EM9305 SoC
Software Implementer's Guide

Product Family:	em|bleu
Part Number:	EM9305
Keywords:	EM9305, SOC, SDK, DVK, Software, Development, Implementer, Guide, User manual, API

DOCUMENT HISTORY

Release	Author/Contributor	Date	Description
1.0	Christophe Ducommun Kevin Bernard Charles Ingels	15.09.2023	First official release
1.1	Charles Ingels	23.10.2023	Completed §15.2 with CRC calculation. Completed interruptions behavior in §10.1.1 Added maximum interruption latency in §10.1.2 Completed description of interruption source in Table 10-2
1.2	Charles Ingels	23.11.2023	Added the extended application tutorial (§17.2) Added reference to em|bleu Bluetooth LE Application Developer's Guide in §2. Added §7.4 related to weak function description and override. Added details on protocol timer explanations in §11.2. Started adding a "get started" on how to implement a simple Bluetooth LE beacon sample application in §17.2. Added §10.1.4 related to interrupt priority management. Added §10.2 related to exception management.
1.3	Charles Ingels	19.12.2023	Added information on how to switch to deep sleep mode in §12.3. Removing references to design iterations except for the screenshots and the application examples. Added §9.7 introducing the RTC. In §9.9, added limitations on SPI slave library version with and w/o DMA. Removed §12.3 Added §17 on em|bleu Bluetooth LE stack Added the em|bleu logo Refactored complete list of SDK targets in §6.3 Added an information on the default build target Refactored Figure 6-4 Added Table 8-6 Completed §12.3 with extra details Updated latest EM-Core version Updated EM-Core description in Table 7-4 Updated DS and FS versions in reference documents
1.4	Charles Ingels	17.01.2024	§7.7 Updated EM-Core version to 3.4.0

	Thomas Wolfgang		Updated Table 19-6 and Table 19-7 Updated list of targets in §6.3 Figure 6-3, Figure 6-4, Figure 6-9, Figure 6-11 and Table 6-3 updated to match with build process §6 updated to match with latest build process modification Added list of Bluetooth LE libraries in Table 17-1 Updated PAwR Library Descriptions in Table 17-1 Added firmware update facilities notification in §14 Updated Figure 17-1 Updated Figure 17-3 Completed Table 19-4 with some audio profile acronyms
1.5	Charles Ingels Kevyn Kelso	29.02.2024	Updated Table 19-6 and Table 19-7 (information pages with temperature indicator calibration data) Added Table 10-5 with list of supported exceptions. Unsupported exceptions removed. Added §17.3 related to bonding information database Adding §19.6 related to assertions management for debugging Updated Appendix 6 to ROM v3.0 Introducing firmware update process in §14.
1.6	Charles Ingels	29.03.2024	In §10.2, replaced "_Interrupt" keyword by "_Exception" for exception handler management. In §16.4, added more explanations on the infinite loop at the end of the `NVM_ApplicationEntry()` function.
1.7	Charles Ingels	29.04.2024	Figure 8-2 Improved explanations on how the memory manager works. In §10.2, adding exception examples on integer and floating-point division by zero. §10 reworked. Split into interruptions and exceptions parts. §10.2.2.3 completed. §10.2.4 added. §7.7 Updated EM-Core version to 3.5.0
1.8	Charles Ingels	08.07.2024	Refactored the sample application structure description in §7.5, §7.6 and 7.7. In Table 17-1, added following new EMB configurations: • Peripheral legacy • Peripheral with advertising extensions • Central with advertising extensions In Table 17-1, added the following EMB library: • Controller with ISO

			In Table 2-1, fixed EM-Core standard version application note version. In Table 4-1, set DI03 (ROM v1.0) to deprecated. Added description to access to `gPML_Config` structure in §9.5. Change EM-Core version in §7.7.
2.0	Charles Ingels	08.10.2024	This version describes the new common and consistent way of writing a sample application that is not compliant anymore with the previous description. Completed §7.4 on QP/C weak functions that can be overloaded by the user. Chapters §7.5, §7.6 and §7.7 completed with additional information on the description of a typical sample application based on the new application structure. Table 7-3 updated accordingly. Exception EV_ProtVHandler added in Table 10-5. Exception numbers added in Table 10-5. Completed Table 10-3 with interruptions description details. Added §9.3.4 related to GPIO interrupt management. Complete refactoring of §16.4 based on the new sample application structure. Refactored §9.3. Refactored example provided in §16.4.6. Fixed broken link in page 118.
2.1	Charles Ingels	05.12.2024	Added description of `_Interrupt` keyword in §10.1. Added description of `_Exception` keyword in §10.2. Completed §9.3 with code snippet to configure GPIO6 and GPIO7 with UART function. Completed glossary in §Appendix 1 Completed §10.1.2 on interruption management Added a ToDo statement in §10.1.4 Added reference to ARC timers 0 and 1 in Table 10-3. Introduced *unitimer* term in §11.3 Added Appendix 7 listing free device resources that can be used by the end user application. Updated Figure 16-2. Updated basic sample application in §16.4 Updated interruption critical section in §10.1.4 Added §11.4 related to timer #0 and timer #1. Added reference to [9] Solved *To Be Issued* in Table 10-3 (QDEC, RC Calibration, Universal timers 2 & 3, I2S, USB)

			Added §1.1 and §1.2 Minor improvements and typo fix in §10.1.3 and §10.1.4

Table of Contents

INDEX OF TABLES

INDEX OF FIGURES

1. INTRODUCTION

The EM9305 SoC represents the new, tiny, ultra-low power and high-performance Bluetooth 5.4 low energy chip.

Coming in QFN and CSP packages, it combines high performance, low power and very small size. It is well suited for embedded applications where size and power consumption are the biggest constraints.

It combines an ARC EM7D 32-bit CPU running at 48MHz with a dedicated DSP for floating point optimized operations.

It also provides several means for communicating with the external world: SPI, I^{2}C, I^{2}S, UART, Bluetooth LE, GPIO, USB.

Including a radio module as well, Bluetooth LE communication can be achieved by relying on the em|bleu software stack working on top of this device and provided as third-party software.

For an end customer to be able to use the chip by developing their own application, EM provides an SDK along with documentation and various examples on how to use the available functionalities.

This implementer's guide is also part of the provided documentation, and its goal is to help an end user get started with the software part of the EM9305 chip.

The SDK provides several software components that the developer selects, configures and integrates into their application depending on their needs. Beyond the device drivers that give access to on-chip hardware functionalities through a high-level abstracted API, the SDK also provides a complete Bluetooth LE stack made of a link layer, a host layer and several profiles definition.

Third party libraries are also provided in order to achieve specific needs like implementing a real time constrained application by using QP/C which provides a tiny real time development footprint.

Among all chapters, some focus on specific topics that might be of interest to an end user.

The §4 gives a detailed overview on the content of the SDK with some description of each provided component. Thus, it provides an insight of the device's block architecture as a preliminary introduction to the SDK modules.

The §6 explains the build process and how to build and install the provided examples.

The §7 explains how to create a new project either from scratch or from an existing example.

The §8 is dealing with the memory organization, in RAM and in NVM as well. It describes RAM retention process that deals with important concerns when it comes to dealing with power-saving.

The §9 contains description of all drivers provided within the SDK. So depending on the end user interest, focus can be put on the relevant ones.

The §10 is also an important part that deals with interruptions and exception management.

The §11 might be of a high interest to understand how timers work, which is useful for understanding the core of the power saving process in which the device is put into sleep mode during small periods of time. It is completed by explanations on the various available sleep modes in §12.

The §13 contains information related to how to lock memory parts and functionalities from a user standpoint. This can be useful in case a user needs to prevent the non-volatile memory from being erased, for example to prevent the device being tampered.

The §15 provides advices on what should be considered for optimization and better performances, while §16 and §17 expose how to write application software using QP/C and the Bluetooth LE stack in a comprehensive step by step process.

As a complement of this document, more technical information on the EM9305 device can be found in documents [1] and [2].

1.1 HOW TO READ THIS DOCUMENT

This document is intended to be used by software engineers working on the development of an application software running on the EM9305 device and using some or all the features provided by EM through the software development kit.

Since this document is contained in the SDK, its reading shall be combined with the HTML SDK's documentation which should be the first entry point. Then, depending on the requested information, the reader can complete its reading throughout this document which contains valuable technical information to go further.

1.2 ICONS

Some icons are used throughout the document. Their meaning is described hereafter in Table 1-1.

Icon	description
ⓘ	This icon is used to introduce an extra information. It can be safely ignored.
✋	This icon indicates an important information that the end user should be aware of.
⚠️	This icon indicates a very important information that the end user shall be aware of. If the highlighted instruction is not respected, then this might end up in having nonfunctional device.
💿	This icon indicates that the reader should refer to resource provided in the SDK. It is usually the HTML documentation that can be read using a web browser.

Table 1-1: Description of used icons throughout the document

2. REFERENCE DOCUMENTS

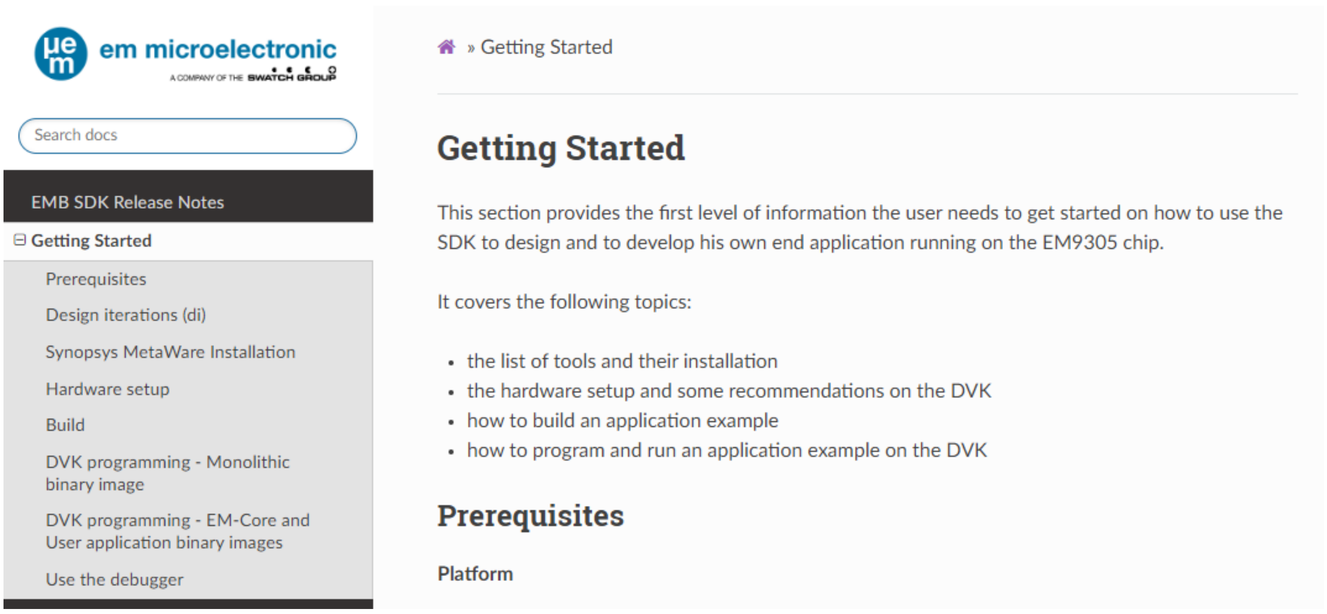
Reference	Title	Version
[1]	EM9305-DS, Multi-Protocol Bluetooth 5.4 SoC and Companion Chip with Flash Data Sheet	4.5.1
[2]	Multi-Protocol Bluetooth 5.3 SoC and Companion Chip with Flash Fact Sheet	4.5.1
[3]	IRAM0 remap to DRAM application note	
[4]	EM9305 Security lock bits	1.0
[5]	EM-Core Standard Model	1.1, Jan 2024
[6]	BLEngine SDK documentation (<SDK>/doc/blengine/index.html)	1.9
[7]	Quantum Leap – Application Note – Getting Started with QP/C	Rev H Oct 2021
[8]	em|bleu Bluetooth LE Application Developer's Guide (SDK)	-
[9]	DesignWare ARCv2 ISA Programmer's Reference Manual for ARC EM Processors	Version 5755-103, July 2022

Table 2-1: Reference documents

3. DEVELOPMENT PLATFORM PREREQUISITE

The development platform prerequisite is listed in the Getting started HTML documentation provided in the SDK (file `SDK_GettingStarted.html`). Please refer to this documentation for the latest technical information.

The Figure 3-1 shows a snapshot of the Getting started documentation provided in the SDK. It is given here as an example and might evolve over time (tools version …).


em microelectronic
A COMPANY OF THE SWATCH GROUP
Search docs
EMB SDK Release Notes
⊟ Getting Started
Prerequisites
Design iterations (di)
Synopsys MetaWare Installation
Hardware setup
Build
DVK programming - Monolithic binary image
DVK programming - EM-Core and User application binary images
Use the debugger
» Getting Started

Getting Started

This section provides the first level of information the user needs to get started on how to use the SDK to design and to develop his own end application running on the EM9305 chip.

It covers the following topics:

- the list of tools and their installation
- the hardware setup and some recommendations on the DVK
- how to build an application example
- how to program and run an application example on the DVK

Prerequisites

Platform

Figure 3-1: Extract from the SDK's Getting started documentation

4. DEVICE OVERVIEW

4.1 DEVICE GENERAL OVERVIEW

The document [1] introduces the EM9305 device description with a general overview along with its features and capabilities. Refer to this document for any concern related to hardware device features.

The EM9305 device is being improved over time which leads to the delivery of new DI with new features. Each DI includes a ROM code which can be same as the one inside the previous DI or a new ROM version when bug fixes and/or new features are made available.

The Table 4-1 gives the ROM version that comes along a specific DI.

DI	ROM	
01	1.0	Deprecated
02	1.0	Deprecated
03	1.0	Deprecated
04	2.0	Not supported in future releases
05	3.0	Default target for build process

Table 4-1 : Dis versus ROM versions

From a software point of view, an SDK supports Dis up to the latest available. However, it might happen that old Dis are not supported anymore. In such a case, this is clearly highlighted in the SDK's release note.

4.2 DEVICE START-UP PROCESS

When the EM9305 device is powered up, or if it resumes from a software reset or a sleep period, it executes the start-up sequence given in Figure 4-1. This high-level start-up process description shows the decision tree used to define which application to start depending on the following status:

- the requested application to start
- the presence and validity of a firmware image

The diagram illustrates the launch of an end user application by revealing the underlying start-up logic. This provides transparency to the end user, allowing them to understand how the device initiates.

However, it is important to note that this diagram hides the initial bootstrapping process that occurs prior to the execution of the logic shown in Figure 4-1. This bootstrapping process is embedded within the ROM and its specific operations are explained in Figure 4-2.

But first, let's have a detailed look at the start-up decision tree shown in Figure 4-1. This figure shows paths to rectangular boxes that represent a specific firmware to start. According to it, there can be up to three firmware images stored in the device but only one will end-up to be started depending on this decision three. These firmware images are the following:

- EM-Core (refer to application note [5] for detailed information)
- Customer bootloader
- Customer application

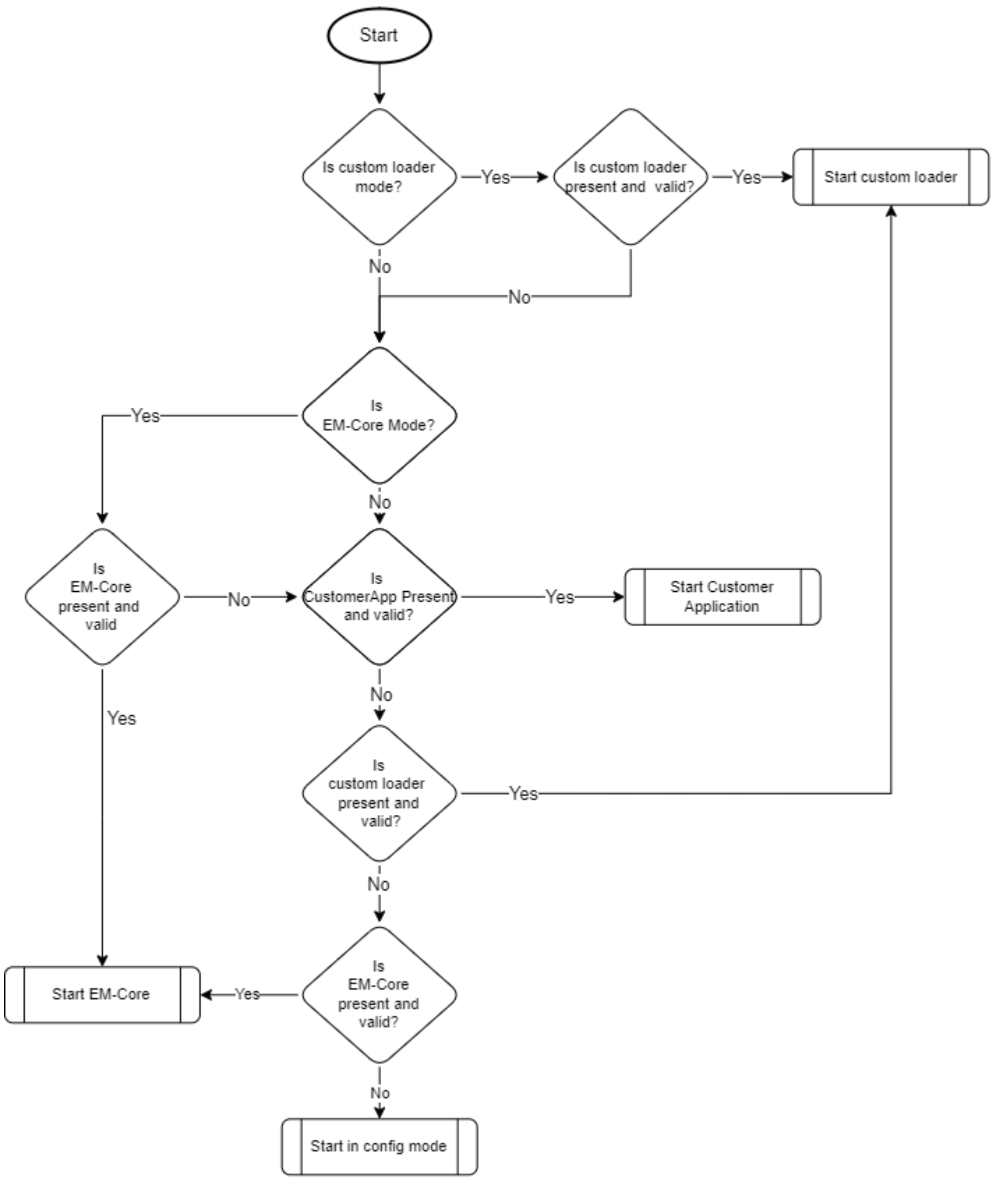

Start
Is custom loader mode?
Yes
Is custom loader present and valid?
Yes
Start custom loader
No
No
Is EM-Core Mode?
Yes
No
Is EM-Core present and valid
No
Is CustomerApp Present and valid?
Yes
Start Customer Application
Yes
No
Is custom loader present and valid?
Yes
No
Start EM-Core
Yes
Is EM-Core present and valid?
No
Start in config mode

Figure 4-1: NVM Bootloader start-up decision diagram

The final goal of the start-up process is to start the end user application that resides in non-volatile memory. However, there is a possibility to start a secondary end user application which can be used as a bootloader from an end user perspective. This can be the case where the end user needs a dedicated application to execute some actions rarely done, like to do a firmware upgrade or to do a first application initialization for example. Otherwise, most of the time, the end user main application will be the one to be executed.

This diagrams also highlights the fact that there is a so-called "EM-Core" firmware stored in the non-volatile memory and installed by default[1]. The device can end up executing the small application it contains in cases where some conditions are not fulfilled. In some cases, it can be executed mainly for testing purpose without having any end user application installed. But in the nominal device start-up when an end user application is present in NVM, this EM-Core application is only started when explicitly requested.

[1] Depending on the end user requested configuration, this EM-Core firmware image might not be present. But without specific request, it is present by default.

At the end, if the non-volatile memory is empty, which means neither an end user application nor EM-Core are installed (or maybe installed but invalid), then the device's behavior is to fall back to the configuration mode. This code is inside the ROM which means that it is always present and cannot be altered.

The Figure 4-1 logic execution is achieved by a small piece of software called the `NVM Bootloader` that is stored at the beginning of the NVM. It is the entry point after the ROM boot sequence has completed its job.

As it can be seen on this figure, this NVM-Bootloader is able to detect which firmware is installed in the NVM and to verify their integrity by using a CRC check. When the list of firmware is built, then the NVM-Bootloader reads the requested start-up configuration stored in a permanent register and executes the corresponding firmware.

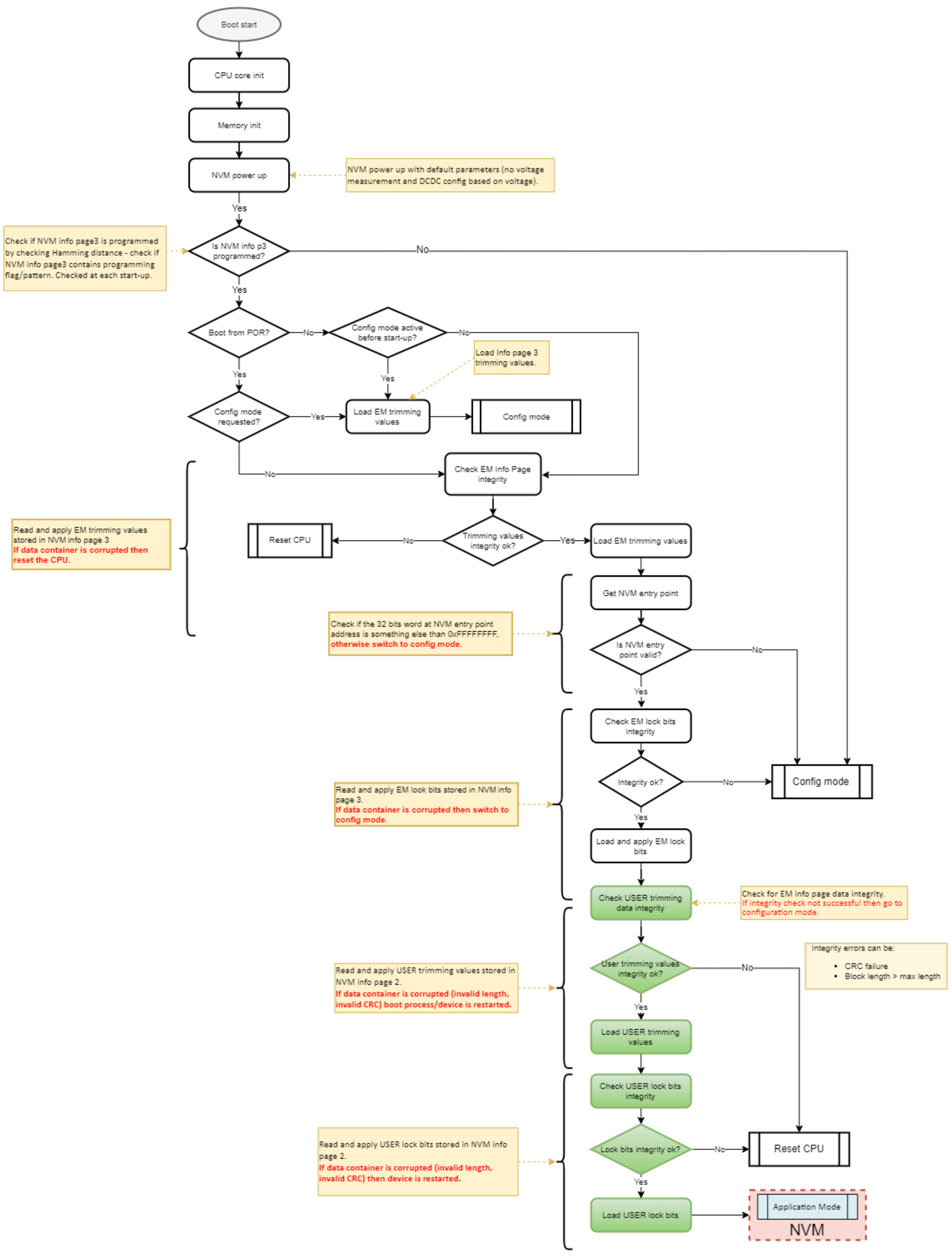
In case this firmware is not present or corrupted, then the NVM-Bootloader follows the logic diagram to start-up another available firmware with the following priorities:

1. End user application
2. Custom user bootloader
3. EM-Core application
4. Configuration mode

Consequently, the basic start-up procedure would be to start-up the configuration mode located in the ROM.

Now that we have a high-level overview of the global start-up process, let's dive in the very early start-up process that is executed by the ROM.

The Figure 4-2 dives a bit more into the details of the ROM start-up sequence giving some low level details.


Boot start
CPU core init
Memory init
NVM power up
Yes
Is NVM info p3 programmed?
No
Yes
Boot from POR?
No
Config mode active before start-up?
No
Yes
Yes
Config mode requested?
Yes
Load EM trimming values
Config mode
No
Check EM Info Page integrity
Reset CPU
No
Trimming values integrity ok?
Yes
Load EM trimming values
Get NVM entry point
Is NVM entry point valid?
No
Yes
Check EM lock bits integrity
Integrity ok?
No
Config mode
Yes
Load and apply EM lock bits
Check USER trimming data integrity
User trimming values integrity ok?
No
Yes
Load USER trimming values
Check USER lock bits integrity
Lock bits integrity ok?
No
Reset CPU
Yes
Load USER lock bits
Application Mode
NVM

NVM power up with default parameters (no voltage measurement and DCDC config based on voltage).

Check if NVM info page3 is programmed by checking Hamming distance - check if NVM info page3 contains programming flag/pattern. Checked at each start-up.

Load Info page 3 trimming values.

Read and apply EM trimming values stored in NVM info page 3
If data container is corrupted then reset the CPU.

