porting Guide*, 2018. Published at <https://os.mbed.com/docs/v5.11/porting/index.html> - last visit: Feb. 2019.
- [10] ARM. *Mbed OS reference book - version 5.11*, 2018. Sections "Reference, Tools, Tutorials", published at <https://os.mbed.com/docs/v5.11/reference/index.html> - last visit: Feb. 2019.
- [11] Scott Chacon and Ben Straub. *Pro Git - Everything you need to know about Git*. Apress®, second edition, 2018. Published at <https://git-scm.com/book/en/v2>.
- [12] Lewin Edwards. *Embedded System Design on a Shoestring: Achieving High Performance with a Limited Budget (Embedded Technology)*. Newnes, 2003.
- [13] Liviu Ionescu. *GNU MCU Eclipse - A family of Eclipse CDT extensions and tools for GNU ARM & RISC-V development*, 2018. Published at <https://gnu-mcu-eclipse.github.io> - last visit: Nov. 2018.
- [14] David Kalinsky. *Context Switch*, 2001. Published at <https://www.embedded.com/design/prototyping-and-development/4023300/Context-Switch> - last visit: Jan. 2019.
- [15] Keil. *Cortex Microcontroller Software Interface Standard - version 5.4.0*, 2018. Published at <http://www.keil.com/pack/doc/CMSIS/General/html/index.html> - last visit: Nov. 2018.
- [16] Vikash Kumar. *Compiling a C program - Behind the scenes*, 2017. Published at <https://www.geeksforgeeks.org/compiling-a-c-program-behind-the-scenes/> - last visit: Dec. 2018.
- [17] ARM Mbed. *Understanding the different types of BLE Beacons - iBeacon Data Spec*, 2015 (last updated). Published at <https://developer.apple.com/ibeacon/Getting-Started-with-iBeacon.pdf>.

- [18] Oracle. *Java SE Development Kit 8 Downloads*, 2018. Published at <https://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html> - last visit: Oct. 2018.
- [19] Petar Popovski, Hiroyuki Yomo, and Ramjee Prasad. *Strategies for adaptive frequency hopping in the unlicensed bands*, 2006. Published at <https://ieeexplore.ieee.org/document/4052302>.
- [20] Miro Samek. *Building Bare-Metal ARM Systems with GNU*, 2007. Published (in ten parts) at <https://www.embedded.com/development/mcus-processors-and-socs> - last visit: Nov. 2018.
- [21] Bluetooth SIG. *Bluetooth Core Specification*, 5.0 edition, 2016. Published at <https://www.bluetooth.com/specifications/bluetooth-core-specification> - last visit: Dec. 2018.
- [22] Bluetooth SIG. *Bluetooth Technology - The global standard for communication*, 2018. Published at <https://www.bluetooth.com/bluetooth-technology> - last visit: Dec. 2018.
- [23] Bluetooth SIG. *Radio Versions - The right radio for the right job*, 2019. <https://www.bluetooth.com/bluetooth-technology/radio-versions> - last visit: Dec. 2018.
- [24] Bluetooth SIG. *Topology Options - Devices need multiple ways to connect*, 2019. <https://www.bluetooth.com/bluetooth-technology/topology-options> - last visit: Dec. 2018.
- [25] Richard M. Stallman, Roland Pesch, and Stan Shebs. *Debugging with GDB*, 2018. Published at <https://www.gnu.org/software/gdb/documentation> - last visit: Nov. 2018.
- [26] Richard M. Stallman and the GCC Developer Community. *Using the GNU Compiler Collection - For GCC version 8.2.0*, 2018. Published at <https://gcc.gnu.org/onlinedocs/gcc-8.2.0/gcc.pdf>.