Check if the 32 bits word at NVM entry point address is something else than 0xFFFFFFFF, **otherwise switch to config mode.**

Read and apply EM lock bits stored in NVM info page 3.
If data container is corrupted then switch to config mode.

Check for EM info page data integrity.
If integrity check not successful then go to configuration mode.

Integrity errors can be:
- CRC failure
- Block length > max length

Read and apply USER trimming values stored in NVM info page 2.
If data container is corrupted (invalid length, invalid CRC) boot process/device is restarted.

Read and apply USER lock bits stored in NVM info page 2.
If data container is corrupted (invalid length, invalid CRC) then device is restarted.

Figure 4-2: Detailed ROM boot sequence

This figure shows some important steps regarding the device configuration stored in both the EM (information page 3) and user (information page 2) and how it is applied.

The EM information page contains trimming values that are used to configure the device internal Ips like the NVM access time for example. It might also contain a lock bits configuration (see §13 for more details) and EM authentication public key along with some other information. The detailed content of this page is given in Appendix 3 .

Usually, the EM information page is locked before a device is sent to the customer premises so its content cannot be modified afterward.

Each data stored in the EM information page is protected against any corruption by a computed CRC which is checked during the start-up process before actually applying the corresponding values. In case a CRC check fails and according to Figure 4-2, a device reset is triggered, and the trimming values are not applied. This behavior can be easily explained by the fact that the device might not operate properly if the trimming values are not configured in the right way. In this case, the only way to be able to use the device is to enforce starting-up the device in configuration mode.

The user information page is reserved for the end user to finalize the device configuration regarding some trimming data, lock bits, authentication keys or MAC address. By default, this page is not locked so it can be erased and written. Its content structure is imposed by EM but it is up to the end user to fill in its content with relevant values.

Note that a reset will not occur if the user information page is either empty or only partially filled. In such cases, the device will start up normally, allowing the end user to launch their application even if the user information page is empty. However, if the content of the user information page is corrupted, a CPU reset will be triggered, similar to the EM information page.

When this start-up sequence is completed, the ROM code jumps at the beginning of the NVM with the intent to start executing the end user application.

At this stage, the EM9305 device default configuration is given by Table 4-2. This configuration can be changed in the end user application like switching the transport layer to use the UART instead of the SPI for example or reconfiguring the RAM0 block as a data RAM instead of an instruction RAM.

Sub-system	Configuration
RAM0	Block configured as instruction RAM (IRAM0) – see Figure 8-2
DRAM	No blocks configured for retention

Table 4-2: Default device configuration at start-up

4.3 EM SYSTEM

The SDK provides a software library that an end user can use to add a transport layer and a set of predefined HCI commands to their application. This software library is called the “EM System” and it is a quick and easy way to be able to add communication functionality with the application using UART, SPI or USB communication bus.

This EM System content is described in [1] so refer to this document for further details on its content, and moreover the list of provided commands.

This Implementer’s Guide covers the EM System library from the software developer’s point of view. This topic’s technical details are addressed in §18 and, in particular, provides explanations on how to use the EM System in an end user application.

4.4 CONFIGURATION MODE

The configuration mode is the most basic mode the device can enter in, assuming certain conditions are met. Some of them are depicted in Figure 4-1 and in Figure 4-2 mainly when there is no other way to start-up any end user application. But in any case, a fresh new device that has never been programmed will directly enter into this mode for its first start-up. It is then possible to finalize the device by programming the requested firmware along with the EM information pages.

However, according to document [1], it is also possible to reach this mode when a 32 kHz signal is applied to GPIO5 during the device start-up process. In this case, the lock bits are not applied, and the device directly enters the configuration mode. It is considered as an open device. This feature allows an end user to program their application in the NVM main area and to write the user information page along with other pages containing application related information (like keys in the key container). However, it is not possible for an end user to erase or write the EM information page.

Once a device is programmed, its regular start-up is to start the end user application. However, in case of issues, switching back to the configuration mode can be seen as a last chance to recover from a desperate situation where the device does not start as expected. In this mode, a subset of commands sent through the SPI bus are available to reconfigure the device in the proper way. This means that in order to operate in this mode, a physical connection to the device's SPI bus is mandatory.

The list of commands supported by the configuration mode is given in Appendix 5 .

4.5 EM-CORE MODE

Running the device into the so called "EM-Core" mode means that the EM-Core application is being executed by the CPU.

This application resides in NVM and its main goal is to provide a subset of commands to be executed by testers at the end of the manufacturing process at EM.

The application note [5] gives a comprehensive description of the EM-Core standard model for a better understanding. However, having this level of knowledge for an end user is not mandatory, since this specific firmware might not be present on devices in the field. It is then up to the end user to decide if they want to keep it or not. However, the end user might actually use this software in case some commands match their device testing needs. The application note [5] gives the list of commands supported by EM-Core.

Since EM-Core comes along with a set of libraries listed in [5], the end user application will have to embed these libraries in case they are needed in the end user application and the EM-Core firmware is removed from the device's non-volatile memory.

5. SDK GETTING STARTED

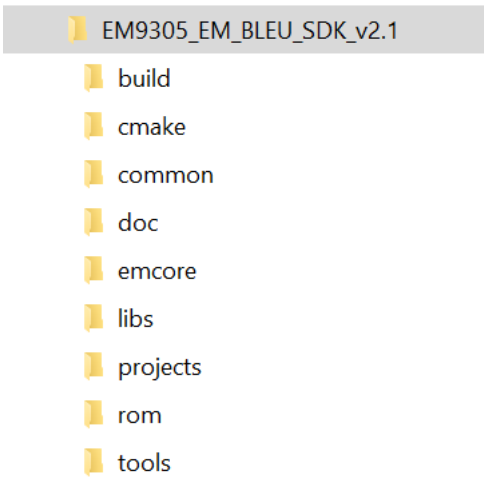
5.1 OVERVIEW

The SDK is a package that contains all libraries, drivers, examples, documentation and tools an end user needs to create, build and test his application.

Prior to using it, it shall be installed on a computer used for development purposes.

There is no Metaware ARC toolchain in the SDK. This toolchain has to be installed separately from the SDK before starting any build process. Thus, it is essential to have the Toolchain installed beforehand.

The Figure 5-1 shows the structure of the SDK's folder content (the SDK version shown in this screenshot is not representative of the actual latest version).



EM9305_EM_BLEU_SDK_v2.1
build
cmake
common
doc
emcore
libs
projects
rom
tools

Figure 5-1: SDK's folder structure used by the CMake process

Folder	Description
build	created by the build process
cmake	contains all Cmake definitions and macros used by the Cmake build process
common	contains EM9305 specific headers with linker scripts and compilation options
doc	contains the SDK's online documentation (HTML files) along with application notes
emcore	contains all EM-Core flavors
libs	contains all the drivers (also called modules or libraries) in binary forms ready to be embedded
projects	contains project examples an end user can consider for initiating his own application
rom	contains all header files and ROM symbol file for embedded calls to functions present in ROM
tools	contains tools and related files to be able to use a DVK (BLEngine is part of it)

Table 5-1: Content description of SDK's folders

To open the SDK's documentation, open the file `SDK.html` in your favourite web browser. You will then get access to all available documentation.

5.2 SDK'S INSTALLATION

5.2.1 Overview

The SDK comes in the form of a Windows self-installation executable.

```
EMB-SDK-X.Y-setup.exe
```

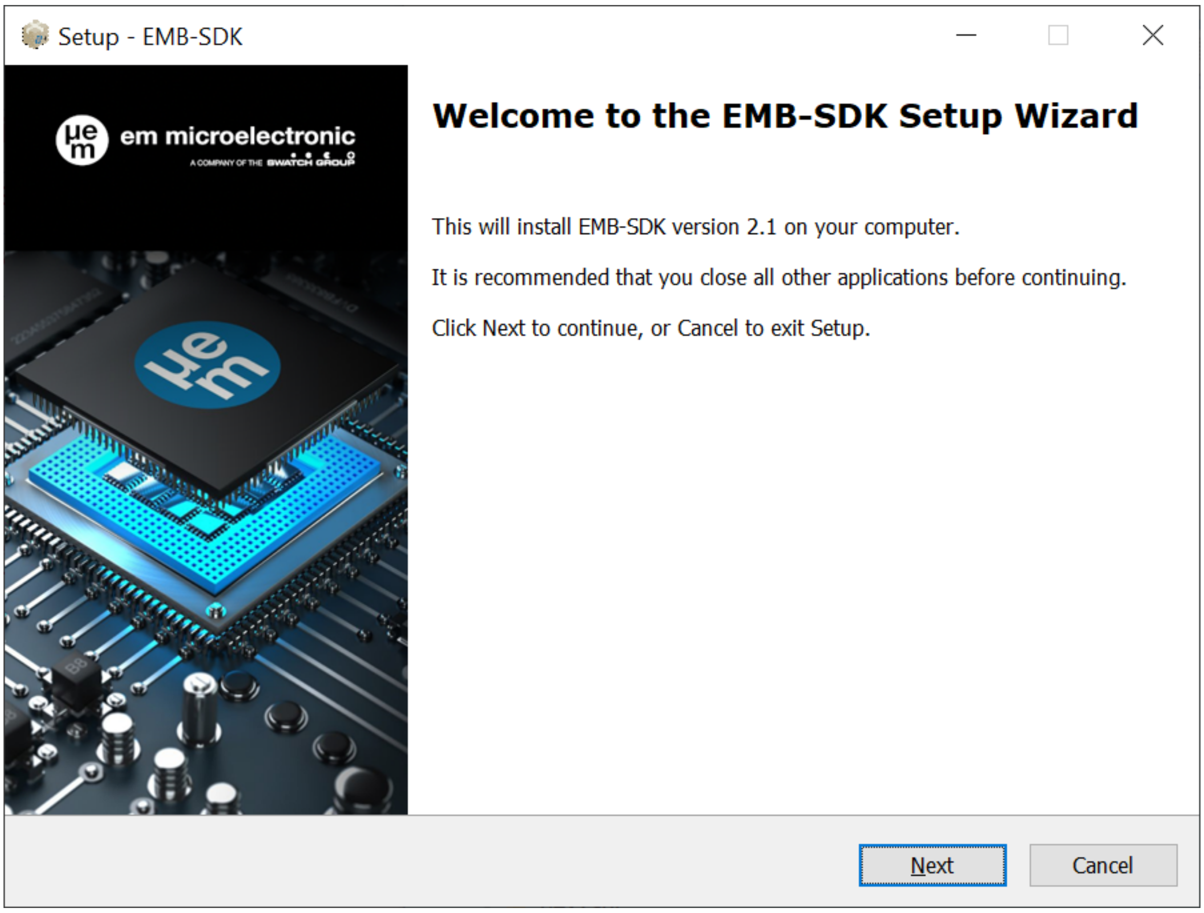
Where X.Y represents the SDK's version. The X number is the major number while the Y number is the minor one.

Between two SDKs version, two different minor numbers mean that these two SDKs are compatible. An application developed with the former version will also work with the new one without any refactoring. However, the most recent version might provide additional functionalities, libraries, documentation…

If the major numbers are not the same, then these two SDKs are not compatible. This can be a change in the API for some functions in some libraries (see the ChangeLog file for details) and if an application actually uses one (or more) of these functions, then it has to be refactored accordingly.

5.2.2 Step 1

To install the SDK, double-click on the EM-SDK-X.Y-setup.exe file. The welcome window opens.



Welcome to the EMB-SDK Setup Wizard

This will install EMB-SDK version 2.1 on your computer.

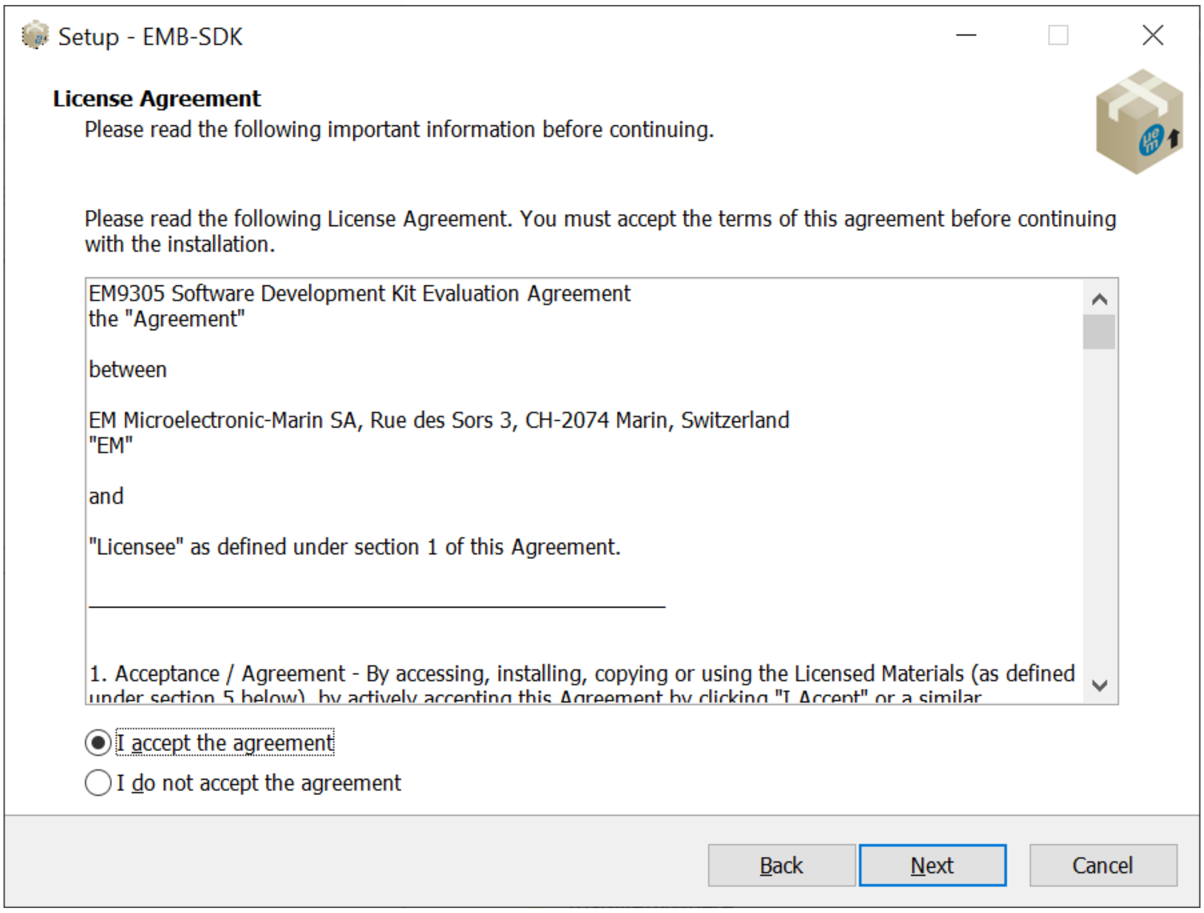
It is recommended that you close all other applications before continuing.

Click Next to continue, or Cancel to exit Setup.

Figure 5-2: SDK's installer welcome window

5.2.3 Step 2

Going further requires mandatory acceptance of the license agreement. After carefully reading the text, click "I accept the agreement", then click the next button. By doing so, you explicitly agree with the license agreement.



License Agreement
Please read the following important information before continuing.

Please read the following License Agreement. You must accept the terms of this agreement before continuing with the installation.

EM9305 Software Development Kit Evaluation Agreement
the "Agreement"

between

EM Microelectronic-Marin SA, Rue des Sors 3, CH-2074 Marin, Switzerland
"EM"

and

"Licensee" as defined under section 1 of this Agreement.

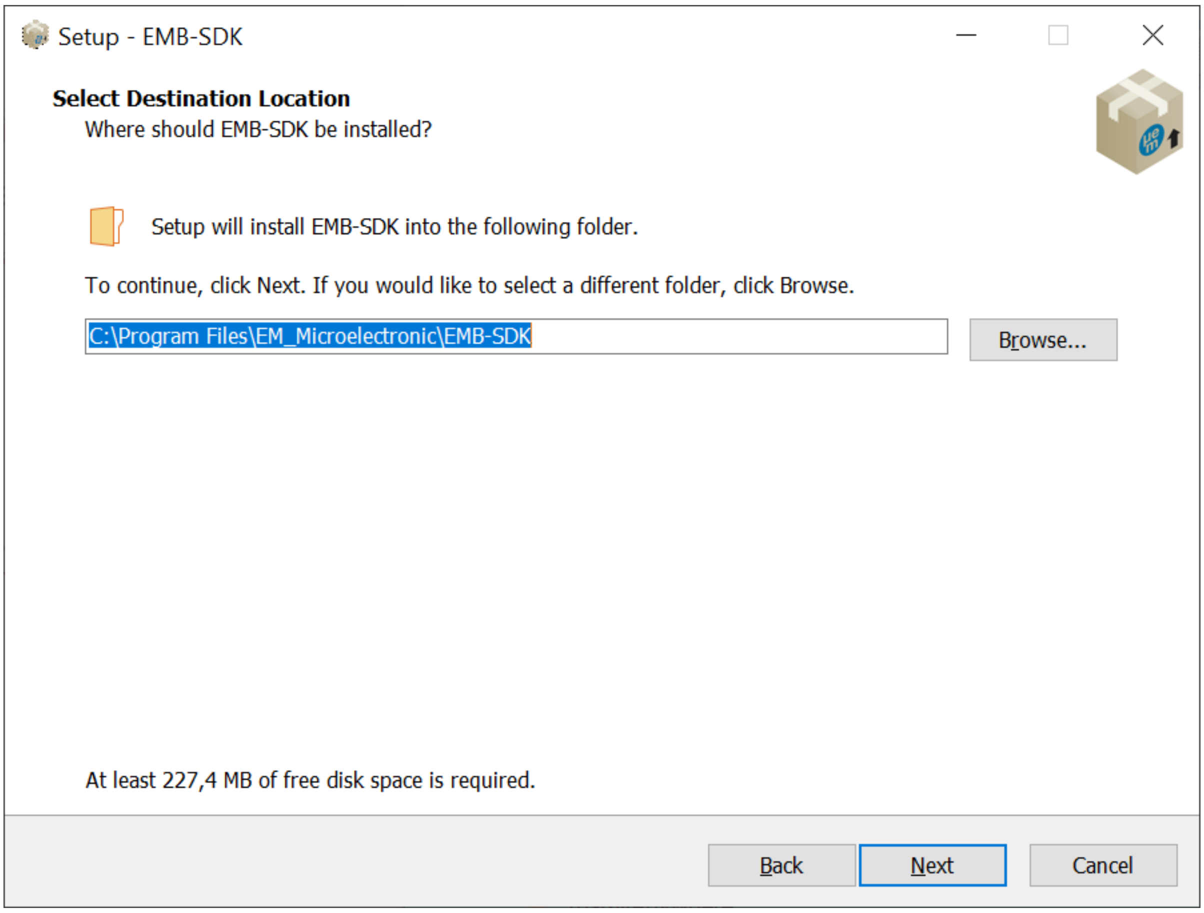
1. Acceptance / Agreement - By accessing, installing, copying or using the Licensed Materials (as defined under section 5 below), by actively accepting this Agreement by clicking "I Accept" or a similar

[✓ I accept the agreement]
I do not accept the agreement

Figure 5-3: SDK’s license agreement

5.2.4 Step 3

Select the destination folder where the SDK should be installed. Note that the targeted location shall be writable since new projects or existing example modification are foreseen.


Setup - EMB-SDK

Select Destination Location

Where should EMB-SDK be installed?

Setup will install EMB-SDK into the following folder.

To continue, click Next. If you would like to select a different folder, click Browse.

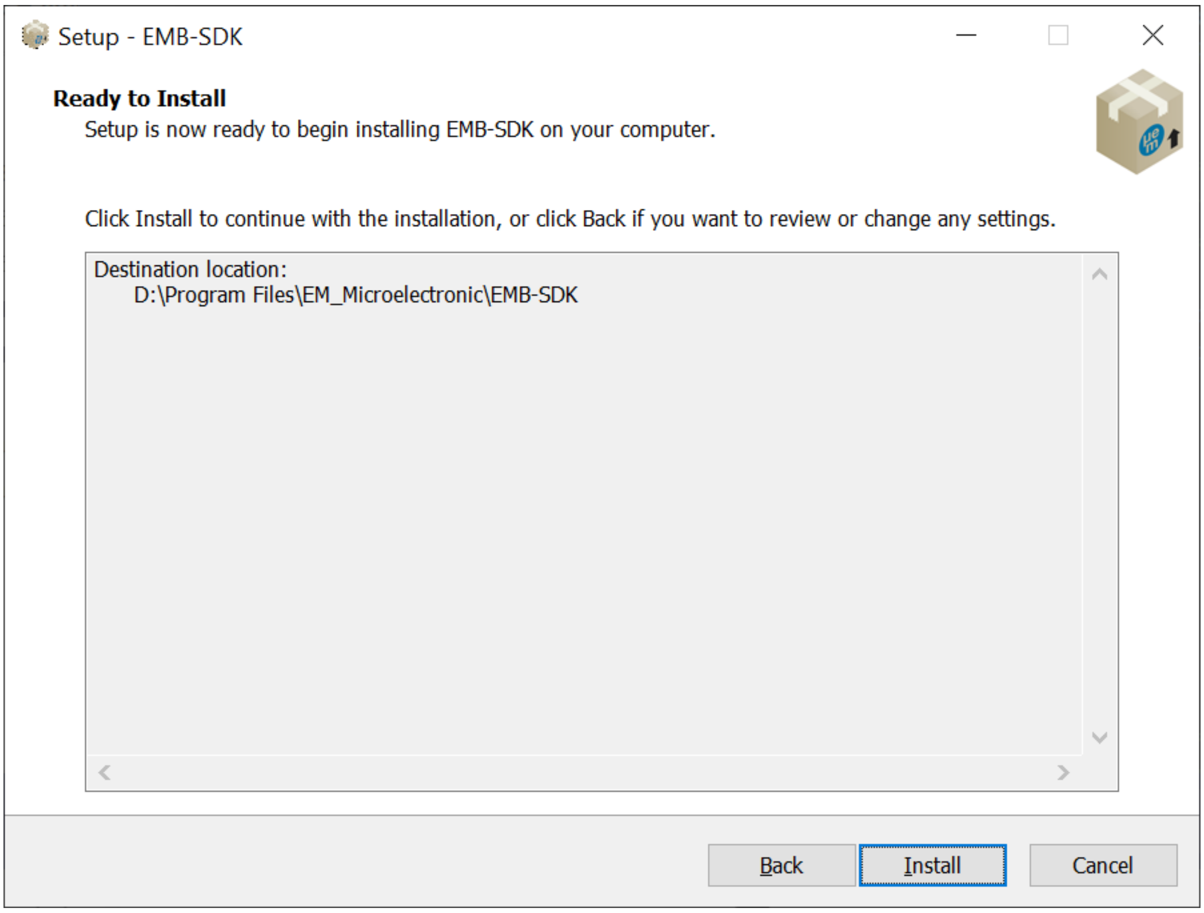
C:\Program Files\EM_Microelectronic\EMB-SDK

At least 227,4 MB of free disk space is required.

Figure 5-4: Select SDK folder installation

5.2.5 Step 4

The next window summarizes information before proceeding with the SDK installation. Just click the next button to actually start the installation.



Ready to Install

Setup is now ready to begin installing EMB-SDK on your computer.

Click Install to continue with the installation, or click Back if you want to review or change any settings.

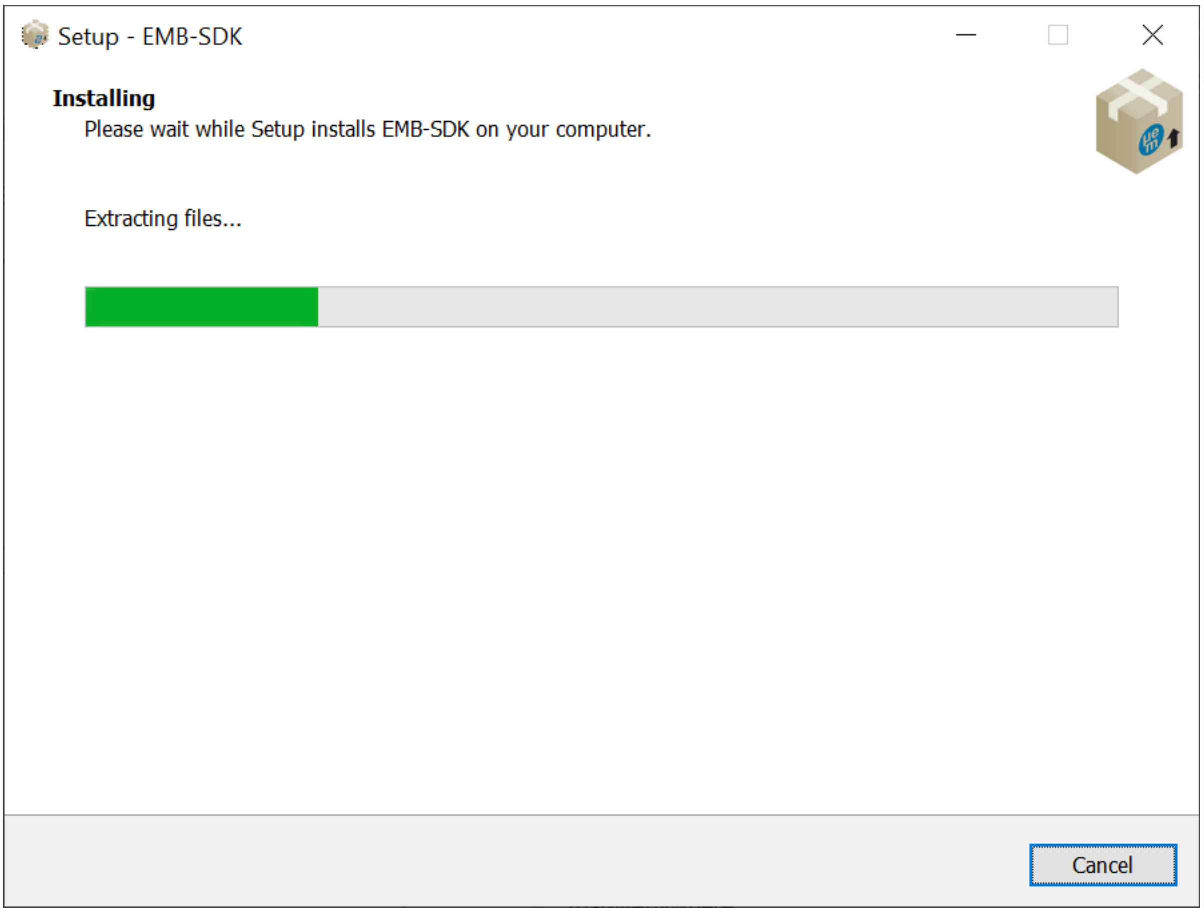
Destination location:
D:\Program Files\EM_Microelectronic\EMB-SDK

Figure 5-5: SDK installation information summary

5.2.6 Step 5

The next window shows the SDK installation progress.

Once done, you can click on the next button.


Setup - EMB-SDK

Installing

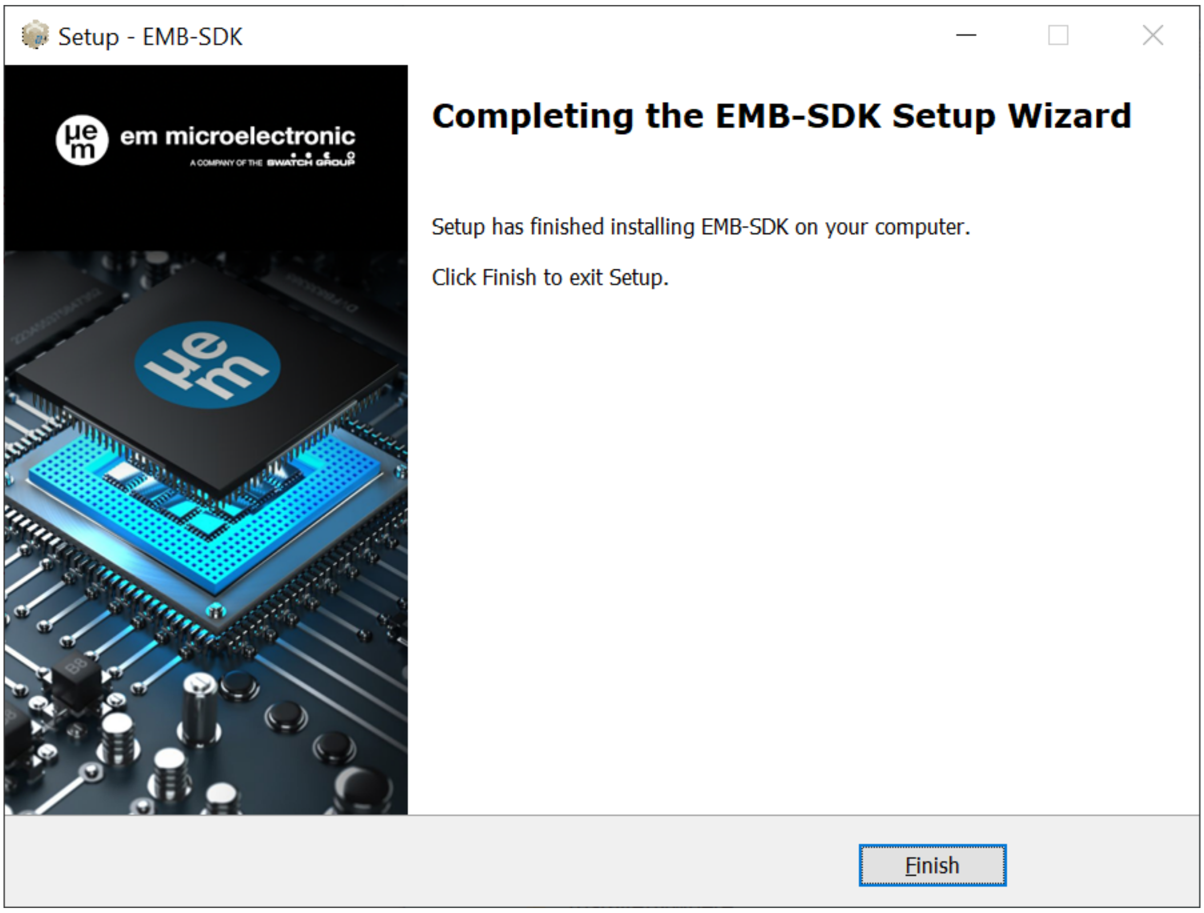
Please wait while Setup installs EMB-SDK on your computer.

Extracting files...

Figure 5-6: SDK installation process progress

5.2.7 Step 6

Once everything has been installed, you can click on the finish button. You are then done with the SDK installation.


Setup - EMB-SDK

Completing the EMB-SDK Setup Wizard

Setup has finished installing EMB-SDK on your computer.

Click Finish to exit Setup.

Figure 5-7: SDK installed

Once done, you can open a terminal and point to your new SDK installation to start playing with it. You now need to get familiar with the build process that is explained in §6.

6. CMAKE BUILD PROCESS

6.1 OVERVIEW

Building an end user application is done by applying a process that relies on the usage of the ’cmake’ tool. While this tool is available on both Windows and on Linux, the required platform to use is Windows.

This process can be used to build the provided examples and it is strongly recommended that it is also used to build the end customer application relying on the folder structure provided in the SDK.

The minimum ’cmake’ version that is required to build any example or application is the 3.10 at the time this implementer’s guide is written. This might evolve over time.

It is also assumed that the end user has a correct installation with the requested tools installed. The Table 6-1 lists the required tools that need to be install on the development platform.

Tool	Version
Synopsys Metaware	⩾ 2019.09 Windows version
cmake	3.10.x

Table 6-1: Required build tools

6.2 GENERAL STRUCTURE

Since the ’cmake’ tool uses the so called `CmakeLists.txt` file, there is one such file at the root of the SDK that acts as the starting point. This file instructs ’cmake’ on how to create the build environment that will be used afterward to build any provided example or to build the end user application.

All modules (or libraries) provided in the SDK under the `<SDK_INSTALLATION_PATH>\libs` folder come with their own `CmakeLists.txt` file that contains relevant instruction on how to integrate them into an example or into the end user application.

For example, the `i2c` module (otherwise called the i2c library) is introduced by the following statement in its own `CmakeLists.txt` file:

```
ADD_LIBRARY(i2c STATIC IMPORTED GLOBAL)
```

This statement will instruct the build process to statically link the `lib_i2c.a` library to the application. Fortunately, this library is provided in the SDK.

And the following statement give the exact location from where the `lib_i2c.a` file is to be imported.

```
SET_TARGET_PROPERTIES(i2c PROPERTIES IMPORTED_LOCATION
                                        “${CMAKE_CURRENT_SOURCE_DIR}/lib_i2c.a”)
```

All modules located under the `<SDK_INSTALLATION_PATH>\projects` folder are application examples that can be built and tested and used as a baseline to create the end user application. The `CmakeLists.txt` file of each application includes the list of libraries it is actually using.

The `CmakeLists.txt` within each of these examples contains all instructions to build the example itself by specifying the following details:

- Specific flags if relevant
- Project name
- The list of libraries to link with (like i2c if needed)
- Where the final executable will reside in memory (iRAM or NVM)

- Where to install the built binary file in memory (at which address)

The i2c example is an application that communicates over the i2c. In its `CmakeLists.txt` file, the list of needed libraries is written like this:

```
SET( ${PROJECT_NAME}_LIBS
     ${NVM_COMMON_LIBS}
     QPC
     unitimer
     uart_dma
     printf
     i2c
)
```

With this statement, the build process links the final application example with the following libraries:

- common libraries (see Appendix 2)
- `lib_QPC.a`[2]
- `lib_unitimer.a`
- `lib_uart_dma.a`
- `lib_printf.a`
- `lib_i2c.a`

Note that there is a subset of common libraries that are always embedded within an application. They are listed through the `NVM_COMMON_LIBS` variable. See Appendix 2 for a detailed description of the common libraries.

All other files used by the `cmake` tool are located in the `<SDK_INSTALL_PATH>\cmake` folder. While looking out at its content might provide a good understanding on how the build process works, it is strongly recommended not to change anything inside this folder.

For a better understanding of the build mechanism, the Figure 6-1 shows the Cmake files structured hierarchy on which the build process has been designed.

The entry point is the highest level `CmakeLists.txt` file at the root of the application source code that includes some of the underneath `.cmake` files.

In turn, these files include underneath `.cmake` files and so on in a recursive way. This is how the Cmake tool will browse all these files following this hierarchical structure.

All the Cmake core files have been gathered within a `cmake` folder which contains a bunch of other files that define variables and macros that are used throughout the build process.

Having a look at these files can give a good understanding on how the process work.

The libs and projects folders content has not been detailed but they contain several folders with one `CmakeLists.txt` file within each.

[2] QP/C is a third party library provided by Quantum Leap

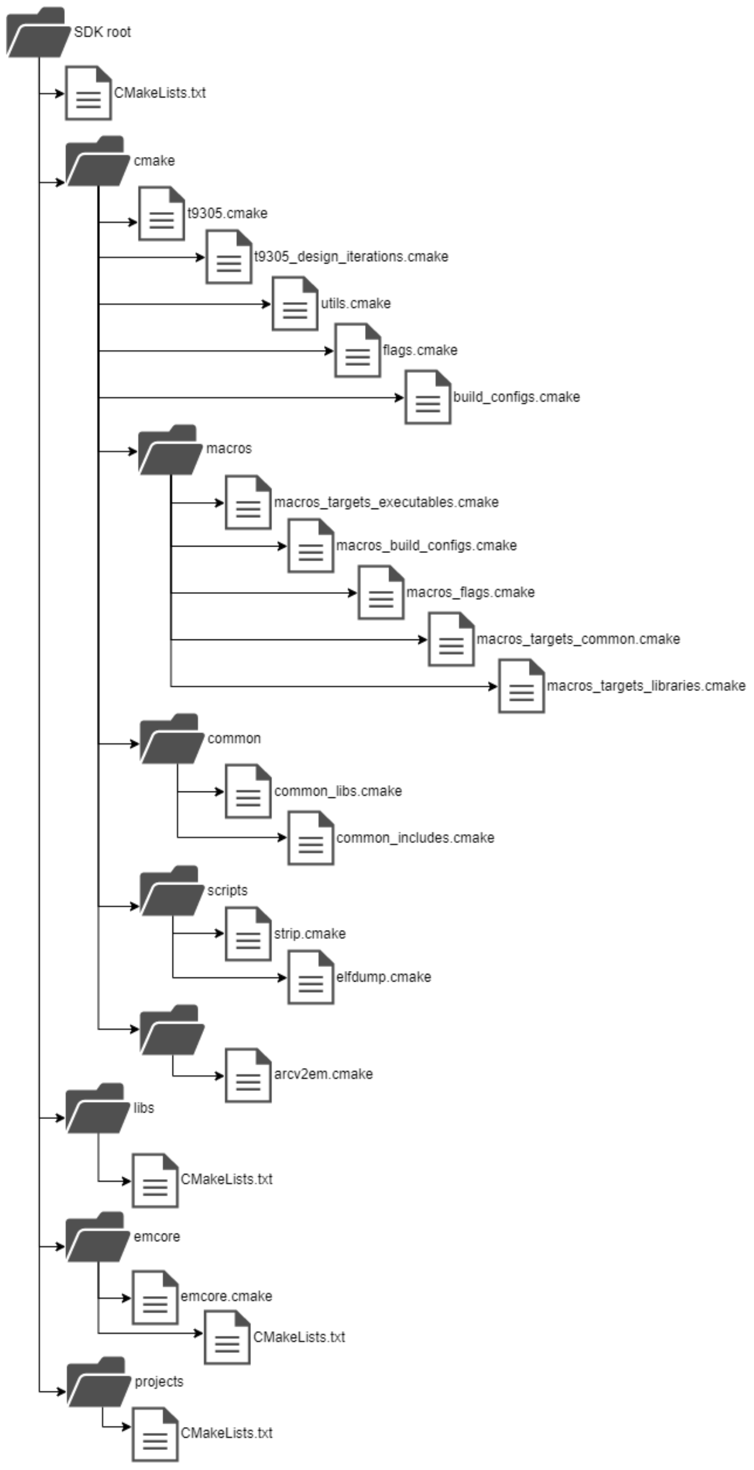


Figure 6-1: CMake folder structure hierarchy

6.3 SDK BUILD ENVIRONMENT INITIALIZATION

After the SDK has been installed, the mandatory step to do is to create the build environment that will be used for building further examples. This step shall be done only once, or if it is needed to restart from a clean SDK.

The SDK provides the `init.bat` script that includes the sequence of commands to be issued in order to create the build environment.

Go to the SDK installation root folder:

```
# cd <SDK_INSTALL_PATH>
```

Run the script from the SDK location:

```
init.bat
```

The details of this script are explained below.

The first step is to define the build folder name (build by default) and to create it. Note that an already existing build folder will be deleted.

```
Set BUILD_ROOT=”.\build”
mkdir %BUILD_ROOT%
```

Then the build environment is created in the `build` folder:

```
# cd %BUILD_ROOT%
# cmake -G”Eclipse CDT4 - Unix Makefiles”
        -D_ECLIPSE_VERSION=8.5
        --Wno-dev
        -DCMAKE_BUILD_TYPE=debug
        -DCMAKE_TOOLCHAIN_FILE=”cmake/toolchains/arcv2em.cmake” ..
```

The options are explained in Table 6-2.

Option	Description
--G”Eclipse CDT4 – Unix Makefiles”	With this option, cmake is instructed to create makefiles along with an Eclipse project file. However, these project files are not actually used this is why they are deleted at the end of the build script.[3]
-D_ECLIPSE_VERSION=8.5	Set Eclipse version for which the project files are created. See note 3.
-Wno-dev	This option tells \`cmake\` to stop generating development warnings. This will prevent \`cmake\` from spamming the console with multiple warning messages which can be safely skipped and that might not be of any interest for the end user.
-DCMAKE_BUILD_TYPE=debug	The environment is configured for building debug versions. The other option is to build for release.
-DCMAKE_TOOLCHAIN_FILE=…	Through the indicated file, the cmake tool will setup the toolchain binaries for the compiler (ccac), the archiver (ccar) and so on through the definition of reserved cmake variables: • CMAKE_FORCE_C_COMPILER • CMAKE_FORCE_CXX_COMPILER • CMAKE_ASM_COMPILER • CMAKE_AR • …

Table 6-2: cmake options for build environment setup

[3] In the next SDK’s releases, the Eclipse environment might be removed sticking to “Unix Makefile” option only.

Then, the Eclipse project files are deleted because they are not actually used.

And that's it, the build environment is ready to use!

Optionally, the list of targets can be exported into a text file:

```
# cmake --build . --target help > target_list.txt
```

This command line is not part of the script but it can be manually executed by the user to build this targets list file.

```
The targets list file contains all the targets (or executables) that can be built. The Figure
6-2 shows all the targets that can be built:
The following are some of the valid targets for this Makefile:
... all (the default if no target is provided)
... clean
... depend
... edit_cache
... rebuild_cache
... adc_continuous_example
... adc_single_example
... all_di03
... all_di04
... all_di05
... basic_app_tutorial
... boot_selector
... bootloader
... custom_emcore_my_flavor
... firmware_updater
... i2c_example
... i2s_example
... nvm_emb_audio_control
... nvm_emb_audio_control_emcore
... nvm_emb_controller
... nvm_emb_controller_emcore
... nvm_emb_controller_usb
... nvm_emb_datatransfer_server
... nvm_emb_ext_adv
... nvm_emb_fit
... nvm_emb_fit_emcore
... nvm_emb_fit_legacy
... nvm_emb_hid_device
... nvm_emb_hid_usb
... nvm_emb_hrs
... nvm_emb_hrs_emcore
... nvm_emb_hrs_emcore_my_flavor
... nvm_emb_lite_hrs
... nvm_emb_lite_hrs_emcore
... nvm_emb_power_control
... nvm_emb_power_control_emcore
... nvm_emb_rpa
... nvm_emb_tag
... nvm_emb_tag_emcore
... nvm_emsas_example_spi
... nvm_fwu_target
... nvm_fwu_target_emcore
... nvm_fwu_target_v2
... nvm_fwu_target_v2_emcore
... nvm_qpc_example_spi
... nvm_unit_test_spi
... printf_example
... qdec_example
... rtc_example
... spi_master_example
... spi_slave_example
... ti_example
... timer_example_expresso
... timer_example_oneshot
... timer_example_periodic
... watchdog_example
... adc_continuous_example_di03
... adc_continuous_example_di04
... adc_continuous_example_di05
... adc_single_example_di03
... adc_single_example_di04
... adc_single_example_di05
... basic_app_tutorial_di03
... basic_app_tutorial_di04
... basic_app_tutorial_di05
... boot_selector_di03
... boot_selector_di04
... boot_selector_di05
... bootloader_di03
... bootloader_di04
... bootloader_di05
... custom_emcore_my_flavor_di03
... custom_emcore_my_flavor_di04
... custom_emcore_my_flavor_di05
... custom_emcore_my_flavor_no_crc_check_di03
... custom_emcore_my_flavor_no_crc_check_di04
... custom_emcore_my_flavor_no_crc_check_di05
... emcore_information_default
... emcore_information_my_flavor
... emcore_information_standard
... emdt_profile_server
... emdt_service
... firmware_updater_di03
... firmware_updater_di04
... firmware_updater_di05
... fwu_profile_target
... fwu_profile_target_light
... fwu_service
... fwu_service_light
... i2c_example_di03
... i2c_example_di04
... i2c_example_di05
... i2s_example_di03
... i2s_example_di04
... i2s_example_di05
... nvm_emb_audio_control_di03
... nvm_emb_audio_control_di04
... nvm_emb_audio_control_di05
... nvm_emb_audio_control_emcore_di03
... nvm_emb_audio_control_emcore_di04
... nvm_emb_audio_control_emcore_di05
... nvm_emb_controller_di03
... nvm_emb_controller_di04
... nvm_emb_controller_di05
... nvm_emb_controller_emcore_di03
... nvm_emb_controller_emcore_di04
... nvm_emb_controller_emcore_di05
... nvm_emb_controller_usb_di03
... nvm_emb_controller_usb_di04
... nvm_emb_controller_usb_di05
... nvm_emb_datatransfer_server_di03
... nvm_emb_datatransfer_server_di04
... nvm_emb_datatransfer_server_di05
... nvm_emb_ext_adv_di03
... nvm_emb_ext_adv_di04
... nvm_emb_ext_adv_di05
... nvm_emb_fit_di03
... nvm_emb_fit_di04
... nvm_emb_fit_di05
... nvm_emb_fit_emcore_di03
... nvm_emb_fit_emcore_di04
... nvm_emb_fit_emcore_di05
... nvm_emb_fit_legacy_di03
... nvm_emb_fit_legacy_di04
... nvm_emb_fit_legacy_di05
... nvm_emb_hid_device_di03
... nvm_emb_hid_device_di04
... nvm_emb_hid_device_di05
... nvm_emb_hid_usb_di03
... nvm_emb_hid_usb_di04
... nvm_emb_hid_usb_di05
... nvm_emb_hrs_di03
... nvm_emb_hrs_di04
... nvm_emb_hrs_di05
... nvm_emb_hrs_emcore_di03
... nvm_emb_hrs_emcore_di04
... nvm_emb_hrs_emcore_di05
... nvm_emb_hrs_emcore_my_flavor_di03
... nvm_emb_hrs_emcore_my_flavor_di04
... nvm_emb_hrs_emcore_my_flavor_di05
... nvm_emb_lite_hrs_di03
... nvm_emb_lite_hrs_di04
... nvm_emb_lite_hrs_di05
... nvm_emb_lite_hrs_emcore_di03
... nvm_emb_lite_hrs_emcore_di04
... nvm_emb_lite_hrs_emcore_di05
... nvm_emb_power_control_di03
... nvm_emb_power_control_di04
... nvm_emb_power_control_di05
... nvm_emb_power_control_emcore_di03
... nvm_emb_power_control_emcore_di04
... nvm_emb_power_control_emcore_di05
... nvm_emb_rpa_di03
... nvm_emb_rpa_di04
... nvm_emb_rpa_di05
... nvm_emb_tag_di03
... nvm_emb_tag_di04
... nvm_emb_tag_di05
... nvm_emb_tag_emcore_di03
... nvm_emb_tag_emcore_di04
... nvm_emb_tag_emcore_di05
... nvm_emsas_example_spi_di03
... nvm_emsas_example_spi_di04
... nvm_emsas_example_spi_di05
... nvm_fwu_target_di03
... nvm_fwu_target_di04
... nvm_fwu_target_di05
... nvm_fwu_target_emcore_di03
... nvm_fwu_target_emcore_di04
... nvm_fwu_target_emcore_di05
... nvm_fwu_target_v2_di03
... nvm_fwu_target_v2_di04
... nvm_fwu_target_v2_di05
... nvm_fwu_target_v2_emcore_di03
... nvm_fwu_target_v2_emcore_di04
... nvm_fwu_target_v2_emcore_di05
... nvm_qpc_example_spi_di03
... nvm_qpc_example_spi_di04
... nvm_qpc_example_spi_di05
... nvm_unit_test_spi_di03
... nvm_unit_test_spi_di04
... nvm_unit_test_spi_di05
... printf_example_di03
... printf_example_di04
... printf_example_di05
... qdec_example_di03
... qdec_example_di04
... qdec_example_di05
... rtc_example_di03
... rtc_example_di04
... rtc_example_di05
... spi_master_example_di03
... spi_master_example_di04
... spi_master_example_di05
... spi_slave_example_di03
... spi_slave_example_di04
... spi_slave_example_di05
... ti_example_di03
... ti_example_di04
... ti_example_di05
... timer_example_expresso_di03
... timer_example_expresso_di04
... timer_example_expresso_di05
... timer_example_oneshot_di03
... timer_example_oneshot_di04
... timer_example_oneshot_di05
... timer_example_periodic_di03
... timer_example_periodic_di04
... timer_example_periodic_di05
... watchdog_example_di03
... watchdog_example_di04
... watchdog_example_di05
```

Figure 6-2: Example of targets that can be built

As you might have noticed, a set of targets is first listed without any design iteration in its name, i.e. without being suffixed with something like `_dixx`. These are the targets that are built to be executed on the latest version of the hardware which is DI05. This is the default behavior when no design iteration is indicated.

Then the second parts of available targets in the list are suffixed with a design iteration pattern `_diXX` (e.g. `_di03,` `_di04` or `_di05)`. This pattern is used to indicate for which DI the software shall be built. This useful in case a specific version other than the last one has to be built.

6.4 BUILD EXAMPLES

6.4.1 The printf example

Now that the build environment is set up and that we have the complete list of targets, some examples can be built.

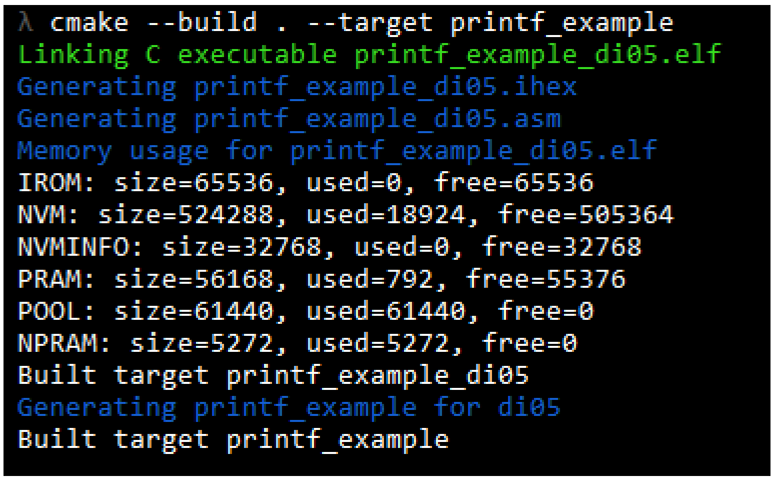
The simplest example is the `printf` which sends messages to the host computer through a serial line.

To build this example for the default design iteration, run the following command from the `build` folder:

```
# cmake –build . –target printf_example
```

The expected output is the `printf_example.ihex` file located in the `<sdk>/<build_folder>/projects/`
`print_example/` folder.

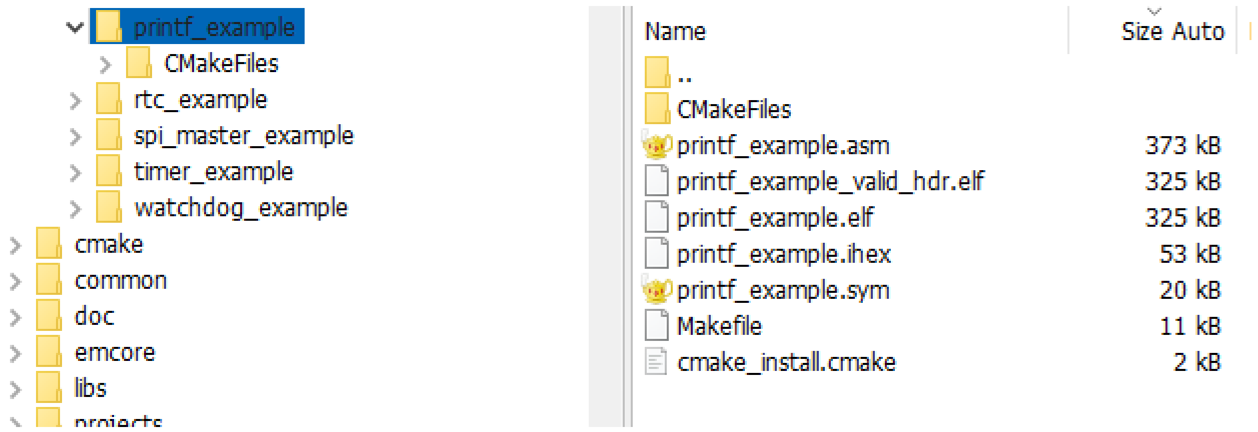
The Figure 6-3 shows the build process output where it reached 100% indicating that the build process successfully completed.



```
λ cmake --build . --target printf_example
Linking C executable printf_example_di05.elf
Generating printf_example_di05.ihex
Generating printf_example_di05.asm
Memory usage for printf_example_di05.elf
IROM: size=65536, used=0, free=65536
NVM: size=524288, used=18924, free=505364
NVMINFO: size=32768, used=0, free=32768
PRAM: size=56168, used=792, free=55376
POOL: size=61440, used=61440, free=0
NPRAM: size=5272, used=5272, free=0
Built target printf_example_di05
Generating printf_example for di05
Built target printf_example
```

Figure 6-3: Printf example build output

The Figure 6-4 shows the output files for the `printf_example` which are located under `<SDK_INSTALLATION_PATH>/build/projects/printf_example` folder.