- [27] STMicroelectronics. *BlueNRG-1 and BlueNRG-2 low power modes*, 2018. [AN4820]
- Published at https://www.st.com/content/ccc/resource/technical/document/application_note/group0/17/f2/d8/23/03/01/47/a9/DM00263007/files/DM00263007.pdf/jcr:content/translations/en.DM00263007.pdf.
- [28] STMicroelectronics. *BlueNRG-1, BlueNRG-2 BLE stack v2.x programming guidelines*, 2018.
[PM0257] - Published at https://www.st.com/content/ccc/resource/technical/document/programming_manual/group0/03/12/05/a4/84/de/47/35/DM00294449/files/DM00294449.pdf/jcr:content/translations/en.DM00294449.pdf.
- [29] STMicroelectronics. *BlueNRG-1, BlueNRG-2 development kits*, 2018. [UM2071]
- Published at https://www.st.com/content/ccc/resource/technical/document/user_manual/group0/a3/3a/74/0f/5d/89/44/3d/DM00298232/files/DM00298232.pdf/jcr:content/translations/en.DM00298232.pdf.
- [30] STMicroelectronics. *BlueNRG-2 Datasheet - Bluetooth® low energy wireless system-on-chip*, 2018. [DS12166] - Published at <https://www.st.com/resource/en/datasheet/bluenrg-2.pdf>.
- [31] STMicroelectronics. *BlueNRG DK (Development Kit)*, 2018. <https://www.st.com/en/embedded-software/stsw-bluenrg1-dk.html> - last visit: Nov. 2018.
- [32] STMicroelectronics. *BlueNRG GUI SW package*, 2018. [UM2058] - Published at https://www.st.com/content/ccc/resource/technical/document/user_manual/group0/b6/e0/6e/ef/bd/ed/4a/62/DM00286976/files/DM00286976.pdf/jcr:content/translations/en.DM00286976.pdf.
- [33] STMicroelectronics. *STM32 Nucleo expansion board for power consumption measurement*, 2018. [UM2243] - Published at <https://www.st.com/content/ccc/>

resource/technical/document/user_manual/group0/f7/ff/95/ce/29/53/49/98/DM00406577/files/DM00406577.pdf/jcr:content/translations/en.DM00406577.pdf.

- [34] STMicroelectronics. *STM32CubeMonitor-Power software tool for power and ultra-low-power measurements*, 2018. [UM2202] - Published at https://www.st.com/content/ccc/resource/technical/document/user_manual/group0/6f/4e/e7/1d/9b/e2/46/1b/DM00386264/files/DM00386264.pdf/jcr:content/translations/en.DM00386264.pdf.
- [35] Kevin Townsend, Carles Cufi, Akiba, and Robert Davidson. *Getting Started with Bluetooth Low Energy: Tools and Techniques for Low-Power Networking*. O'Reilly Media, first edition, 2014.
- [36] Zheng Wang and Michael O'Boyle. *Machine Learning in Compiler Optimisation*, 2018. Published at <https://ieeexplore.ieee.org/document/8357388>.
- [37] Joseph Yiu. *The Definitive Guide to ARM® Cortex™-M0 and Cortex-M0+ Processors*. Newnes - Elsevier, second edition, 2017.

Giunto alla fine del mio percorso di studi desidero rivolgere un sentito ringraziamento a tutti coloro che hanno contribuito al raggiungimento di questo importante traguardo.

Prima di tutto ringrazio i docenti del Corso di Laurea Magistrale in Ingegneria Elettronica del Politecnico di Torino. Seguire le loro lezioni, seminari, laboratori ed ascoltare i loro consigli è stato un privilegio ed un piacere e quanto mi lasciano in dote sarà fondamentale per il prosieguo della mia vita, non solo quella professionale.

Un ringraziamento speciale va al mio Relatore, il Professor Maurizio Martina, per la sua preziosa guida in questo periodo di tesi, la sua disponibilità e pazienza, e soprattutto per avermi supportato e incoraggiato nell'affrontare quest'esperienza in azienda.

Ringrazio tutto lo staff di STMicroelectronics Lecce, per avermi reso il luogo di lavoro accogliente come una seconda casa: da ognuno ho percepito quotidianamente, fin dal primo incontro, l'affetto prima della stima. Tra tutti Gianmarino (responsabile di sede) per avermi arricchito con la sua enorme esperienza e i suoi consigli, Antonio (correlatore), per tutto il supporto e per aver creduto in me e nella mia capacità di sviluppare questo progetto sin dal primo incontro e a Riccardo, che con estrema gentilezza in più di un'occasione ha messo da parte addirittura i suoi impegni per fornire preziosi chiarimenti ai miei quesiti.

Un ringraziamento va anche ai miei colleghi di università, in particolare Giuseppe e Guido, con cui ho condiviso tanti momenti divertenti e le parti più faticose di questo percorso, e ci tengo ora a condividere la soddisfazione più grande.

Grazie a tutti i miei amici ed in particolare ad Alberto, Gabriele, Federico, Daniele, Michele, Francesco e Mino, per aver condiviso con me questi meravigliosi anni.

Rivolgo infine un grazie di cuore, il più grande e più importante, alla mia famiglia, per aver creduto in me, per avermi incitato e incoraggiato nei momenti di difficoltà. Il raggiungimento di questo traguardo è soprattutto merito dei miei familiari, che mi hanno dato le possibilità, la fiducia e i mezzi necessari ad affrontare il percorso. Grazie, perché senza di voi non avrei potuto vivere questa magnifica avventura.

Ad maiora!

Antonio Orlando