Name	Size Auto
..	
CMakeFiles	
printf_example.asm	373 kB
printf_example_valid_hdr.elf	325 kB
printf_example.elf	325 kB
printf_example.ihex	53 kB
printf_example.sym	20 kB
Makefile	11 kB
cmake_install.cmake	2 kB

Figure 6-4: Output files for the printf example

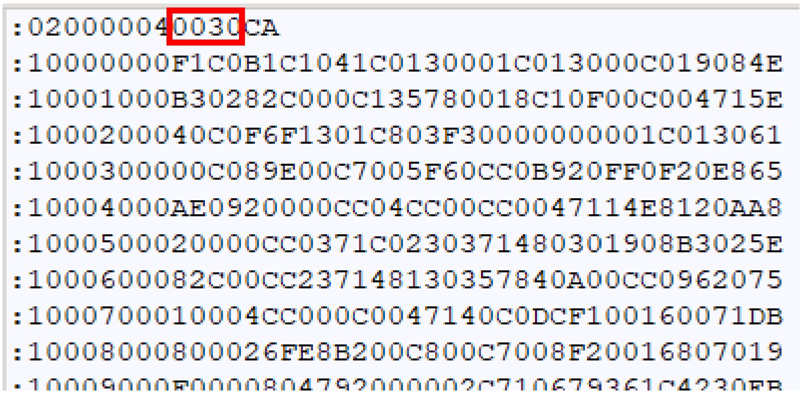
The Table 6-3 provides an explanation and description of the files generated by the build process on behalf of the printf example.

File	Description
printf_example.asm	the source assembly file
printf_example.elf	the elf executable file as a direct output from the linker without the header
printf_example.ihex	the Intel hex file used to program the device containing the header
printf_example.sym	the symbol files
printf_example _valid_hdr.elf	the elf executable with the header prepended and used to identify this image in NVM

Table 6-3: Build process output files description for printf_example

The most important file here is the `.ihex` file that shall be used to program the device.

By diving into this file, we can see that this executable is mapped into the NVM at address 0x300000 (beginning of the flash memory). This is shown on Figure 6-5.

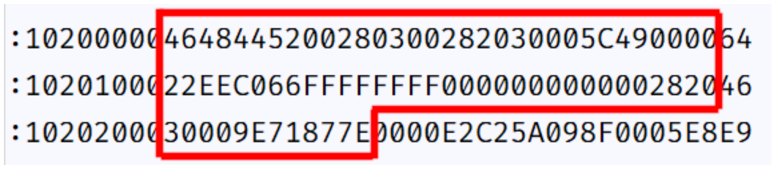


```
:020000040030CA
:10000000F1C0B1C1041C0130001C013000C019084E
:10001000B30282C000C135780018C10F00C004715E
:1000200040C0F6F1301C803F30000000001C013061
:1000300000C089E00C7005F60CC0B920FF0F20E865
:10004000AE0920000CC04CC00CC0047114E8120AA8
:1000500020000CC0371C02303714803019O8B3025E
:1000600082C00CC23714813035784OA00CC0962075
:1000700010004CC000C0047140C0DCF100160071DB
:1000800080026FE8B200C800C7008F20016807019
:10009000F0000804792000002C710679361C4230EB
```

Figure 6-5: Beginning of printf example ihex file

The first line indicates the two most significant bytes of the address which is `0x0030`. Then any further data will be stored starting from `0x0030YYYY` where the YYYY pattern is related to the two least significant bytes found on each further line of the file.

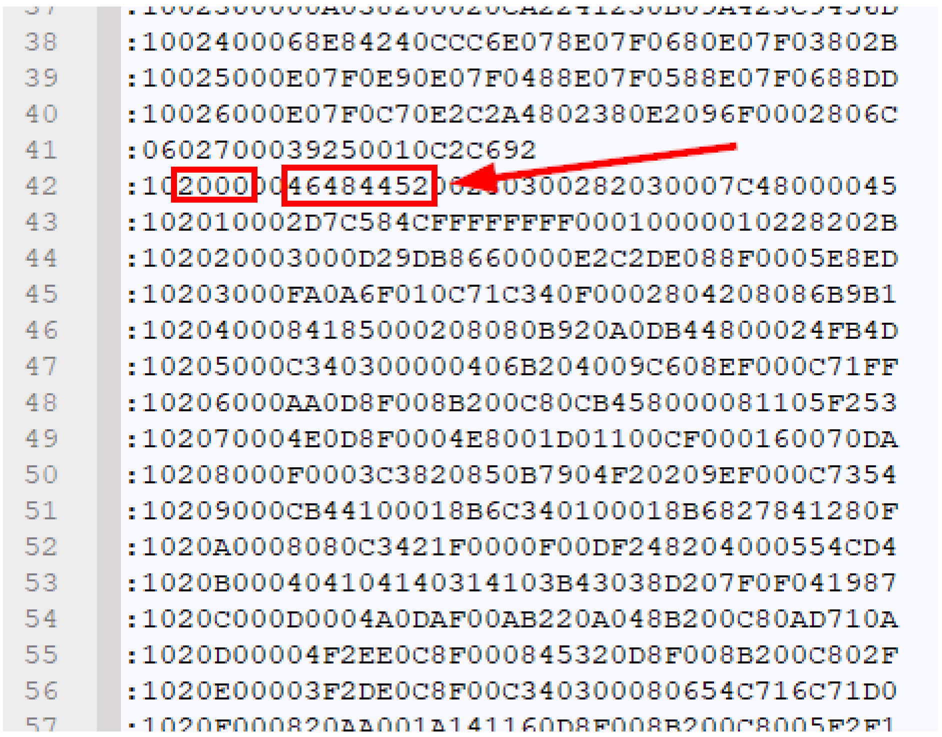
The example application starts at address `0x302000` and the first word that is found at this location is the so-called magic number `0x52444846` (little endianness). These four bytes are the hex representation of the string “FHDR” or “file header”.



```
:10200000464844520028030028203000 5C49000064
:1020100022EEC066FFFFFFFF0000000000002820 46
:1020200030009E71877E0000E2C25A098F0005E8E9
```

Figure 6-6: Printf example header

The Figure 6-7 shows an example on where this magic number can be found.



```
:1002300000A038200020CA2241230B05A423C9456D
:1002400068E84240CCC6E078E07F0680E07F03802B
:10025000E07F0E90E07F0488E07F0588E07F0688DD
:10026000E07F0C70E2C2A4802380E2096F0002806C
:0602700039250010C2C692
:1020000046484452062003002820300 07C48000045
:102010002D7C584CFFFFFFFF00010000010228202B
:102020003000D29DB8660000E2C2DE088F0005E8ED
:10203000FA0A6F010C71C340F00028042080086B9B1
:1020400084185000208080B920A0DB44800024FB4D
:10205000C340300000406B204009C608EF000C71FF
:10206000AA0D8F008B200C80CB458000081105F253
:102070004E0D8F0004E8001D01100CF000160070DA
:10208000F0003C3820850B7904F20209EF000C7354
:10209000CB44100018B6C340100018B6827841280F
:1020A0008080C3421F0000F00DF248204000554CD4
:1020B00040410414031410 3B43038D207F0F041987
:1020C000D0004A0DAF00AB220A048B200C80AD710A
:1020D0004F2EE0C8F000845320D8F008B200C802F
:1020E0003F2DE0C8F00C340300080654C716C71D0
:1020F000820AA001A141160D8F008B200C8005F2F1
```

Figure 6-7: Header magic number identification

The Figure 6-7 is a helper to identify each header field from the Intel Hex file and can be used to quickly identify if a FW image header has been generated.

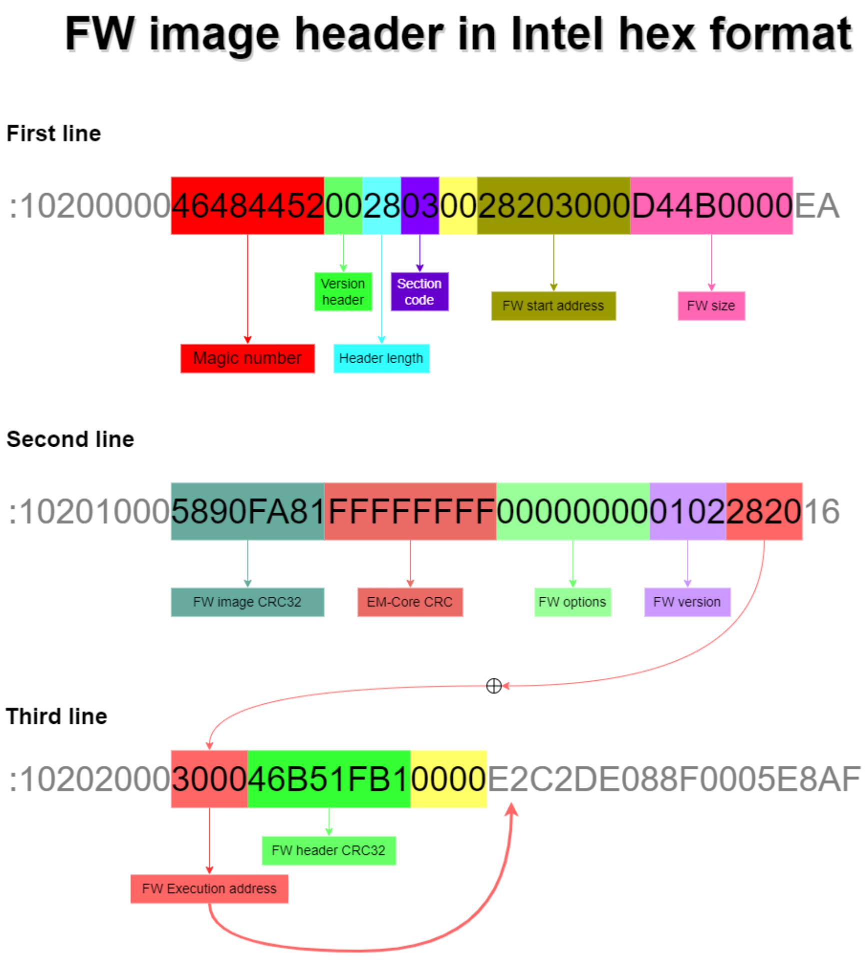


Figure 6-8: How to decode a FW image header from Intel Hex file

Programming this application example is done using `blengine_cli.py` provided in the tools folder of the SDK.

```
# cd <SDK_INSTALLATION_PATH>\tools\blengine
# pythonblengine_cli.py -port COM5 run emsystem_prog
                              ..\..\build\projects\printf_example\printf_example.ihex
```

Here, it is assumed that the DVK is connected to the host computer through the USB link and that a virtual COM port named COM5 has been created. This configuration might be different on other installations.

The blengine_cli.py invokes the run command to execute the emsystem_prog operation. It is then followed by the relative path to the .ihex file to be programmed. Note that an absolute path can be also specified.

More technical details on blengine are provided in [6].

Also check the SDK documentation to get information on how to setup the `printf_example` with the DVK.

6.4.2 EM-Bleu controller example

The second example exposed here is a more complete application that integrates the following modules:

- Bluetooth LE controller

- EM-core
- Transport over SPI

The build process is the same as above but the target is different:

```
# cmake -build . -target nvm_emb_controller
```

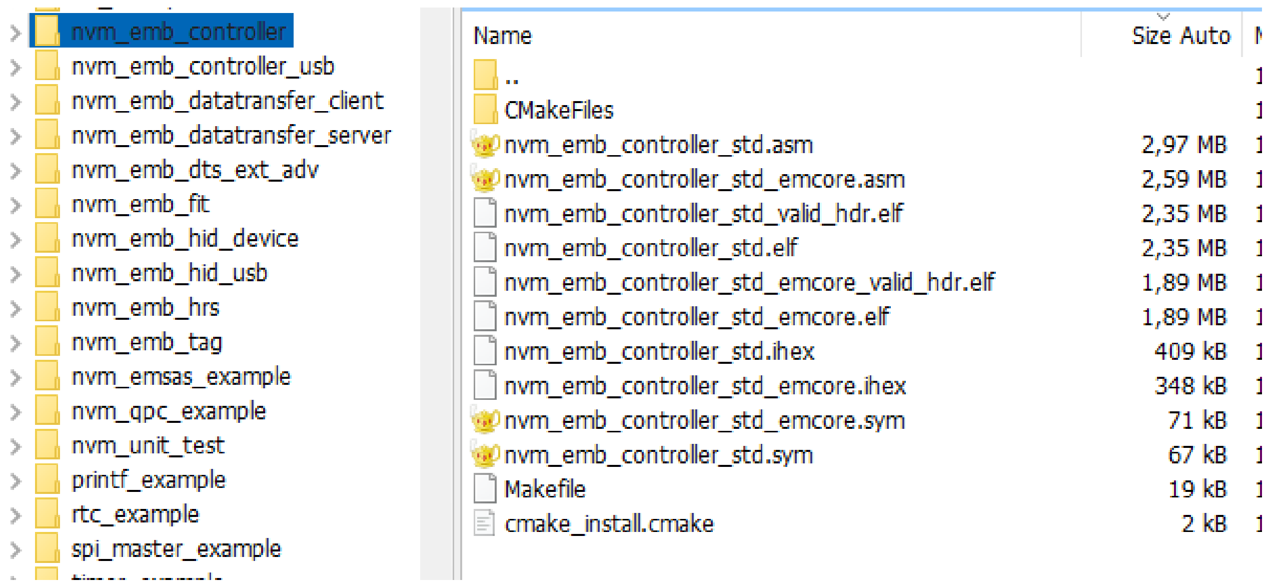
This targets produces an executable that will integrate the following functionalities:

- em|bleu Bluetooth LE controller 5.3
- Transport layer over SPI (HCI commands are received through the SPI bus)
- EM-Core (as a standalone application)

It will be also executed from the NVM since the link process has mapped the binary in this memory.

The built files are located in `<SDK_INSTALLATION_PATH>/build/projects/nvm_emb_controller` as shown in Figure 6-9.

With this application example, the EM9305 device is used as a Bluetooth LE controller compliant with the Bluetooth LE core specification version 5.3 using the em|bleu Bluetooth LE stack. As shown in Figure 6-9, binaries have been built for the latest version of the device.



nvm_emb_controller
nvm_emb_controller_usb
nvm_emb_datatransfer_client
nvm_emb_datatransfer_server
nvm_emb_dts_ext_adv
nvm_emb_fit
nvm_emb_hid_device
nvm_emb_hid_usb
nvm_emb_hrs
nvm_emb_tag
nvm_emsas_example
nvm_qpc_example
nvm_unit_test
printf_example
rtc_example
spi_master_example

Name	Size Auto
..	
CMakeFiles	
nvm_emb_controller_std.asm	2,97 MB
nvm_emb_controller_std_emcore.asm	2,59 MB
nvm_emb_controller_std_valid_hdr.elf	2,35 MB
nvm_emb_controller_std.elf	2,35 MB
nvm_emb_controller_std_emcore_valid_hdr.elf	1,89 MB
nvm_emb_controller_std_emcore.elf	1,89 MB
nvm_emb_controller_std.ihex	409 kB
nvm_emb_controller_std_emcore.ihex	348 kB
nvm_emb_controller_std_emcore.sym	71 kB
nvm_emb_controller_std.sym	67 kB
Makefile	19 kB
cmake_install.cmake	2 kB

Figure 6-9: nvm_emb_controller application output files

So, in this application example, the 9305 device receives its HCI commands from the SPI bus[4] and sends Bluetooth LE frames to the host as depicted on Figure 6-10.

[4] An UART can also be used as a transport layer instead of the SPI.

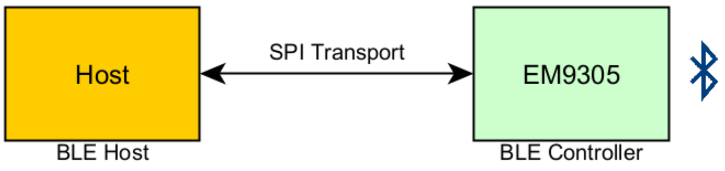


Figure 6-10: Using EM9305 as a Bluetooth LE controller

With this application, the EM9305 is a passive device receiving its instructions from the host which can be another SOC (9305, Raspberry Pi …).

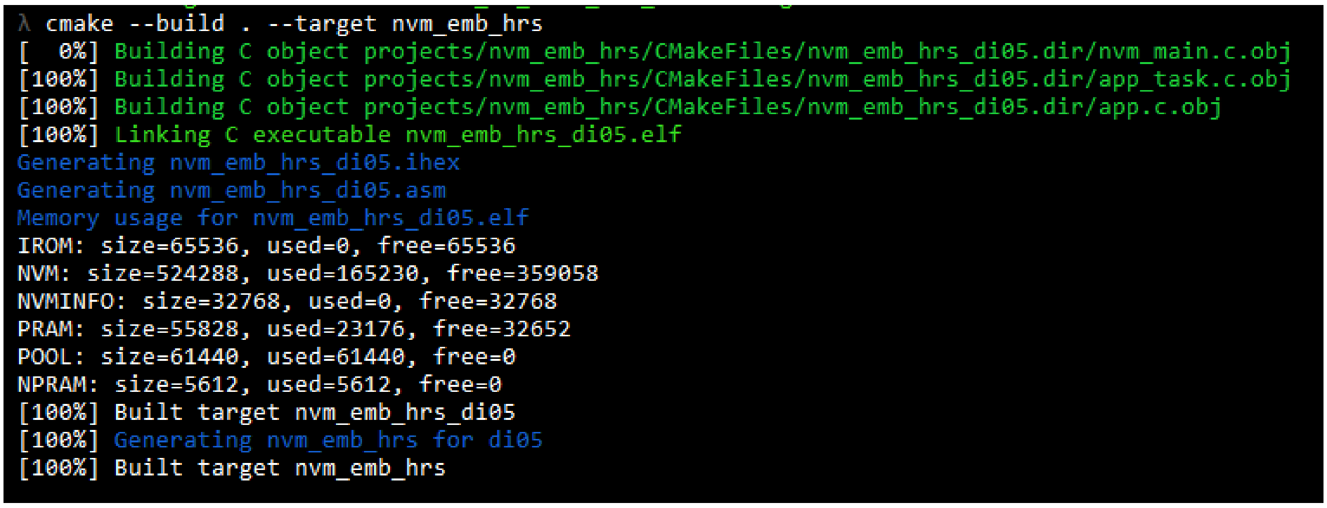
6.4.3 Heart rate sensor

The final application example is the heart rate sensor acting as the peripheral while the connected smartphone is acting as the central. When the central is connected to the peripheral and the notifications are enabled on the 'heart rate measurement characteristic', the central starts receiving the expended energy on a periodic basis.

To build the target, execute the following command, still in the `build` folder:

```
# cmake –build . –target nvm_emb_hrs
```

The Figure 6-11 shows the build process output.



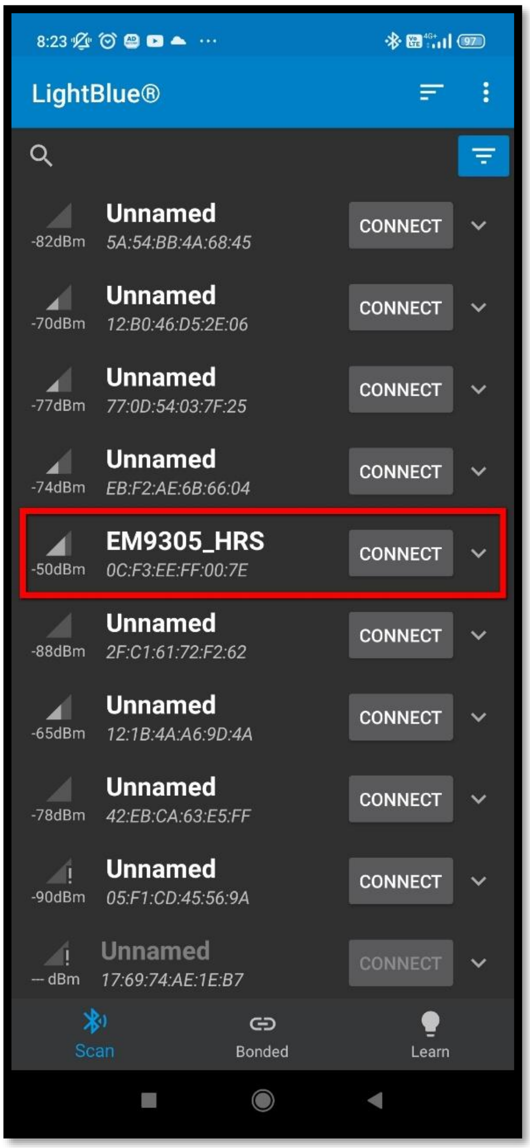
```
λ cmake --build . --target nvm_emb_hrs
[  0%] Building C object projects/nvm_emb_hrs/CMakeFiles/nvm_emb_hrs_di05.dir/nvm_main.c.obj
[100%] Building C object projects/nvm_emb_hrs/CMakeFiles/nvm_emb_hrs_di05.dir/app_task.c.obj
[100%] Building C object projects/nvm_emb_hrs/CMakeFiles/nvm_emb_hrs_di05.dir/app.c.obj
[100%] Linking C executable nvm_emb_hrs_di05.elf
Generating nvm_emb_hrs_di05.ihex
Generating nvm_emb_hrs_di05.asm
Memory usage for nvm_emb_hrs_di05.elf
IROM: size=65536, used=0, free=65536
NVM: size=524288, used=165230, free=359058
NVMINFO: size=32768, used=0, free=32768
PRAM: size=55828, used=23176, free=32652
POOL: size=61440, used=61440, free=0
NPRAM: size=5612, used=5612, free=0
[100%] Built target nvm_emb_hrs_di05
[100%] Generating nvm_emb_hrs for di05
[100%] Built target nvm_emb_hrs
```

Figure 6-11: Heart rate sensor build process output

Then programming this example is done through the following command line:

```
# cd <SDK_INSTALLATION_PATH>/tools/blengine
# cd <SDK>/tools/blengine
# python blengine_cli.py –port COMXX run emsystem_prog
                    <SDK_INSTALLATION_PATH>/build/projects/nvm_emb_hrs/nvm_emb_hrs.ihex
```

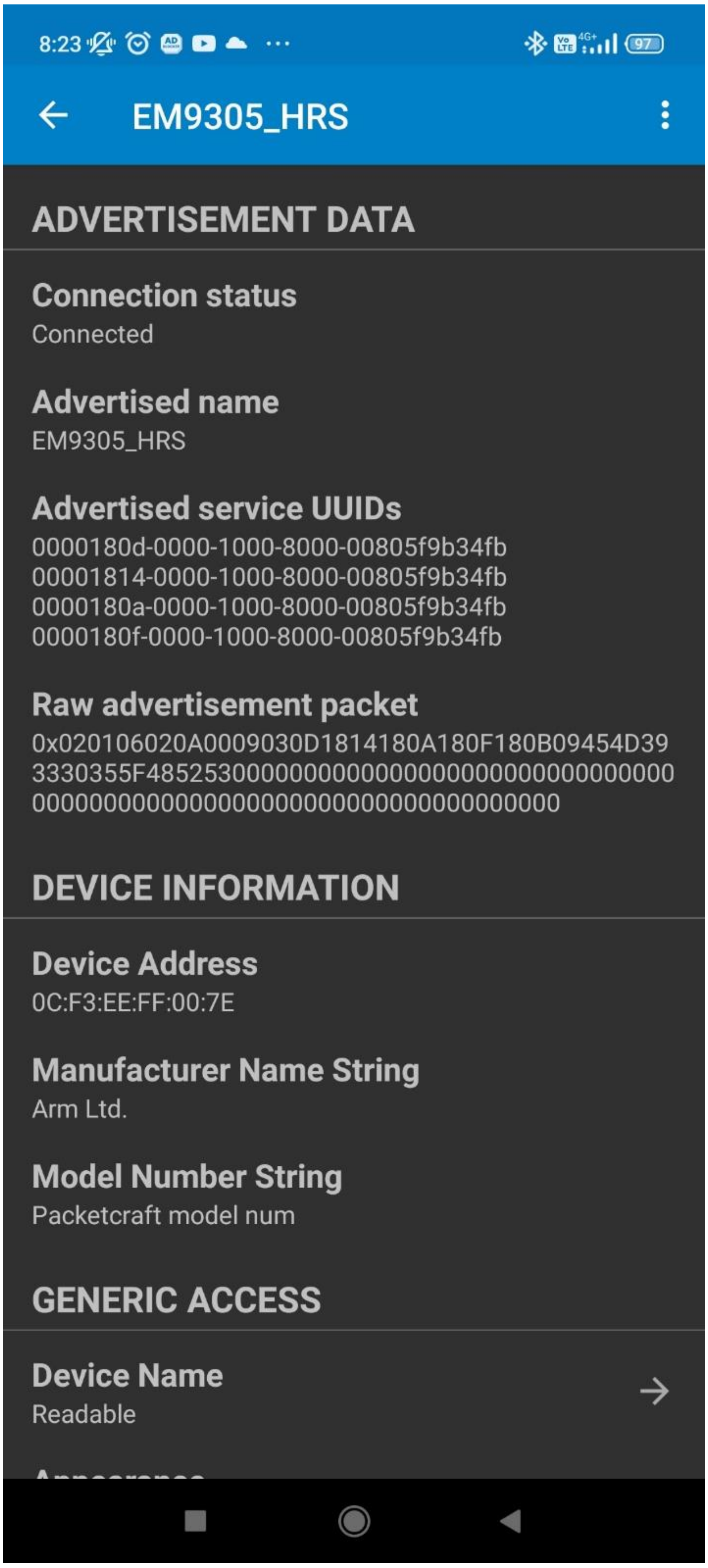
Once the application is programmed in the device, it starts advertising. This can be seen on a smartphone running the Light Blue application showing a beacon which name is "`EM9305_HRS`" as exposed in Figure 6-12.


LightBlue®

-82dBm	Unnamed	5A:54:BB:4A:68:45	CONNECT
-70dBm	Unnamed	12:B0:46:D5:2E:06	CONNECT
-77dBm	Unnamed	77:0D:54:03:7F:25	CONNECT
-74dBm	Unnamed	EB:F2:AE:6B:66:04	CONNECT
-50dBm	EM9305_HRS	0C:F3:EE:FF:00:7E	CONNECT
-88dBm	Unnamed	2F:C1:61:72:F2:62	CONNECT
-65dBm	Unnamed	12:1B:4A:A6:9D:4A	CONNECT
-78dBm	Unnamed	42:EB:CA:63:E5:FF	CONNECT
-90dBm	Unnamed	05:F1:CD:45:56:9A	CONNECT
— dBm	Unnamed	17:69:74:AE:1E:B7	CONNECT

Figure 6-12: Using LightBlue application

Click on the Connect button to pair the smartphone with the EM9305. Once done, device information is displayed as depicted in Figure 6-13 along with the connection status (in this example “Connected”).


8:23

EM9305_HRS

ADVERTISEMENT DATA

Connection status
Connected

Advertised name
EM9305_HRS

Advertised service UUIDs
0000180d-0000-1000-8000-00805f9b34fb
00001814-0000-1000-8000-00805f9b34fb
0000180a-0000-1000-8000-00805f9b34fb
0000180f-0000-1000-8000-00805f9b34fb

Raw advertisement packet
0x020106020A0009030D1814180A180F180B09454D39
3330355F48525300000000000000000000000000000000
0000000000000000000000000000000000000

DEVICE INFORMATION

Device Address
0C:F3:EE:FF:00:7E

Manufacturer Name String
Arm Ltd.

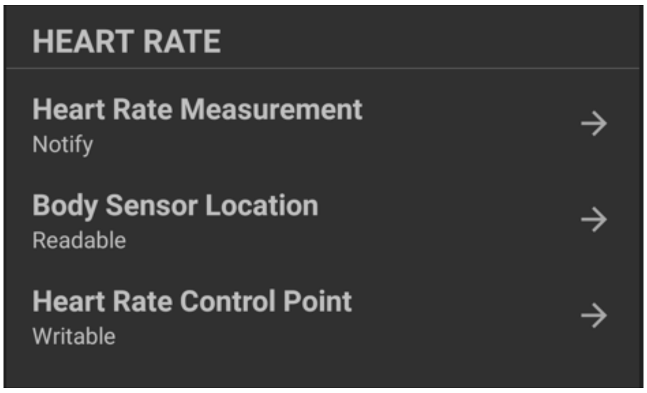
Model Number String
Packetcraft model num

GENERIC ACCESS

Device Name
Readable

Figure 6-13: Device information after connection

By scrolling down this screen to the “Heart Rate” part, it is possible to access to the heart rate measurement subscription page by clicking the “Notify” button as shown on Figure 6-14.


HEART RATE

Heart Rate Measurement
Notify

Body Sensor Location
Readable

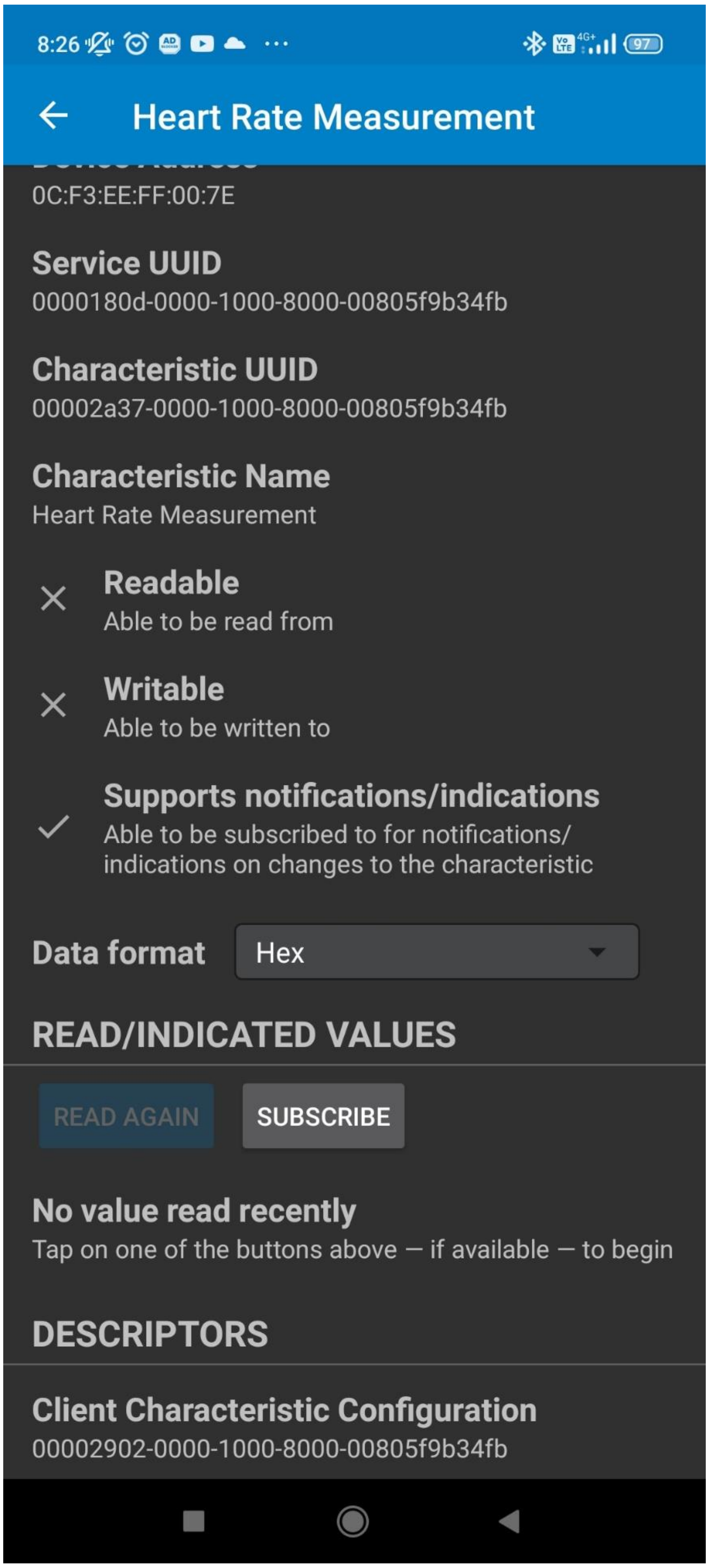
Heart Rate Control Point
Writable

Figure 6-14: Heart rate notify

The next page shows a subscription button (see Figure 6-15). Clicking on this button will activate the reception of a simulated expended energy by the heart rate device. This value is incremented once per second and the smartphone application displays it each time it is updated.

The Figure 6-16 shows a series of bytes in which the second byte from right is incremented each second. This value is actually sent by the EM9305, is received by the smartphone and displayed.

The “Unsubscribe” button is displayed as well. When unsubscribing, the value refresh is stopped.


8:26

Heart Rate Measurement

0C:F3:EE:FF:00:7E

Service UUID
0000180d-0000-1000-8000-00805f9b34fb

Characteristic UUID
00002a37-0000-1000-8000-00805f9b34fb

Characteristic Name
Heart Rate Measurement

✗ **Readable**
Able to be read from

✗ **Writable**
Able to be written to

✓ **Supports notifications/indications**
Able to be subscribed to for notifications/indications on changes to the characteristic

Data format Hex

READ/INDICATED VALUES

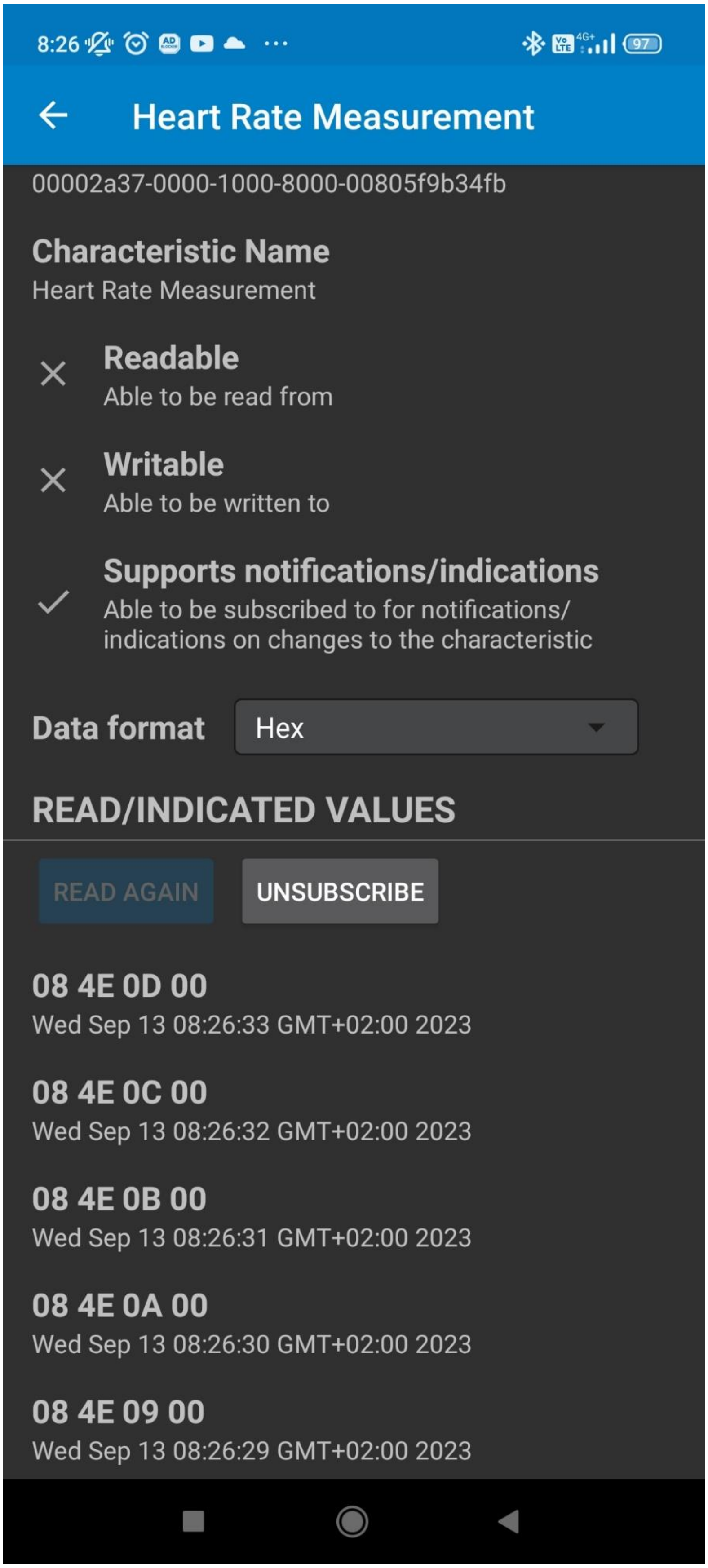
READ AGAIN SUBSCRIBE

No value read recently
Tap on one of the buttons above — if available — to begin

DESCRIPTORS

Client Characteristic Configuration
00002902-0000-1000-8000-00805f9b34fb

Figure 6-15: Heart rate service subscription



00002a37-0000-1000-8000-00805f9b34fb

Characteristic Name
Heart Rate Measurement

× Readable
Able to be read from

× Writable
Able to be written to

✓ Supports notifications/indications
Able to be subscribed to for notifications/ indications on changes to the characteristic

Data format: Hex

READ/INDICATED VALUES

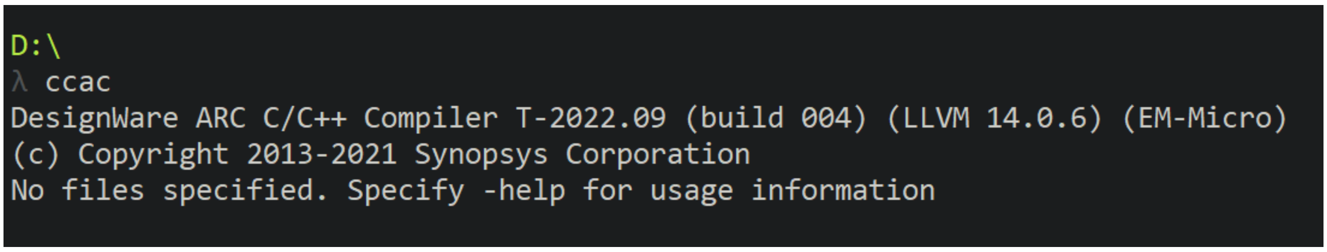
```
08 4E 0D 00
Wed Sep 13 08:26:33 GMT+02:00 2023

08 4E 0C 00
Wed Sep 13 08:26:32 GMT+02:00 2023

08 4E 0B 00
Wed Sep 13 08:26:31 GMT+02:00 2023

08 4E 0A 00
Wed Sep 13 08:26:30 GMT+02:00 2023

08 4E 09 00
Wed Sep 13 08:26:29 GMT+02:00 2023
```

Figure 6-16: Heart rate simulated expended energy

6.5 METAWARE TOOLCHAIN SELECTION

By default, the build process will search for a Metaware ARC toolchain installed in the system path. Consequently, the toolchain has to be installed prior to the SDK's installation. Thus, all the tools provided by this toolchain shall be accessible from any location on your computer.

To check for the installed toolchain version, open a terminal window and type in the `ccac` command. If everything is ok, the version of the compiler should be displayed.



```
D:\
λ ccac
DesignWare ARC C/C++ Compiler T-2022.09 (build 004) (LLVM 14.0.6) (EM-Micro)
(c) Copyright 2013-2021 Synopsys Corporation
No files specified. Specify -help for usage information
```

In this example, the Metaware 2022.09 is installed system wide.

It is possible to enforce a specific Metaware toolchain version by setting the path to this version using the `MW_INSTALL_PATH` variable like in the following example:

```
set MW_INSTALL_PATH=C:\Toolchains\MetaWare_2019\MetaWare
```

Once done, re-run the `init.bat` script located at the SDK's root and start building an example with the `-v` option to increase verbosity.

```
Cd <sdk>
./init.bat
cd build
cmake -build . -target printf_example -v
```

You should then see the MetaWare_2019 toolchain being used to build-up the application example.

6.6 CONCLUSION

There are more example applications and it is recommended to study them in detail in order to understand how they are built. They can also serve as a baseline to create a more complex end user application that will make use of all the EM9305 SOC power.

7. CREATING A NEW PROJECT

7.1 OVERVIEW

The goal of the SDK is to provide the end user with software bricks, tools and build ecosystem so that they can create and build his own final application that will run on an EM9305 device.

Depending on their needs, the end user will use some of the provided software blocks at will. However, due to the way the device works, in order to be compliant with the build process, any application shall observe a specific architecture. Consequently, starting a new project has to be done by following some rules which are explained in this chapter.

7.2 APPLICATION TYPE

7.2.1 Overview

Before diving into the design of an application, the type of this application shall be defined first. The two available possibilities are the following:

- standalone application
- EM-Core based application.

7.2.2 Standalone application

This type of application does not depend on external libraries or other firmware to properly run. It embeds every driver and library that it needs for its own usage.

The only exception is that it might use functions that are provided in the ROM like the NVM driver or any security related function.

Otherwise, it can completely get rid of any EM-Core version already stored in the NVM. Thus, it can use all of the NVM space except the first page which shall contain the NVM-Bootloader.

The Figure 7-1 shows the NVM content in case of a standalone end user application.

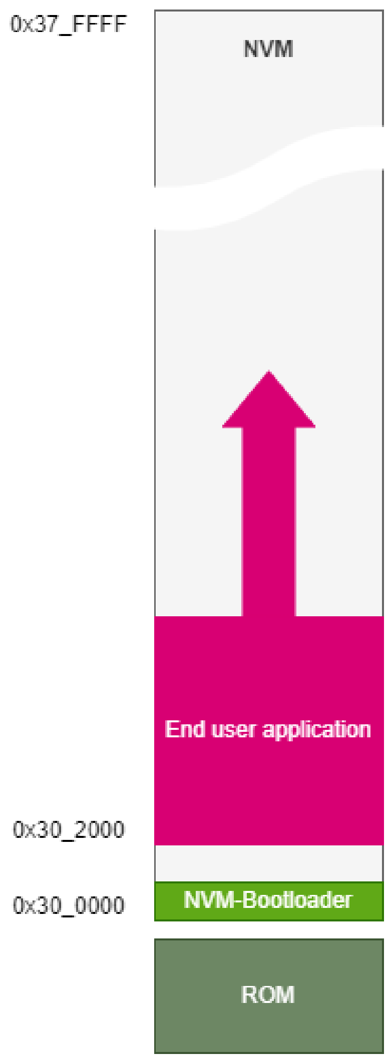


Figure 7-1: NVM content for standalone application

The application starts at the first page boundary after page 1 and can grow up to reach the end of the NVM. With exception to functions provided in the ROM, the end user shall embed all it needs for its proper execution.

7.2.3 EM-Core based application

An EM-Core based application is an end user application that will rely on the drivers and libraries made available through an already programmed EM-Core firmware image.

The Figure 7-2 shows how things are organized in the NVM in this case.

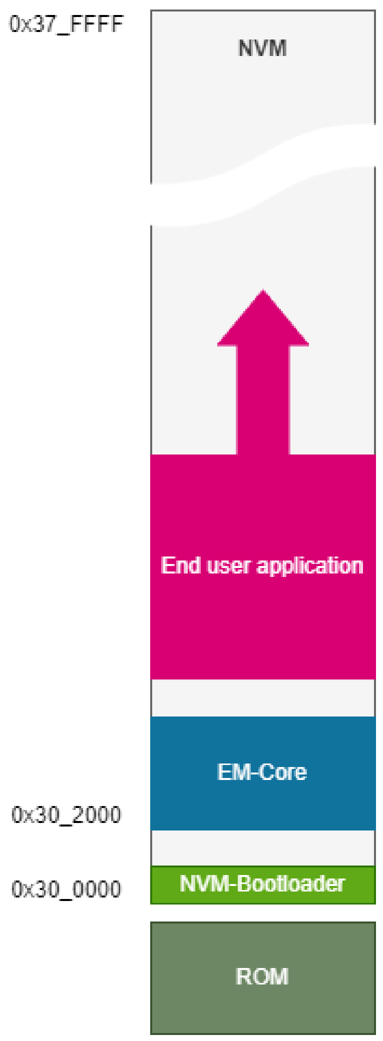


Figure 7-2: NVM organization for EM-Core based application

The pros for this type of application is that it takes less space in NVM and is smaller than the standalone type application which can be more efficient when upgrading through Bluetooth LE for example. But the con is that the exact same EM-Core version that the end user application has been linked with shall be already stored in the NVM. Otherwise, the application will not work.

Depending on the type of application to build and since the end user application will not have the same starting address, two different linker scripts are provided and used. For a standalone application, the linker script `linker.cmd` located under `<sdk>\common\9305\NVM` enforces the start address at `0x30_2000` which is the first page after the page containing the NVM-Bootloader.

The second `linker.cmd` script located under `<sdk>\common\9305\APP` is used to map the end user application after the EM-Core location in NVM, when the end user application relies on EM-Core.

Fortunately, it is not expected for an end user to directly deal with these scripts. The right script selection is done through the use of dedicated Cmake macros.

Later on, this guide will show how to select the type of application to be built. However, it is quite possible to build both types of an application at the same time in one build shot. It is just a matter of editing the application's `CmakeLists.txt` file.

7.2.4 Conclusion

The Table 7-1 lists the pros and cons for both types of application.

Application type	pros	cons
standalone	No need to rely on other FW (except the ROM) Can use all the NVM except first page Embeds exactly what it needs	Bigger
EM-Core based	Smaller Best suited and more efficient for FOTA	Same version of EM-Core shall be already programmed otherwise the application cannot work. Some drivers part of EM-Core might not be useful for the application which wastes some space. Some NVM space shall be reserved for storing EM-Core.

Table 7-1: Pros and cons for standalone and EM-Core based applications

7.3 BASIC STRUCTURE OF AN APPLICATION

As explained above, all applications shall be compliant with a specific architecture. This is imposed by the fact that the end user application is executed after the ROM boot sequence has been done. Thus, the ROM code expects to find specific and dedicated functions somewhere in the NVM in order to properly configure the device and start the application.

The end user application can be a simple bare metal application that executes its code in sequence. However, some applications like Bluetooth LE applications are usually event driven. This means that they can rely on the usage of a real time scheduler like QP/C which will provide a real time framework in which the user can handle all the incoming events.

For example, after the device has initialized the Bluetooth LE stack and a connection request is received, the stack sends a “connected” event which is processed by the application while in idle state. Later, when a “disconnected” event is received, it is processed in the same way.

Having this event driven management eases application development. When not executing an “active” task and if switching to sleep mode has not been forbidden, QP/C puts the application in idle state where the device can go to sleep mode to save energy.

So, for this example, let’s consider an application that relies on the QP/C real time scheduler framework.

An end user application is designed to be in the NVM and to be executed from this location. It shall define the following two functions:

- `NVM_ConfigModules()`
- `NVM_ApplicationEntry()`

The `NVM_ConfigModules()` function is where the application defines how the device will be set up and configured. This is for example the JTAG configuration and the initializion of QP/C as well.

Moreover, this function is also the right place to register the modules that will be used by the application.

It can also be decided what to do depending on whether the device is resuming from a sleep period, or after a cold reset or a warm reset. To achieve this, the SDK provides an API that provides such information (refer to §9.5).

The Figure 7-4 shows a code snippet of a typical `NVM_ConfigModules()` function where some modules are registered. In this example, the UART is resumed after a Power on Reset to be used for UART communication by registering the UART module.

Then the system executes the first stage start-up. It then calls the user defined `NVM_ConfigModules()` function.

Once done, it enters into the second stage start-up in which the configuration previously set up by the `NVM_ConfigModules()` function is actually applied.

Then, the user defined `NVM_ApplicationEntry()` function is called.

This behaviour is described in Figure 7-3 which shows a high level sequence diagram.

The `gGPIO_Config` is a global variable that can be read and written. It is used to configure the right GPIO scheme that will be applied after exiting from the `NVM_ConfigModules()` function.

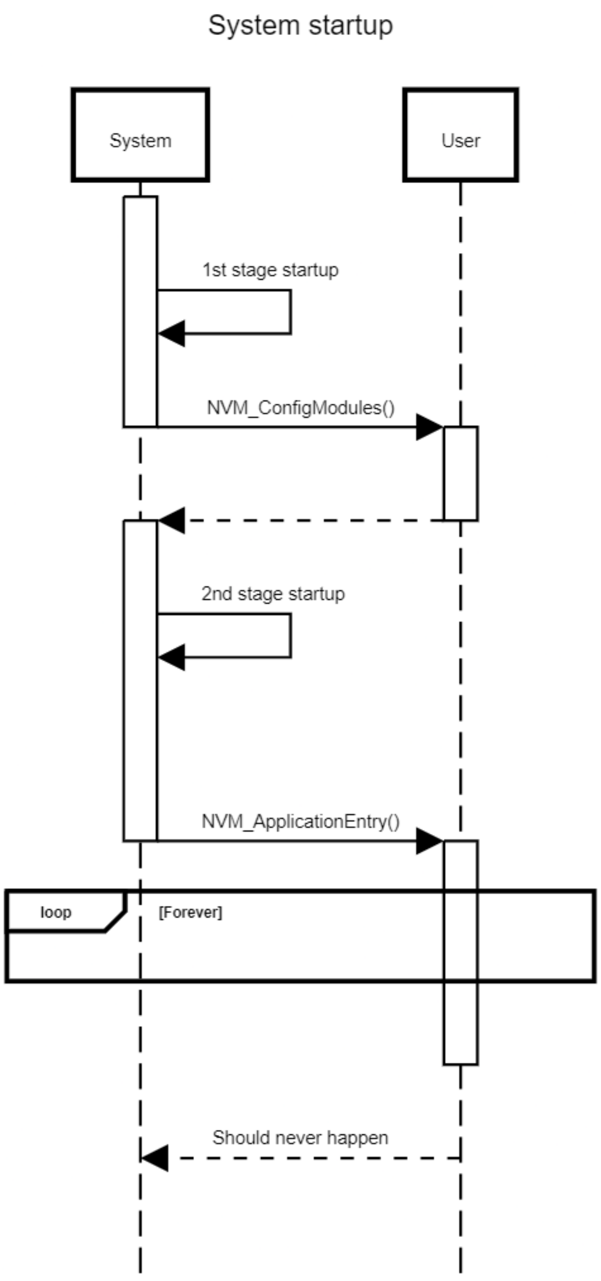


Figure 7-3: The two stage start-up

In the Figure 7-3, the user code is shown on the right side through following functions:

- `NVM_ConfigModules()`

- `NVM_ApplicationEntry()`

Both functions are called once but unlike the `NVM_ConfigModules()` function, the `NVM_ApplicationEntry()` function shall never return. It is quite usual to rely on QP/C scheduler (see §16) with an idle task plus one dedicated task which activation is scheduled based on external events. When not active, the idle task is executed in which the device can be switched to sleep mode.

```
Void NVM_ConfigModules(void)
{
   // Register the timer driver.
   Timer_RegisterModule();
   // Register the UART driver.
   UART_RegisterModule();
   // Register the SPI master.
   SPIM_RegisterModule();

   // Enable the unitimer module
   gTimer_Config.enabled = true;
   // Enable the UART module.
   gUART_Config.enabled = true;
   // Enable the SPIM module.
   gSPIM_Config.enabled = true;

   // If we have started the device not resuming from sleep,
   // then the GPIO configuration shall be done. Otherwise,
   // this configuration is stored in a persistent memory so
   // there is no need to do it again over sleep-wakeup cycles.
   If(!PML_DidBootFromSleep())
   {
      // Do relevant initialization if we do not resume from a sleep period (POR or SW reset)
   }
}
```

Figure 7-4: Example of NVM_ConfigModules() function

The `NVM_ApplicationEntry()` function is the actual entry point of the application. It must be compliant with the following prototype:

```
void NVM_ApplicationEntry(void);
```

This is where the end user writes their own application, and this function shall never return. This is why, it is usual and recommended to add an infinite loop at the end of the `NVM_ApplicationEntry()` function like shown in the code snippet below:

```
void NVM_ApplicationEntry(void)
{
      // some code here
      …
      // In normal operation, the line below shall never be reached!
```

```
    While(true) HaltCPU();
}
```

A more comprehensive and complete step by step tutorial showcasing how to write a QP/C based application is exposed in §16.4.

7.4 WEAK FUNCTIONS

The two functions `NVM_ConfigModules()` and `NVM_ApplicationEntry()` are defined as weak functions in the SDK. Even if they must be provided by the end user application, they have a default body so an end user application can be built even if the user defined versions of these functions are not provided.

The way they are declared is the following:

```
__attribute__((weak)) void NVM_ConfigModules(void);
__attribute__((weak)) void NVM_ApplicationEntry(void);
```

The default `NVM_ConfigModules()` function is empty. It does not contain any code. Thus, the `NVM_ApplicationEntry()` function simply resets the CPU.

Consequently, an application built without the definition of user defined functions will continuously reset the CPU.

A weak function can be easily overridden and overwritten by a user-defined version simply by defining it as a regular function. No special keyword is required here.

Consequently, the following function definition written somewhere in the end user application will override the default empty one:

```
void NVM_ConfigModules(void)
{
    // Add user defined configuration and initialization.
}
```

The linker will use this function implementation instead of using the one to build the final application. It will resolve the function reference in favor of the user defined which is called a “strong” function prototype definition which takes precedence over the weak version of the function. The result is that it will override the one that is declared as “weak” and the user defined one will then be used.

The Table 7-2 lists the weak function that can be overridden by the end user.

Function	Module/Description
NVM_ConfigModules()	NVM Entry. First stage application initialization.
NVM_ApplicationEntry()	NVM Entry. Second stage application initialization.
QF_onStartupExt()	Can be overridden by the end user application to execute custom startup code after QP/C has completed its own startup.
QF_onResumeExt()	Can be overridden by the end user application to execute custom code when QP/C resumes.
QK_onIdleExt()	Can be overridden by the end user application to execute code when the system switches to idle state, and before going to sleep.
Q_onAssertExt()	Used to capture assertion test failures inside the end user application.

Table 7-2: List of weak functions that can be overridden by user defined functions

The default behavior of the functions `QF_onStartupExt()`, the `QF_onResumeExt()` and the `QK_onIdleExt()` functions is to do nothing. They are just a placeholder with an empty core.

Here is how the function `QF_onStartupExt()` function is defined:

```
__attribute__((weak)) void QF_onStartupExt(void)
{
}
```

It can easily be redefined and overridden with a function of the exact same prototype containing user defined code.

7.5 FOLDER STRUCTURE OF AN APPLICATION

A typical folder structure of an application is depicted in Figure 7-5. The example here showcases the heart rate sensor sample application that emulates sending a virtual heart rate sensor data on a periodic basis after a BLE connection has been completed.

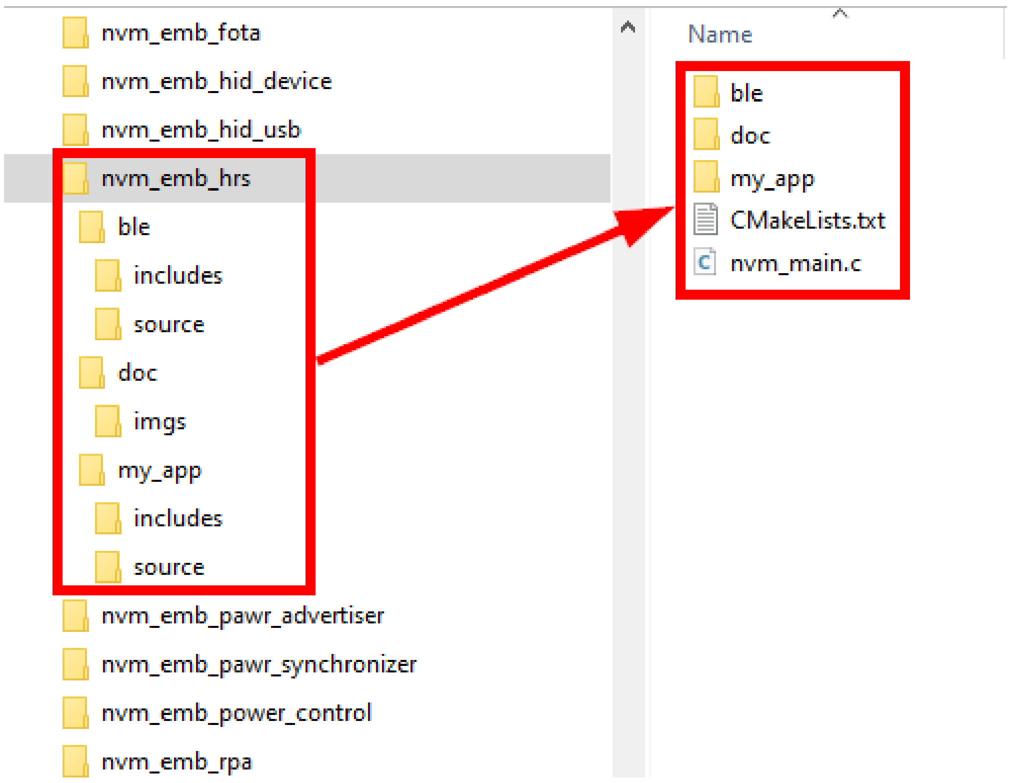


Figure 7-5: Typical folder structure of an application

Within this structure, each file's role is described in Table 7-3, so checkout this table for a detailed description.

File	description
`nvm_main.c`	The upmost top level file containing the `NVM_ConfigModule()` and `NVM_ApplicationEntry()` functions. This file contains the entry point code of the whole application and is located at the root level of the

	application's folder. Its role is to create all required tasks and to start them so the application can start running.
`my_app/source/my_app.c`	This file contains the core of the `my_app` application that does custom event processing. This file is the main source file where the user shall focus on his application and in which he/she will add code to process all of the user defined events.
`my_app/source/my_app.h`	Protected header file related to the above mentioned `my_app_task.c`
`my_app/source/my_app_task.c`	Contains functions called from the `nvm_main.c` file to create and manage the application QP/C tasks (creation, execution, …). Beyond the tasks creation and activation dedicated functions, this file also contains functions which contain the processing to be done based on the application state. For example, it contains the idle function in which all events are processed when the scheduler is in idle mode. It also contains the function that defines the initial processing after the task is created, and some other functions related to the scheduling states management. See the Figure 7-6 showing the functions call hierarchy.
`my_app/includes/my_app_task.h`	Header file declaring the function prototypes defined in the above mentioned `my_app_task.c`
`my_app/includes/my_app_signals.h`	Header file defining the various user defined signals that will be processed in the app by all the tasks
`ble/source/ble.c`	Contains data structure and functions that are used to configure and to initialize the Bluetooth LE part. For example, it contains name that is advertised by the device, along with all Bluetooth LE related parameters (intervals, number of attempts, …). This file should be modified to match the end user need.
`ble/source/ble.h`	Header file related to the above mentioned `ble.c` containing variable definition and function prototypes.
`ble/source/ble_task.c`	File containing all functions used to create Bluetooth LE dedicated tasks. The goal is to define functions that wrap the Bluetooth LE stack API functions call underneath.
`ble/includes/ble_task.h`	Header file related to the above mentioned `ble_task.c` file.

Table 7-3: Folder structure and files description for the EMB HRS sample application

This structure exposed above is the one defined for the sample applications delivered within the SDK. It is good practice to follow this recommendation even if this is not mandatory. In any case, if the final folder structure is different, the `CMakeLists.txt` file will have to be updated accordingly.

It shall be also noted that the `ble` folder (and its content) is present only if the application has Bluetooth LE connectivity capabilities.

In the files listed in Table 7-3, the locations where the user code shall be written are identified with a comment. Refer to the source code of this sample app to get more detailed information.

The Figure 7-6 shows where the idle processing function is located. It also shows that at some points, it calls the user defined event handling function in which the user would focus to process all of its specific events.

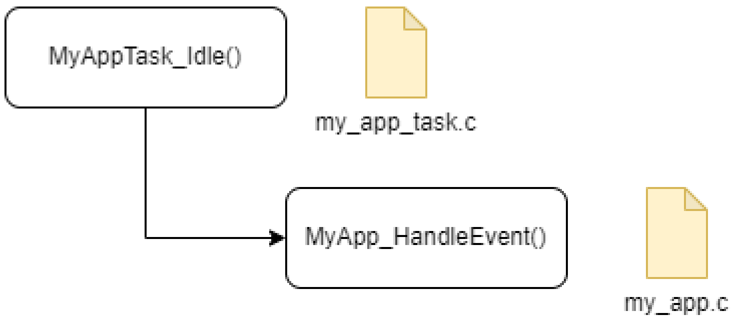


Figure 7-6: Hierarchy of idle processing call

7.6 STARTING FROM AN EXISTING EXAMPLE

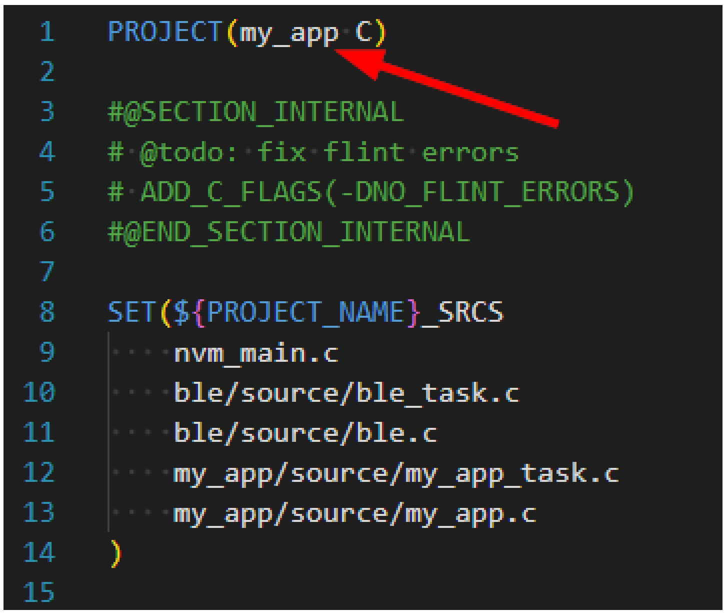
The easiest and fastest way of creating an application is to start from the existing example that best matches the end user application's goals.

It consists of copying an example folder (e.g. the one exposed in §7.5) and pasting it under the project directory by assigning a new name.

```
# cd <SDK_INSTALLATION_PATH>/projects
# cp -pr nvm_emb_hrs my_app
```

Inside the newly created folder, the project name shall then be changed from `nvm_emb_hrs` to `my_app` (or anything else that fit your needs) in the `<SDK_INSTALLATION_PATH>/projects/my_app/CMakeLists.txt` file.

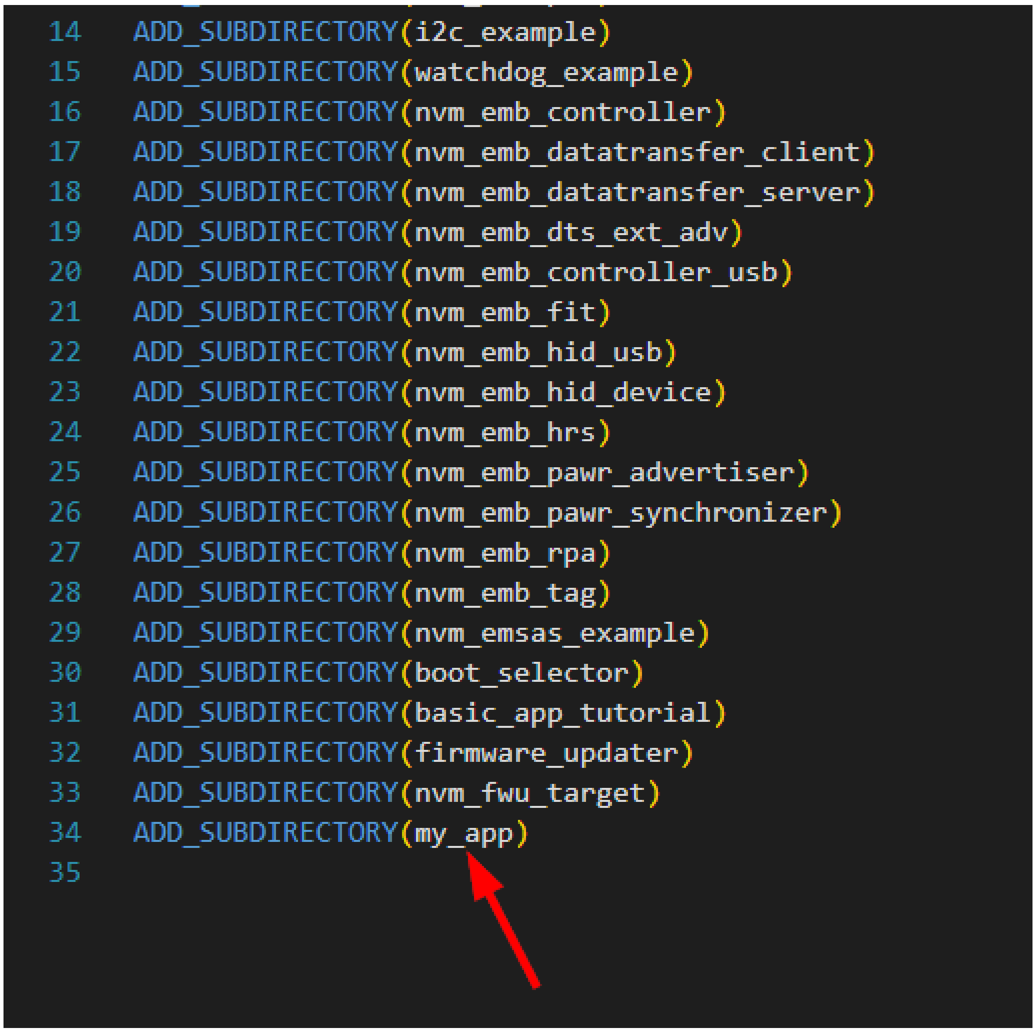
The Figure 7-7 shows the first line of the `CMakeLists.txt` file coming from the `nvm_emb_hrs` project where the name of the application has been changed in the `PROJECT` statement.



```
PROJECT(my_app C)

#@SECTION_INTERNAL
# @todo: fix flint errors
# ADD_C_FLAGS(-DNO_FLINT_ERRORS)
#@END_SECTION_INTERNAL

SET(${PROJECT_NAME}_SRCS
    nvm_main.c
    ble/source/ble_task.c
    ble/source/ble.c
    my_app/source/my_app_task.c
    my_app/source/my_app.c
)

```

Figure 7-7: Changing existing project name to a new one

Then, for the new project to be considered by the build system, the new project shall be added into the `<SDK_INSTALLATION_PATH>/projects/CMakeLists.txt` as depicted in Figure 7-8:



```
ADD_SUBDIRECTORY(i2c_example)
ADD_SUBDIRECTORY(watchdog_example)
ADD_SUBDIRECTORY(nvm_emb_controller)
ADD_SUBDIRECTORY(nvm_emb_datatransfer_client)
ADD_SUBDIRECTORY(nvm_emb_datatransfer_server)
ADD_SUBDIRECTORY(nvm_emb_dts_ext_adv)
ADD_SUBDIRECTORY(nvm_emb_controller_usb)
ADD_SUBDIRECTORY(nvm_emb_fit)
ADD_SUBDIRECTORY(nvm_emb_hid_usb)
ADD_SUBDIRECTORY(nvm_emb_hid_device)
ADD_SUBDIRECTORY(nvm_emb_hrs)
ADD_SUBDIRECTORY(nvm_emb_pawr_advertiser)
ADD_SUBDIRECTORY(nvm_emb_pawr_synchronizer)
ADD_SUBDIRECTORY(nvm_emb_rpa)
ADD_SUBDIRECTORY(nvm_emb_tag)
ADD_SUBDIRECTORY(nvm_emsas_example)
ADD_SUBDIRECTORY(boot_selector)
ADD_SUBDIRECTORY(basic_app_tutorial)
ADD_SUBDIRECTORY(firmware_updater)
ADD_SUBDIRECTORY(nvm_fwu_target)
ADD_SUBDIRECTORY(my_app)

```

Figure 7-8: Adding a new project in CMakeLists.txt

For this new project to be part of the build environment as a target (along with other existing examples), the `init.bat` script is provided at the root of the SDK. By executing this script, the build folder will be created and populated with all required files to build the different targets.

This script is a straightforward convenient way to create the build environment. For those who are curious, the main operations achieved by this script are listed below and consist in creating the folder and invoking the `cmake` tool with relevant parameters.

```
# cd <SDK_INSTALLATION_PATH>/build
# cmake -G"Unix Makefile" -Wno-dev -DCMAKE_BUILD_TYPE=debug –
                                  DCMAKE_TOOLCHAIN_FILE="cmake/toolchains/arcv2em.cmake" ..
```

The Figure 7-9 shows the output of the `init.bat` script execution.

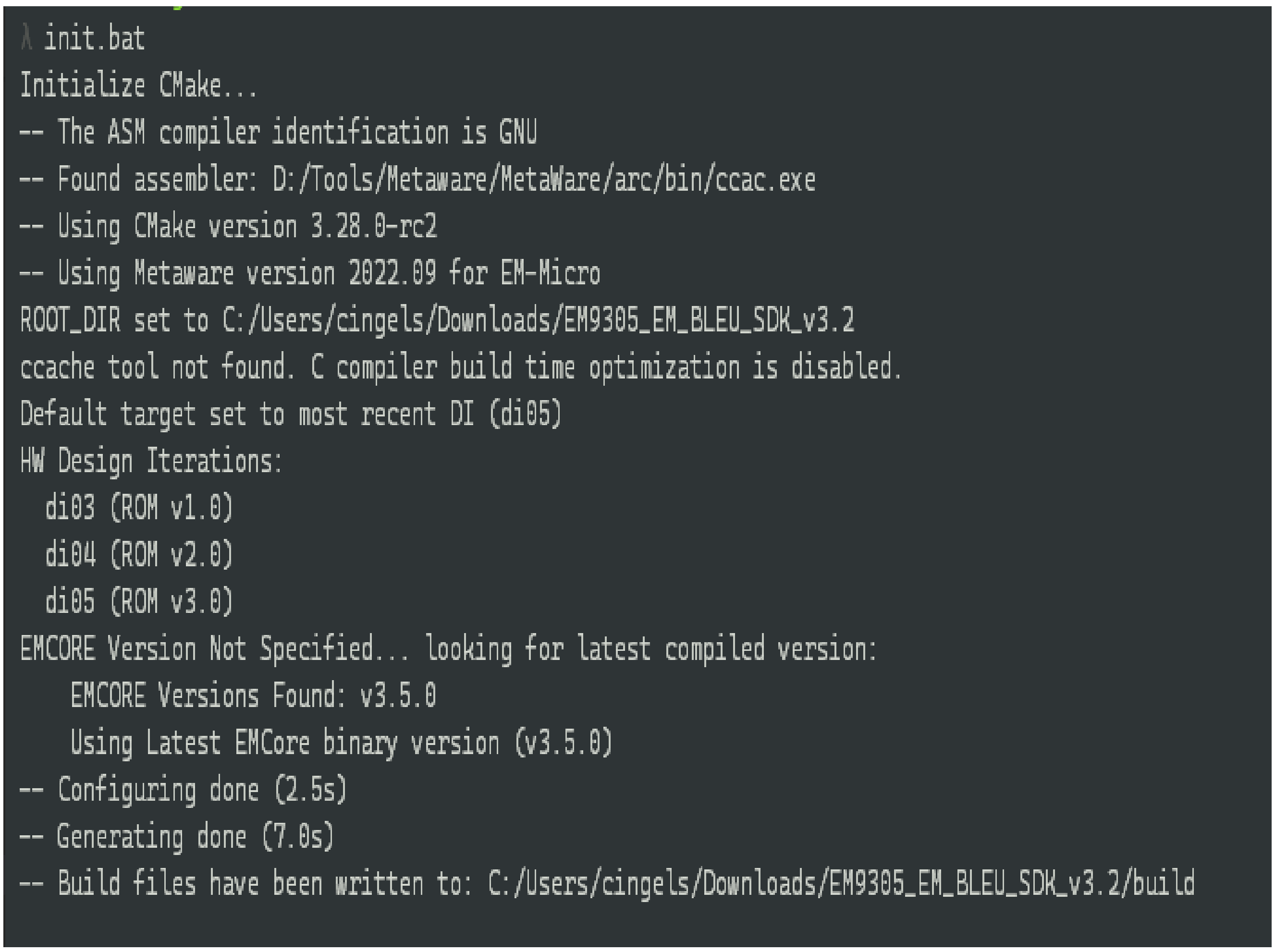

```
λ init.bat
Initialize CMake...
-- The ASM compiler identification is GNU
-- Found assembler: D:/Tools/Metaware/MetaWare/arc/bin/ccac.exe
-- Using CMake version 3.28.0-rc2
-- Using Metaware version 2022.09 for EM-Micro
ROOT_DIR set to C:/Users/cingels/Downloads/EM9305_EM_BLEU_SDK_v3.2
ccache tool not found. C compiler build time optimization is disabled.
Default target set to most recent DI (di05)
HW Design Iterations:
  di03 (ROM v1.0)
  di04 (ROM v2.0)
  di05 (ROM v3.0)
EMCORE Version Not Specified... looking for latest compiled version:
    EMCORE Versions Found: v3.5.0
    Using Latest EMCore binary version (v3.5.0)
-- Configuring done (2.5s)
-- Generating done (7.0s)
-- Build files have been written to: C:/Users/cingels/Downloads/EM9305_EM_BLEU_SDK_v3.2/build
```

Figure 7-9: Creating the build environment

The new target related to the newly added project is now part of the build environment. This means that to build it, the following command can be issued:

```
# cd build
# cmake –build . –target my_project
```

There is a possibility to create a full list of all available targets by issuing the following command:

```
cd <SDK_INSTALLATION_PATH>/build
cmake –build . –target help > targets_list.txt
```

The file `targets_list.txt` will then contain all available targets that can be built in the newly generated build environment.

7.7 EXPLAINING THE CMAKELISTS.TXT FILE

This paragraph deals with the various statements that appear in the `CMakeLists.txt` file to understand how the end user application is built. From the `nvm_emb_hrs`, some of the statements found in this file are already addressed in §6. They will be extended in this chapter to cover this complete example.

The first statement is the `PROJECT` statement. It introduces the project name and obviously the main part of the final executable file name without the prefix. It also specifies the programming language used.

```
PROJECT(my_app C)
```

The list of C source files is introduced with the creation of a dedicated variable.

```
SET(${PROJECT_NAME}_SRCS
      nvm_main.c
      ble/source/ble_task.c
      ble/source/ble.c
      my_app/source/my_app_task.c
      my_app/source/my_app.c
)
```

This example lists all the C source files that are compiled to build the application. They are organized into folders according to their respective functions.

As already mentioned in Table 7-3, the `./ble` folder contains all the functions dedicated to the creation of the Bluetooth related tasks. Thus, C source files and headers are split and stored in their respective `./ble/source` and `./ble/include` folders.

The `./my_app` folder is dedicated for storing the end user source code files that are the core of the end user application. Like the `./ble` folder, the application source code is split between a `./my_app/source` and a `./my_app/include` folders for C source files and header files.

The `nvm_main.c` file which contains the so called `NVM_ConfigModules()` and `NVM_ApplicationEntry()` functions is located at the top of source tree.

Regarding source code files inclusion, it can sometimes be more convenient to let `CMake` search recursively for any C source file. In this case, the above statement can be replaced with the following one:

```
FILE(GLOB_RECURSE ${PROJECT_NAME}_SRCS *.c)
```

It instructs CMake to do a global recursive look for all C source files and to put the result into the variable `${PROJECT_NAME}_SRCS`. If the application name is `my_app`, then the variable `my_app_SRCS` will be a list containing all the source files listed or found.

In the same manner, it is possible to do a recursive search for all header files with the following statement:

```
FILE(GLOB_RECURSIVE ${PROJECT_NAME}_HEADERS *.h)
```

The search result will be pushed into the `${PROJECT_NAME}_HEADERS` variable.

In this scenario, all found C and H source files will be included.

By using these statements, the CMake tool will also populate the following variable:

`PROJECT_SOURCE_DIR` with `<PROJECT_NAME>_SOURCE_DIR`

The second step is to introduce the libraries that are needed to link with the application as shown in the following code snippet.

```
SET(${PROJECT_NAME}_LIBS
    ${NVM_COMMON_LIBS}
    QPC
    emb_peripheral
    emdt_service
    emdt_profile_server

)
```

The list of libraries to use depends on the final goal of the application. In this example, the “embdt” libraries are used to create an EM Data Server application. Moreover, the “emb_peripheral” library is added here so the end user application will behave as a peripheral from the Bluetooth LE stand point.

These libraries are provided in the SDK in the form of an archive file with the extension “.a”.

The `NVM_COMMON_LIBS` variable is defined in the following file:

`<SDK_ROOT>/cmake/common/common_libs.cmake`

It contains a list of common libraries that are always part of an end user executable for building a basic application without Bluetooth LE capabilities. See Appendix 2 for more details on these common libraries.

The `QPC` library is required to have access to the QP/C framework functions which provide real time capabilities and events management processing. Note that since the end user may decide to develop a bare metal application, in such a case there would be no need to integrate QP/C library and then it can be safely removed from the above statement. Otherwise, it must be included so the QPC library is used as part of the linker requirements.

If other libraries are needed, they must be added to this list to be taken into account by the build process.

The last variable to be defined is the list of folders that contain the requested headers.

As you may have noticed, the SDK contains folders with header files within. For example, if the `i2c` library is needed, then the related `i2c` folder should be included.

```
SET(${PROJECT_NAME}_INCLUDES
     ${EMB_COMMON_INCLUDES}
     my_app/includes
     ble/includes
     i2c/includes
     ${LIB_DIR}/emb_vs/emdt/profile/server/includes
     ${LIB_DIR}/emb_vs/emdt/service/includes
)
```

The `EMB_COMMON_INCLUDES` variable is a wrapper that includes all folders containing header files provided in the SDK. It is defined in the following file:

`<SDK_ROOT>/libs/third_party/emb/cmake/common/includes.cmake`

Including this variable automatically includes a set of folders containing files related to the Bluetooth LE stack. This eliminates the need for the end user having to include each required folder individually.

Building a Bluetooth LE based application and including header files and libraries related to this stack implies including QP/C as well since all timings for Bluetooth link layer protocol compliance require accurate timings. So, it is not possible to create a Bluetooth LE based application without relying on QP/C. However, a non-Bluetooth LE based application can be a simple bare metal application with no link to QP/C.

It might also happen that an end user application has local header files located in the application folder or in a local subfolder like the `ble/include` folder in the above example. The relative path is simply added so it is part of the variable definition.

Generally speaking, any local folder containing header files shall be added.

Another way of building the list of header files is to ask `CMake` to do a recursive search, looking for any *.h file. Any file that is found is added to this list. Such statement can be the following:

```
FILE(GLOB_RECURSIVE ${PROJECT_NAME}_INCLUDE *.h)
LIST(PREPEND ${PROJECT_NAME}_INCLUDE ${EMB_COMMON_INCLUDE})
```

The list is first filled with header files found during the recursive search. Then, the `${EMB_COMMON_INCLUDE}` variable is prepended to this list. Consequently, the search for the header files will start with the paths defined in the `${EMB_COMMON_INCLUDE}` variable, and then might continue to the recursively found header files in the local folder.

It is often recommended to gather local header files in local dedicated folders, let's say the `include` folder, to keep things well organized and separated from each others.

Note that the `${CMAKE_CURRENT_SOURCE_DIR}` variable is automatically created by CMake. It represents the absolute path to where the source files of the application are located. It can eventually be used in the end user `CMakeLists.txt` file.

Following the heart rate sensor sample application, the next block defines the following:

1. where to put the binary (in non-volatile or in volatile memory – NVM or RAM)
2. executable to be built from the listed source files
3. libraries to link with (e.g. QP/C, EMB lib, …)
4. include directories to use

```
❶ APP_IN_NVM()
❷ ARC_EXECUTABLES(${PROJECT_NAME} ${${PROJECT_NAME}_SRCS})
❸ ARC_LINK_LIBRARIES(${PROJECT_NAME} ${${PROJECT_NAME}_LIBS})
❹ ARC_INCLUDE_DIRECTORIES(${PROJECT_NAME} ${${PROJECT_NAME}_INCLUDES})
```

In this block, the `APP_IN_NVM` is a CMake macro that will map the final application in the NVM between address `0x30_2000` and `0x36_0000`. This macro is defined in the following file:

`<SDK_ROOT>/cmake/macros/macros_targets_executable.cmake`

By default, and without any specific directive, the example above will instruct the build process to create a standalone application.

Another possibility is to replace this macro by the `APP_IN_IRAM` macro which will map the application in the iRAM instead of the NVM like in the example below:

```
❶ APP_IN_IRAM()
❷ ARC_EXECUTABLES(${PROJECT_NAME} ${${PROJECT_NAME}_SRCS})
❸ ARC_LINK_LIBRARIES(${PROJECT_NAME} ${${PROJECT_NAME}_LIBS})
❹ ARC_INCLUDE_DIRECTORIES(${PROJECT_NAME} ${${PROJECT_NAME}_INCLUDES})
```

With this statement, the application will be mapped in IRAM at an address between `0x18_4000` and `0x18_BFFF`, also depending on how the RAM blocks are configured. This is the case for RAM0 block which is configured by default as an instruction RAM block (IRAM0), so any example that is using this statement will be located in this area.

Otherwise, it is possible to map the final executable to be executed from the NVM (addresses above 0x300000). This can be achieved by replacing the `APP_IN_IRAM()` macro by the `APP_IN_NVM()` macro (see example below).

The `ARC_EXECUTABLE` macro defines the final executable to build along with all the source files to compile and to link together.

The `ARC_LINK_LIBRARIES` macro introduces the list of libraries the build process will link with to get the final executable. It uses the variable `${PROJECT_NAME}_LIBS` defined earlier.

And finally, the `ARC_INCLUDE_DIRECTORIES` introduces the list of include folders. It uses also the variable `${PROJECT_NAME}_INCLUDES` defined earlier.

It is also possible to build an EM-Core based application as explained in §7.2.3. To achieve that, a dedicated `cmake` macro is provided with the SDK and shall be added to the local `CMakeLists.txt` file like shown in the code extract below:

```
APP_IN_NVM()
ARC_EXECUTABLES(${PROJECT_NAME} ${${PROJECT_NAME}_SRCS})
ARC_LINK_LIBRARIES(${PROJECT_NAME} ${${PROJECT_NAME}_LIBS})
ARC_INCLUDE_DIRECTORIES(${PROJECT_NAME} ${${PROJECT_NAME}_INCLUDES})

## Create the EM-Core application base target
GENERATE_EMCORE_APPLICATION(${PROJECT_NAME}_emcore standard.v3.5.0 ${PROJECT_NAME}_SRCS
${PROJECT_NAME}_INCLUDES "" "")
```

This macro introduces the EM-Core “flavour” to be linked with. It takes several parameters, and its prototype is given here:

```
GENERATE_EMCORE_APPLICATION
(
     <target name>
     <EM-Core flavour + version>
     list of source files
     include files
     flags
     list of libraries to link with
)
```

The Table 7-4 describes each of the macro parameters.

Parameter	Description
target name	This will be the name of the final executable. Usually, it has the same name as the standalone executable version with a distinguished name appended, like “_emcore” for

	example to highlight the fact that it is an EM-Core based application, vs the standalone one.
	So, an example of final application name could be:
	`nvm_emb_hrs_emcore`
	while the standalone application's name would be:
	`nvm_emb_hrs`

EM-Core flavor	The EM-Core flavor comes into the "standard" version not embedding any Bluetooth LE stack. It is up to the end user to define if it is used or not. In all cases, either it is actually used or not, the end user application shall embeds the Bluetooth LE stack for implementing Bluetooth LE operations.
List of source file	This is the exact same source files list that is specified in the `CMake` variable `${PROJECT_NAME}_SRCS` used to build the standalone application. So it can be repeated here.
Include files	This list can be the same as the one specified in the `CMake` variable `${PROJECT_NAME}_INCLUDES` to build the standalone application. So it can be repeated here.
Flags	Through this option, it is possible to provide specific flags used for building this EM-Core based version. But it can be let empty here by specifying empty double quotes "".
List of libraries	Since EM-Core already embeds some libraries, this list here might be shorter than the one provided for the standalone application. It is however possible to enforce a library that is already provided in EM-Core to take into account a new version for example. In such a case, the explicitly specified library in this parameter will override the one provided in EM-Core.

Table 7-4: Parameters description for the GENERATE_EMCORE_APPLICATION macro

ⓘ At the time this guide is written, the most up-to-date EM-Core version is the **4.1.0**.

8. MEMORY ORGANIZATION

8.1 OVERVIEW

The ARC EM7D CPU is connected to a read only memory containing the startup code, to a read and write non-volatile memory for permanent data and code storage, and to a volatile memory used during application execution and for data retention over sleep cycles. The following paragraphs dive into these different types of memory, giving more technical details that might be useful for implementing an end user application.

8.1.1 ROM

This memory has a size of 64kB and contains the device startup code and some essential drivers. It can be neither overwritten nor erased which guarantees that the device will always be able to startup.

It contains a set of functions available for any end user application which avoids an application having to embed these functions in the final binary file. This saves a little bit of space for the application.

Moreover, it provides a light application that is executed in some circumstances or when no valid end user application is found in the non-volatile memory (or if the non-volatile memory is empty). Executing this basic application is called “running in configuration mode” and in this mode, it accepts a limited set of commands received through the transport layer which can be the SPI or the UART.

It can also start in this mode if it finds a corrupted application image in the non-volatile memory and if there is no other valid application to start, or if the data stored in the information pages 2 or 3 are corrupted.

The Figure 4-2 in §4 provides the detailed ROM boot sequence in which some branches lead to starting the configuration mode:

- configuration mode is actually requested
- trimming values on information page 3 are missing or the data is corrupted
- no valid entry point found in non-volatile memory

See the §4 to get more details on the device startup sequence. Here, the main point to keep in mind is that the ROM contains the startup sequence that is always executed in all conditions when the device starts up.

The Table 8-1 provides details on the ROM.

ROM block	Start address	Size	Usage
ROM	0x100000	64kB	Initialization and startup sequence

Table 8-1: ROM block description

8.1.2 RAM

The volatile memory, or RAM, is physically made of blocks that can contain either instruction code or data. Some of them can only be used for storing data, some others can only store instruction, and few of them can be configured to be connected either to the data or instruction bus, but not at the same time!

The Table 8-2 lists the available physical RAM blocks, their size and how they are configured. For those which type is “as per configuration”, this means that they can be configured by software. The “startup config” column indicates how the configurable blocks are configured at device power on reset.

RAM Block	Start address	Size	Type	Startup config
RAM0	`0x800000`[5]	4kB	as per configuration	Instruction (IRAM0)
RAM1	`0x801000`	4kB	Data	DRAM1

[5] The address `0x800000` is valid to access RAM0 when it is configured as data RAM (DRAM). Otherwise, and by default, its access address is `0x1FF000`. See Table 8-3 for more details on instruction RAM block addressing.

RAM2	`0x802000`	4kB	Data	DRAM2
RAM3	`0x803000`	4kB	Data	DRAM3
RAM4	`0x804000`	16kB	Data	DRAM4
RAM5[6]	`0x808000`	16kB	as per configuration	Data (DRAM5)
RAM6[7]	`0x80C000`	16kB	as per configuration	Data (DRAM6)

Table 8-2: RAM blocks configuration

Although the ARC architecture has physically separate instruction and data memories, along with peripheral registers, are mapped to the same address space, but with different address ranges. The peripheral address space provides memory mapped access to peripherals on the internal AHB bus. The auxiliary address space is a separate address space, which is used by the security and encryption libraries.

The naming convention of the RAM blocks depends on how they are configured. As shown in Table 8-2, when all RAM blocks that can be configured as instruction RAM are configured accordingly, they are named IRAM0, IRAM1 and IRAM2.

The Table 8-3 shows the naming convention for the RAM blocks configured as instruction RAM, with the corresponding memory address and respective size.

RAM Block	Address range	Size	Naming convention
RAM0	`0x1FF000…0x1FFFFF`	4kB	IRAM0
RAM5	`0x184000…0x187FFF`	16kB	IRAM2
RAM6	`0x180000…0x183FFF`	16kB	IRAM1

Table 8-3: Instruction RAM blocks naming convention and mapping

Switching a RAM block from DRAM to IRAM can be done with this code snippet that has to be inserted at the early beginning of the `NVM_ApplicationEntry()` entry point.

In this example, the RAM0 and RAM5 blocks are configured as instruction RAM blocks. Thus, their memory mapping is changed to match the one indicated in Table 8-3.

```
#include "types.h"
#include "common.h"
#include "mem_utils.h"
#include "macros.h"
#include "nvm_entry.h"
#include "em_system.h"

NO_RETURN void NVM_ApplicationEntry(void)
{
      IRQ_EnableInterrupts();
      # Configure RAM0 and RAM5 blocks as instruction blocks.
      SYS->RegMemCfg.r32 |= MEM_DRAM5_IN_ICCM(1) | MEM_DRAM0_IN_ICCM(1);
```

[6] Refer to Table 8-3 when configured as instruction RAM

[7] Refer to Table 8-3 when configured as instruction RAM

```
      …
}
```

The SYS variable is automatically defined as a pointer to the system registers starting at address @0xF03800. It can be directly used to access all system related registers.

By default, the RAM0 block is configured as an instruction RAM block. However, it can be configured to store data if the others data blocks do not fit the main application need.

Converting this instruction RAM block to a data RAM block can be done programmatically with the following code snippet.

```
NO_RETURN void NVM_ApplicationEntry(void)
{
      # Configure RAM0 block as DRAM0
      SYS->RegMemCfg.r32 &= ~MEM_DRAM0_IN_ICCM(1);
      # Read current RAM retention configuration
      uint8_t value = PML_GetRamRetentionEnable();
      # Activate RAM retention for RAM0 block
      value |= PML_DRAM_RET_ON_R(1);
      # Write new RAM retention configuration
      PML_SetRamRetentionEnable(value);
}
```

Pay attention to the “~” sign to invert the bits to be written to the RegMemCfg register.

Once configured, the RAM0 block can be used to store data.

The Figure 8-3 shows the content of the RAM retention configuration register. Each bit controls the power domain of one RAM block. Several bits can be Ored to control several power domains at one time.

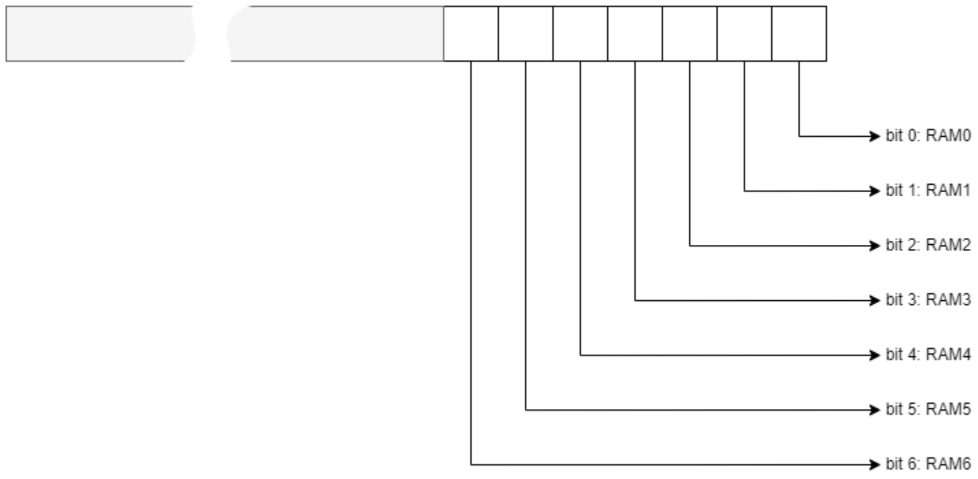


Figure 8-1: RAM retention configuration register

The document [3] describes the solution exposed above to reconfigure IRAM0 block into a DRAM0 block.

Note that in case this RAM block is configured to store data, it will still not be managed by the memory manager which sticks only to DRAM1 to DRAM6 blocks. So the DRAM0 block has to be fully handled by the end user application. As explained in the above code snippet, if data retention is needed, it has to be manually activated for this block.

8.1.3 NVM organization

The NVM is the permanent storage memory which can be erased and reprogrammed several times. It is divided in two parts as depicted in Table 8-4.

NVM part	Address range	Size	Type
NVM0	0x30_0000…0x37_FFFF	512kB	Instruction
NVM1	0x40_0000…0x40_7FFF	32kB	Information pages

Table 8-4: NVM blocks configuration

The biggest part is reserved for storing code, or applications that can be executed by the device. It is an “execute in place” process in which the CPU instructions are directly read and executed from their original location in NVM. No copy from NVM to RAM takes place prior to execution.

The second NVM part (the smallest one located at the highest addresses) is fully dedicated to store factory and user data that can influence the way the device works. It is also in this area that security keys are stored in a dedicated key container that is write only.

8.2 DATA RETENTION

8.2.1 Overview

Data retention refers to the fact that a variable retains its value over “active to sleep” and “sleep to active” transitions. This is what is otherwise called sleep mode.

Variables declared in retention memory retain their values when in sleep mode while variables declared in other RAM blocks (non-retention memories) do not maintain their value.

Retaining data content is done by keeping one (or more) 4kB/16kB RAM block powered while the sleep mode is activated. All variables mapped in this RAM block will keep their values which will be reused when the system is resuming from sleep mode. The content of this RAM block can be considered as a snapshot of the software’s current state just before switching to sleep.

8.2.2 Data retention during sleep mode

By default, all variables declared in a project with no specific keyword are persistent, which means that their content is maintained while in sleep mode. However, for clarity’s sake, this keyword can also be enforced by using the keyword `SECTION_PERSISTENT` even if it is not mandatory.

Any variable declared with the `SECTION_PERSISTENT` keyword will be mapped at the beginning of the RAM in the persistent area. From the Figure 8-2, such variables would be mapped at the beginning of the RAM, for example at address `0x801348`.

If data retention is not required, the declaration macro `SECTION_NONPERSISTENT` shall be used to reduce the use of persistent memory and save power while in sleep mode under specific conditions. By using as few persistent RAM blocks as possible, the power consumption can be reduced.

The following example declaration shows the use of the non-persistent declaration macro applied to an array of 256 bytes.

```
uint8_t SECTION_NONPERSISTENT buffer[256] ;
```

This code snippet declares an array of 256 unsigned bytes which will be mapped in DRAM but outside of the persistent RAM block. This means that this array will lose its content each time the system will enter sleep mode because the DRAM block which contains it will be powered OFF just before switching to sleep mode. In the Figure 8-2, this variable would be mapped in the non-persistent memory area at the end of the RAM (e.g. at address `0x80ffd0`).

The `SECTION_NONPERSISTENT` keyword maps the variable in a dedicated section called `non_persistent` and is declared like this:

```
#define SECTION_NONPERSISTENT SECTION("non_persistent")
```

Powering a DRAM block and maintaining it ON over sleep cycles is useful to store data that needs to be retained. This is why all DRAM blocks can be manually powered ON (or OFF) using the following function call during the application startup, preferably in the `NVM_ConfigModule()` function.

```
PML_SetRamRetentionEnable (ramRetention);
```

This way, each block of RAM power supply can be individually controlled by the end user application. This local configuration can be done after the memory manager has configured which RAM blocks need to be powered ON or OFF depending on the memory usage.

In any case, the end user configuration takes precedence over the one set by the memory manager.

The RAM block parameter can be one value among the list given in Table 8-5.

Constant	RAM block
PML_DRAM0_RETENTION	DRAM0
PML_DRAM1_RETENTION	DRAM1
PML_DRAM2_RETENTION	DRAM2
PML_DRAM3_RETENTION	DRAM3
PML_DRAM4_RETENTION	DRAM4
PML_DRAM5_RETENTION	DRAM5

Table 8-5: List of symbols to drive DRAMx power supply

The above mentioned constants can be OR'ed to enable retention for more than one RAM blocks using one single call to the function `PML_SetRamRetentionEnable()` like shown in the following code snippet:

```
PML_SetRamRetentionEnable (PML_DRAM0_RETENTION | PML_DRAM2_RETENTION |
                                                             PML_DRAM3_RETENTION);
```

The constants used above and the function prototype are defined in the file `<SDK>/libs/pml/includes/pml.h`.

It is possible to know the RAM blocks retention configuration by issuing a call to the function `PML_GetRamRetentionEnable()`. The returned value shall be AND masked with the constants defined in Table 8-5 to isolate each bit to actually know if retention is enabled for one specific RAM block.

The code snippet below showcases an example on how to achieve this:

```
uint8_t status;
status = PML_GetRamRetentionEnable();
// Test DRAM2 retention enable flag
if (status & PML_DRAM2_RETENTION)
{
      // RAM block 2 retention is enabled.
}
```

8.2.3 *Data retention during deep sleep mode*

In this mode, the CPU is powered off. Thus, there is by default no data retention since no RAM block is kept powered ON. However, it is still possible to configure some RAM blocks to stay powered during deep sleep. But in the default configuration and from a software stand-point, resuming from a deep sleep mode is the same as starting from a cold reset with the exception that the PML bits keep the value they had before sleeping.

The main advantage of this mode is that the current consumption is very low (refer to [1] for more detailed). A typical use case can be to keep one RAM block powered to contain data during a long device storage period until it is commissioned. Then the device exits its deep sleep mode and starts working in a nominal way by retrieving values or configuration that was saved during this deep sleep period.

It is possible to exit from the deep sleep mode by using any available GPIO as an interrupt signal or by the sleep timer or through a QDEC event. Thus, it is possible to configure one or more RAM blocks for data retention before switching to this mode. Consequently, the deep sleep mode is not different than the simple sleep mode. The only difference is the clock used when switching to the deep sleep mode. In this mode, the LF RC low power clock running at 100 kHz is used. Note that this clock is not accurate enough (50% accuracy vs 10% accuracy for the 500 kHz clock used in sleep mode) and thus is not suitable for Bluetooth LE operations. Consequently, if a Bluetooth LE connection must be maintained but saving power is important, then the device shall not be switched to the deep sleep mode. Instead, it shall activate the sleep mode in which the LF RC clock running at 500 kHz is used. It is not recommended to switch to deep sleep mode if it is planned to do Bluetooth LE operations.

8.2.4 *Data retention after software reset*

Unlike the sleep mode, a software reset is a process in which the CPU is instructed to restart without any power cycle. While the RAM blocks are not powered OFF, the data they contain are reset to their initial values during the software initialization stage.

When a variable is defined in the end user application with an initial value, it is then stored in the `.data` section and initialized to its specified value at startup. Otherwise, a variable can be let uninitialized and then stored in the `.bss` section. It is then automatically initialized to 0 at startup. However, there is a third possibility where a variable is stored in a non-persistent RAM block but not initialized to 0 at startup after a software reset (e.g. triggered by the watchdog). To do so, the statement `SECTION_NP_NOINIT` shall be used in order to declare the variable, as shown below:

```
static SECTION_NP_NOINIT <type> <variable>;
```

A typical example could be the following one:

```
static SECTION_NP_NOINIT uint8_t gQpcEventPool[16];
```

This example defines a small table for storing QP/C events. Its content will not be reset after a software reset. It will contain the values it had before switching to sleep mode so in this example, this queue process can be resumed.

The definition of `SECTION_NP_NOINIT` is in the file `<SDK>/common/includes/macros.h` and is the following:

```
#define SECTION_NP_NOINIT SECTION(".noinit")
```

The Table 8-7 summarizes the different ways to define a variable.

Statement	Behavior
User defined and initialized variable	section .data, variable initialized with the user defined value
User defined and uninitialized variable	section .bss, variable automatically initialized with 0 or null
User defined and uninitialized with noinit	section noinit, variable value is not initialized at all

Table 8-6: Different ways of variable definition and initialization

8.2.5 Conclusion

The Table 8-7 summarizes the data retention depending on the power save mode.

Mode	Data retention?
Software reset	No
Sleep mode	Yes (configurable)
Deep sleep mode	Yes (configurable)

Table 8-7: Data retention depending on the reset and sleep modes

8.3 IRAM RETENTION CONFIGURATION

The IRAM blocks are automatically turned ON when an application is programmed to the IRAM. Once enabled, the IRAM remains powered ON regardless of the sleep state. These blocks of RAM will revert to their default setting after a power cycle (ON ➡ OFF ➡ ON cycle).

8.4 MEMORY ALLOCATION

The implemented memory allocator does not work in the same way in which a “regular” memory allocation process works.

There is no heap that could be used to record allocated memory blocks. Moreover, there is no way to reserve and free memory blocks at run time as it might be thought of regarding how regular memory allocation work.

Instead, the memory allocation is a process by which the end user application will define the following:

- how many RAM blocks to configure for data retention (these blocks will be switched ON and kept ON over sleep periods)
- how many RAM blocks to use for non-retention data (these blocks will be switched ON but switched OFF during sleep periods.

So when it comes to address memory allocation, it is a way to define which RAM block will be switched ON.

All other RAM blocks are switched off even when the device is up and running to save energy.

Non-retention data blocks are powered up when the device is running but are switched off when the device goes to sleep mode.

And data retention blocks are always kept powered on, even when the device goes to sleep mode.

In deep sleep mode, all RAM block are switched off by default.

The main point to consider when using the memory allocator API is that it is agnostic from the RAM physical structure. This means that the end user application does not know, and does not need to know, what is the block organization and what is the size of each block.

Instead, the end user indicates during its application start-up configuration how many bytes are needed to be part of the persistent memory by a call to the following function:

```
void *Memory_AllocatePersistent( uint32_t numberOfBytes );
```

The memory allocator deals with the memory structure to know which blocks shall be configured as persistent memory blocks and which blocks shall be let as non-persistent ones.

Note that calling this function is done on top of already configured RAM block used for persistent data retention.

The persistent memory grows from the bottom to the top along with the allocation requests with the block size granularity, and the non-persistent memory blocks grow from the top to the bottom.

Following this rule and based on the number of bytes to be part of the persistent or part of the non-persistent memory, the memory manager defines which blocks shall be switched ON after a sleep period and switched OFF before switching to sleep, and which blocks shall be kept always ON over sleep modes switch cycles for data retention. Other blocks that will not contain any data will be kept switched OFF.

The Figure 8-2 shows how the memory allocator works.

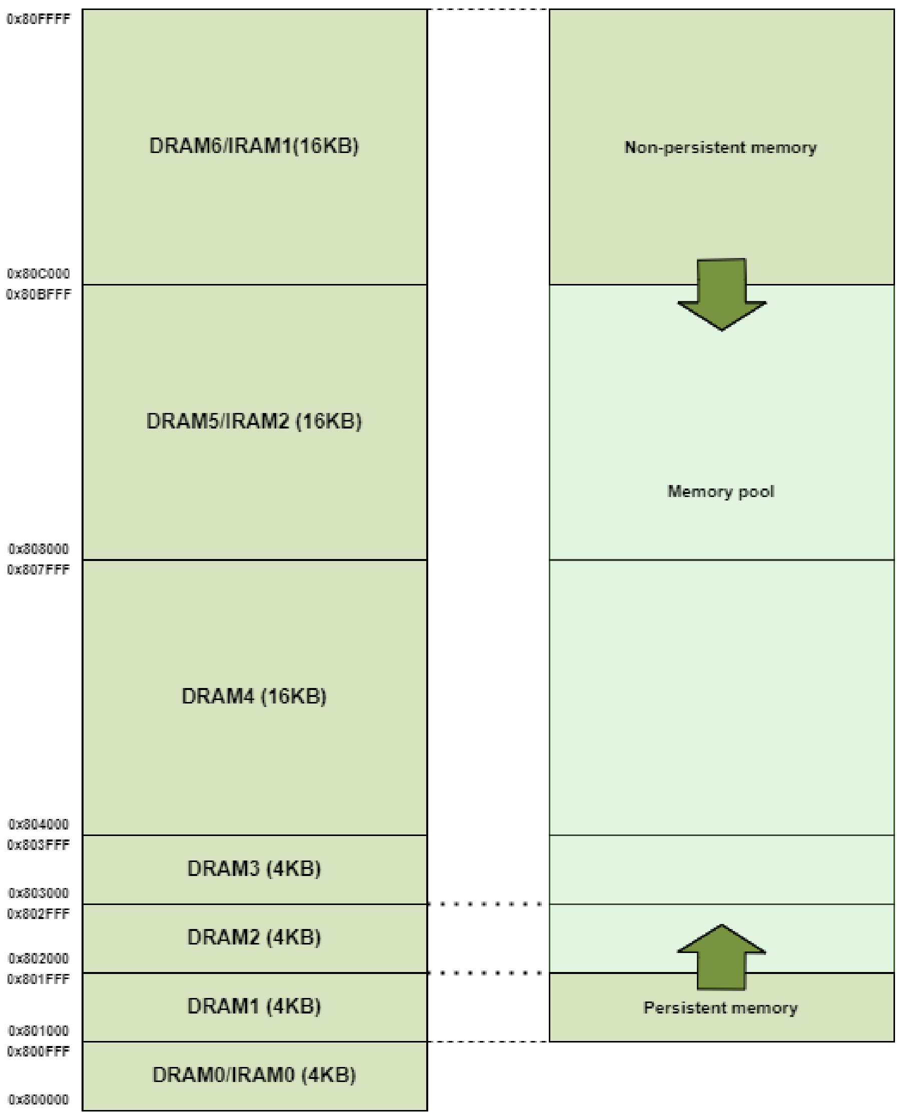


Figure 8-2: RAM layout

With this figure, it is easy to understand how the memory allocator works. For example, if the end user application requests less than 4kB of retention data, then only the DRAM1 block will be powered ON and kept powered even if the device goes to sleep. If the end user application needs 4kB (stored in DRAM1) plus at least 1B of retention memory, then the whole DRAM2 block will be selected as well and powered ON.

On the other side, if less than 16kB is requested to be stored but retention is not important, then the DRAM5 block will be powered ON when the device starts-up and powered OFF when the device goes to sleep mode.

Note that the DRAM0/IRAM0 block is always powered ON. As said earlier, its default configuration is to store instructions (so configured as ICCM). And as already said, its data retention configuration shall be managed by the end user application since it is not controlled by the memory manager.

The RAM blocks allocation is a process that shall be done at start-up. It allows unused RAM blocks to be powered off which saves energy during the product lifecycle.

If a RAM block is configured to be a persistent storage block, then it keeps this configuration during the product lifecycle. The only way to change it is to apply another configuration after power-on reset.

The memory stack used by function calls and local variables is in the non-persistent memory area and has a size of 4096 bytes by default. So, for example, it can range between addresses `0x80fb24` (top) and `0x80eb24` - persistent area, its content is lost when switching to sleep mode.

The Figure 8-3 illustrates the stack location at the end of the RAM.

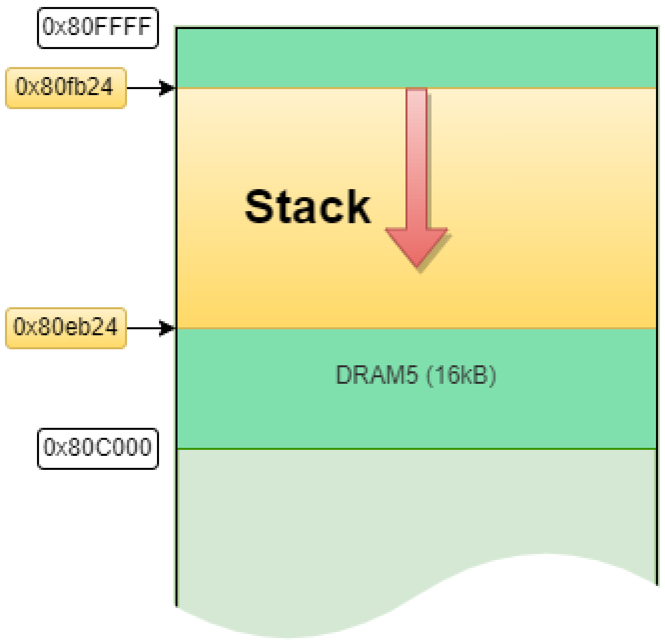


Figure 8-3: Memory stack location in RAM

The stack addresses exposed in this figure are just an example but the point to highlight here is that it is located above 0x80C000 address in the last physical RAM block.

The red arrow shows how the stack grows when used.

9. DEVICE PERIPHERALS

9.1 DMA

9.1.1 Overview

The EM9305 embeds a DMA controller configured and used at software level. It helps transferring data from different sources and targets. It covers the following configurations:

- from a peripheral to central memory
- from central memory to peripheral
- from central memory to central memory (in another memory location obviously ☺)

Having these types of transfers increases the chip performances for transferring big chunks of data without having the CPU actively involved. However, it shall be noted that using the DMA for very a low amount of data transfer degrades the performances so it is not always a good idea to constantly use it. It really depends on the amount of data to be transferred.

Due to the number of available peripherals and to the limited number of DMA channels, some peripherals share the same channels. This implies that these peripherals cannot be used at the same time for transferring data using the DMA channel.

9.1.2 Shared channels

The Table 9-1 gives the list of peripheral controllers where data exchange takes place between them and the EM9305 CPU core along with the DMA channel in use.

Peripheral	Channel	Channel's goal
SPI Slave Rx	5	Receive data from a device connected to the SPI
SPI Slave Tx	4	Transmit data to a device connected to the SPI
UART Rx	7	Receive data from a device connected to the UART
UART Tx	6	Transmit data to a device connected to the UART
Radio Rx	3	Receive data from the radio module (for Bluetooth)
Radio Tx	2	Transmit data from the radio module (for Bluetooth)
ADC	7	Data reception from the ADC
I2S Rx	7	Receive data from the I2S
I2S Tx	6	Transmit data to the I2S

Table 9-1: List of peripherals that exchange data with the EM9305

Bidirectional peripherals have two distinct DMA channels for sending data (Tx) to the peripheral and receiving data (Rx) from the peripheral. This is the case for all the peripherals except for the ADC where the DMA channel is only used for receiving the sampled data.

9.1.3 Configuration matrix

The Table 9-4 list the configurations that can be defined for being able to use the shared DMA channels without having any conflicts in such DMA channel allocations.

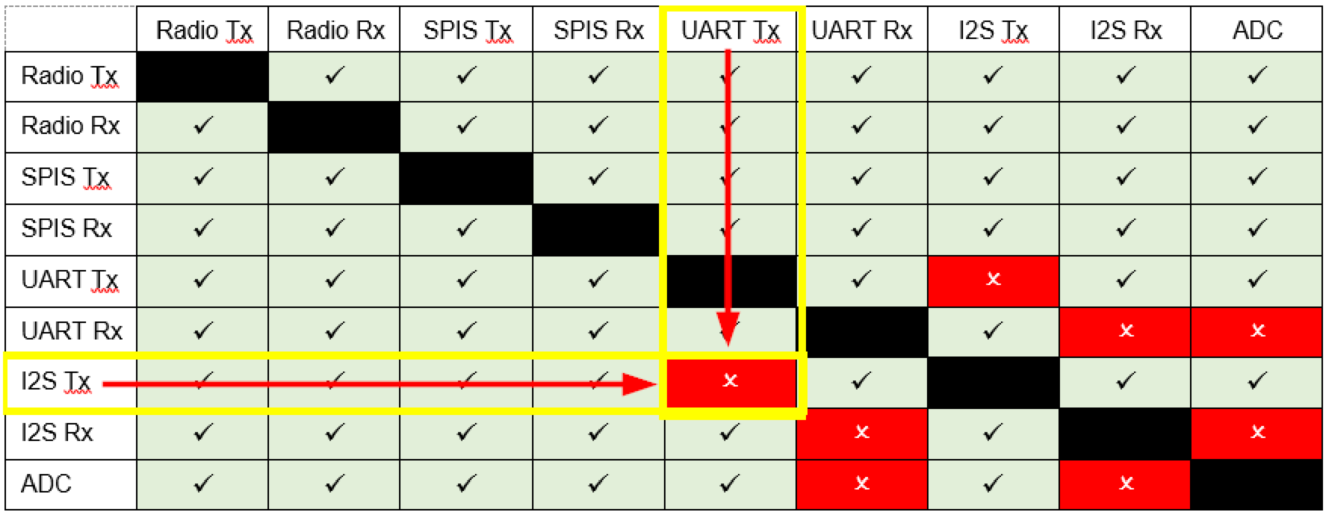
The green cells with a check at cross position between peripheral row and columns indicate that both peripherals can be used at the same time for DMA transfer. In contrast, the red cells with a cross indicate that both peripherals in the corresponding row and column cannot be used at the same time if DMA transfers are needed.

	Radio Tx	Radio Rx	SPIS Tx	SPIS Rx	UART Tx	UART Rx	I2S Tx	I2S Rx	ADC
Radio Tx	■	✓	✓	✓	✓	✓	✓	✓	✓
Radio Rx	✓	■	✓	✓	✓	✓	✓	✓	✓
SPIS Tx	✓	✓	■	✓	✓	✓	✓	✓	✓
SPIS Rx	✓	✓	✓	■	✓	✓	✓	✓	✓
UART Tx	✓	✓	✓	✓	■	✓	×	✓	✓
UART Rx	✓	✓	✓	✓	✓	■	✓	×	×
I2S Tx	✓	✓	✓	✓	×	✓	■	✓	✓
I2S Rx	✓	✓	✓	✓	✓	×	✓	■	×
ADC	✓	✓	✓	✓	✓	×	✓	×	■

Table 9-2: Peripheral DMA channel configuration matrix

This matrix shows that the SPIS Tx can be used with the UART Rx at the same time for example, since they do not share the same DMA channel.

On the contrary, the Table 9-3 shows an example where at the cross position between the I2S Tx row and the UART Tx column, a red cell containing a cross is shown. This means that both peripherals cannot use the DMA for transmitting data at the same time because they share the same DMA channel.



	Radio Tx	Radio Rx	SPIS Tx	SPIS Rx	UART Tx	UART Rx	I2S Tx	I2S Rx	ADC
Radio Tx	■	✓	✓	✓	✓	✓	✓	✓	✓
Radio Rx	✓	■	✓	✓	✓	✓	✓	✓	✓
SPIS Tx	✓	✓	■	✓	✓	✓	✓	✓	✓
SPIS Rx	✓	✓	✓	■	✓	✓	✓	✓	✓
UART Tx	✓	✓	✓	✓	■	✓	×	✓	✓
UART Rx	✓	✓	✓	✓	✓	■	✓	×	×
I2S Tx	✓	✓	✓	✓	×	✓	■	✓	✓
I2S Rx	✓	✓	✓	✓	✓	×	✓	■	×
ADC	✓	✓	✓	✓	✓	×	✓	×	■

Table 9-3: Example on using DMA configuration matrix for I2S and UART

It is the same for the I2S_Rx, the UART_Rx and the ADC peripherals that share the same DMA channel. And so on.

This table should help checking if DMA transfer would be possible depending on the peripherals that are needed. If there are some conflicts, then a possibility would be to have one peripheral transferring data using the DMA channel while the other could do the same through an active data read by the CPU (so obviously without the help of the DMA). But depending on the constraints, it might not be always possible.

The DMA module comes with its own API defined in the file `<SDK_ROOT>/libs/dma/dma.h`. However, this API is not intended to be directly used from an end user application. Instead, it is used by the various drivers provided in the SDK.

This means that when the end user initializes a driver from their application, then the DMA will be enabled and configured. So it is not expected that the user directly uses any of the DMA API functions.

The DMA header file is provided only for the end user to be able to build their own final application that uses drivers which in turn are using the DMA controller for transferring data.

9.1.4 Drivers DMA version

Some drivers are provided in two flavours, one without any DMA controlled data transfer and one fully relying on DMA data transfer. This is the case for the following drivers:

- SPI slave
- UART

The selection between DMA or no DMA driver version shall be done in the `CmakeLists.txt` file of the application to be developed when setting up the list of libraries to link with, like in the following example:

```
SET(${PROJECT_NAME}_SRCS)
      nvm_main.c
)

SET(${PROJECT_NAME}_LIBS
      ${NVM_COMMON_LIBS}
      uart_dma
)

APP_IN_NVM()
…
```

In this example, the DMA version of the UART will be used, and the final application will be actually linked with the `lib_uart_dma.a` file.

If there is no need to use the UART DMA version because the foreseen amount of data to be transferred is very low, then the simple UART driver version shall be used, like in the following example:

```
…
SET(${PROJECT_NAME}_LIBS
      ${NVM_COMMON_LIBS}
      uart
)
…
```

Some other drivers use the DMA for data transfer by default without any possibilities to change the behaviour. These drivers are the following:

- ADC for continuous sampling mode only (single sampling does not use DMA)
- I2S
- Radio

By using the above-mentioned drivers, the DMA is automatically activated. This information must be taken into account to meet the DMA matrix usage requirements exposed in Table 9-2.

9.2 ADC

9.2.1 Overview

The ADC is an 8/9 bits resolution analog to digital converter which can be used to measure 3 different input sources. The different sources along with their voltage ranges are listed in Table 9-4.

Source	Expected voltage range
VBAT1	1.8v ➨ 3.6v
VCC	0.95v ➨ 1.9v
GPIO5 with sampling and hold	0.05v ➨ 0.5v 0.05v ➨ 1.0v
GPIO5 without sampling and hold	1.8v ➨3.6v 0.95v ➨ 1.9v

Table 9-4: ADC voltage range depending on source

GPIO5 input source also implements a sample and hold which can be enabled or disabled. By default it is disabled.

Thus, the ADC has the ability to work in two modes:

- single mode where one single data acquisition is done on request
- continuous mode where the ADC continuously samples data until the requested amount of data is received

In continuous mode, the sampling frequency can be configured from 88888 Hz to 120000 Hz without sample and hold, or from 60000 Hz to 85714 Hz with sample and hold.

9.2.2 Initialization

The API of the ADC driver allows to select the input source and configure the different available features. First of all, the ADC shall be fully configured before it is enabled and can be used.

Since the hardware needs 50 µs to be ready, the `ADC_Enable()` API actually implements this 50 µs delay.

For example, in order to use the ADC on VBAT1 source the code can be the following:

```
// Select the source
ADC_SetSourceSelection(ADC_SOURCE_VBAT1);

// Configure the acquisition clock
ADC_SetClockConfig(ADC_CLK_120000_HZ);

// Set ADC sample width
ADC_SetResolution(ADC_9_BITS);

// Finally enable the ADC
ADC_Enable();
```

In order to work properly after wakeup, the ADC driver shall be resumed, which means that the ADC hardware shall be reconfigured by the driver. When the system enters sleep mode, the ADC is not powered anymore and needs a full reconfiguration after wakeup.

To do that, the `ADC_Resume()` function shall be called after wakeup in `NVM_ApplicationEntry()`. Note that a delay of 50 µs is also implemented in this function for the same reason as `ADC_Enable()`.

Here is the code structure that shall be part of the `NVM_ApplicationEntry()` function in case the ADC is used in an application:

```
NO_RETURN void NVM_ApplicationEntry(void)
{
      // Check from which state we are resuming
      If (PML_DidBootFromSleep())
      {
            ADC_Resume();
      }
}
```

9.2.3 *Single mode acquisition*

9.2.3.1. *Overview*

The single acquisition mode allows the software to request one sample from the ADC. This sample is acquired using the 8 or 9 bits resolution depending on how it has been configured.

This single sample acquisition can be done in synchronous mode (or blocking mode) or in asynchronous mode (or non-blocking mode).

In blocking mode, the API starts a conversion and actively waits for the result. This means that the software is blocked until the sampling is completed.

In asynchronous mode, the API starts an acquisition and returns immediately. When the data acquisition is completed, the API calls a user callback function previously set in the ADC configuration sequence to inform the end user that the acquisition is completed.

Then, in this callback, the user can read the sampled value and pass it to another task for processing.

9.2.3.2. *Synchronous mode*

In case of a blocking request, the API returns false if a conversion times out or in case of overrun. An overrun can occur if a new ADC conversion is requested while another one is running. This can happen in a multi-tasking environment, this is why it is not recommended to use the ADC in a multi-tasking environment that could lead to concurrent access.

So, to start a blocking single data acquisition and get the sampled data, the following code snippet can be used:

```
uint16_t sample;

// Start a single synchronous data sampling
if(ADC_StartBlocking() == true)
{
   if(ADC_GetValue(&sample) == true)
   {
      // The 'sample' variable contains the actual sample.
      // Convert the raw value to engineering value
      uint16_t voltage = ADC_ValueToMillivolt(sample) ;
   }
   else
```

```
    {
        // Process the error, or do something else…
    }
}
else
{
        // Process the error
}
```

As it can be seen in this example, the raw value is then converted into engineering values in millivolts. This step is not mandatory depending on the end user's goal. But to get the corresponding engineering value, the input source is taken into account because the input voltage ranges are not the same.

The `ADC_StartBlocking()` function returns false in case one of the following error occurred:

- A timeout occurred, the data sampling duration exceeded 20µs
- A data overrun occurred, meaning that a conversion has been requested while one is on-going

There is no way to make the distinction between these two errors.

9.2.3.3. Asynchronous mode

In case of an asynchronous request, a callback shall be provided to the ADC driver. This callback is executed in an interrupt context and shall have the following prototype.

```
Void ADC_ISR_Callback(Driver_Status_t status, void* pUserData);
```

For of ADC driver implementation, the status is always `DRIVER_STATUS_OK` and user data always `NULL`. Consequently, for the ADC, these two parameters are useless.

The callback shall be registered by calling the following function:

```
ADC_RegisterCallback(ADC_ISR_Callback);
```

At this point, everything is ready.

A sample in asynchronous mode can be acquired by calling the following API.

```
ADC_StartWithCallback();
```

This function starts the conversion and immediately returns.

The software is then free to do some other operation until the conversion is ready. The user callback function is called when the conversion is done and the sample is ready to be used.

As usual in such situations, the user is responsible for writing his own callback. Therefore, it is strongly recommended to keep the time spent in this function as short as possible.

Usually, this callback function is a good opportunity to defer the data processing by waking up a task that will actually process the sampled data. The callback is called in an interrupt context.

The `adc_single_example` test application shows how to use the asynchronous mode for acquiring one single sample.

In this example, the callback's role is to post an event that will be processed later on, in the idle task outside of any interrupt context.

The TASK_idle(…) function is the core of the idle task in which events are processed. The SIG_ADC_ISR is a user signal and is processed there:

```
// The definition of the 'idle' task that will process the events.
Static Qstate TASK_idle(QL_Task_t* const me, Qevt* pEvt)
{
…
   switch(pEvent->super.sig)
   {
      case Q_ENTRY_SIG:
         // First initialization here
      break;
      …
      …
      // The event 'SIG_ADC_ISR' has been received because an
      // asynchronous ADC conversion has been completed.
      Case SIG_ADC_ISR:
         uint16_t raw_value;
         // Read the sample raw value and test if no error occurred.
         If(ADC_GetValue(&raw_value) == true)
         {
            // Conversion from raw to engineering value
            uint16_t value = ADC_ValueToMillivolt(raw_value);
            // The converted value is displayed on the serial port.
            Printf("Sample: %d mv\n", (uint16_t)value);
         }
         else
         {
            // Print an error message in this example, but error should be processed.
            Printf("Error!\n");
         }
      break;
   };
   …
}
```

Before the above code can work, the callback content is written to post the ADC event:

```
#include "qep.h"
#include "qf_port.h"

Q_DEFINE_THIS_MODULE("Task")

#define SIG_ADC_ISR      Q_USER_SIG + 0x2

…

void ADC_ISR_Callback(Driver_status_t status, void* pUserData)
{
   // Increment counter for nested interrupts (QP/C macro)
   QK_ISR_ENTRY();
```

```
    // Create the ADC event with the SIG_ADC_ISR custom value defined above.
    QeventParams* pEvent = (QeventParams*)Q_NEW(QeventParams, SIG_ADC_ISR);
    // Post the event
    QACTIVE_POST(&AO_Task.super, (Qevt*)pEvent, NULL);

    // Decrement counter of nested interrupts (QP/C macro)
    QK_ISR_EXIT();
}
```

9.2.4 Continuous acquisition mode

The continuous acquisition mode allows the software to start continuous conversions in sequence running at the configured sampling frequency of the ADC. In order to offload the CPU, the DMA is used to transfer each sample from the ADC to a buffer in RAM defined and configured by the software. When the requested number of samples are acquired and transferred to the buffer, the DMA triggers an interrupt and the defined callback is called.

Once done, the ADC is stopped. A new sequence of continuous mode sampling is not automatically restarted.

If the ADC is configured to 9 bits resolution, a buffer using 16 bits unsigned values shall be provided. In this case, there is a waste of 7 bits for each word transferred.

Here is an example of a buffer array declaration for both 8 and 9 bits sample data.

```
Uint8_t  buffer[BUFFER_SIZE] __attribute__((aligned(4))); // for 8 bits ADC resolution
uint16_t buffer[BUFFER_SIZE] __attribute__((aligned(4))); // for 9 bits ADC resolution
```

As already said, a callback shall be provided with the same prototype as the one exposed in ADC single mode acquisition (see §9.2.3.3). The role of this callback is also similar. It lets the user be informed that the transferred is completed and do some more actions.

Once everything is in place, the ADC can be started in continuous mode.

```
Void ADC_StartContinuousWithCallback(buffer, BUFFER_SIZE, 7);
```

The first two parameters are related to specifying the memory area where the DMA will transfer the data coming from the ADC, and its size.

The last parameter is the DMA channel to use. This is fully described in §9.1. Even if this parameter can be freely set by the end user at the extent of the maximum number of available channels, it is recommended to stick to the channel number 7 which is the one reserved for the ADC and shared with the SPI Slave peripheral.

Once the continuous data sampling is started, the software can do other things waiting to be interrupted by the callback when the transfer is completed.

The ADC prevents the EM9305 to switch to sleep mode during a continuous data acquisition. This is due to the fact that the ADC is switched off when switching to sleep mode. Otherwise it would be impossible to complete continuous data acquisition. This information shall be taken into account when designing a product which needs ADC continuous acquisition with a high constraint on saving energy.

9.3 GPIO

9.3.1 Overview

The GPIO module provides a convenient way for an application to configure and to use the available GPIOs. Note that the number of GPIOs is not the same between a QFN or a CSP case. The document [1] summarizes this information.

A complete introduction of the EM9305 GPIOs is provided in [1], so refer to this document for further technical description on this topic.

The following subchapters focus on how to handle GPIOs configuration and use from a high-level end user application.

Unlike the other drivers, the GPIO driver is automatically loaded and does not need an explicit registration at the beginning of the `NVM_ConfigModules()` function. This explains why there is no `GPIO_RegisterModules()` function.

9.3.2 Configuring GPIOs from software

The GPIO module is always used so as such there is no need to register it before changing the configuration. The available GPIOs are by default configured to be used as… GPIOs.

Any of such GPIOs can be set to high or low using the following functions:

- `GPIO_SetHigh(gpio number)`
- `GPIO_SetLow(gpio number)`

Thus, a GPIO can simply be toggled by calling the following function:

- `GPIO_SetToggle(gpio number)`

The header file `gpio.h` provided in the SDK lists all the GPIO related functions that can be used in an end user application.

Knowing that, a specific GPIOs configuration must be done in case the default configuration does not match the end user application needs.

This configuration must be done inside the `NVM_ApplicationEntry()` function and relies on the set of GPIO functions defined in the file `gpio.h` provided in the SDK.

The code snippet below shows how to configure the pins `GPIO_PIN_UART_TXD` (pin #7) and `GPIO_PIN_UART_RXD` (pin #6) to use the UART for sending data to/receiving data from the host computer.

```
#include <gpio.h>

…

void NVM_ApplicationEntry(void)
{
   If(!PML_DidBootFromSleep())
   {
      //    ___ ___ ___ ___ ____        _______  __
      //   / __| _ \_ _/ _ \__  |  ___ |_   _\ \/ /
      //  | (_ |  _/| | (_) |/ /  |___|  | |  >  <
      //   \___|_| |___\___//_/          |_| /_/\_\
      //
      // Configure GPIO 7 as UART Tx line (its primary function)
```

```
     GPIO_SetOutputPinFunction(GPIO_PIN_UART_TXD, GPIO_PIN_FUNC_OUT_UART_TXD);
     // Configure a pull-up resistor on Tx line
     GPIO_EnablePullUp(GPIO_PIN_UART_TXD);
     // Disable the pull-down resistor on Tx line
     GPIO_DisablePullDown(GPIO_PIN_UART_TXD);
     // and finally, enable output on TxD line
     GPIO_EnableOutput(GPIO_PIN_UART_TXD);

     //   ___ ___ ___ ___   __         ___ __  __
     //  / __| _ \_ _/ _ \ / /   ___  | _ \\ \/ /
     // | (_ |  _/| | (_) / _ \ |___| |   / >  <
     //  \___|_| |___\___/\___/       |_|_\/_/\_\
     //
     // Set GPIO6 UART RXD line primary function
     GPIO_SetInputFunctionPin(GPIO_PIN_UART_RXD, GPIO_PIN_FINC_IN_UART_RXD);
     // Configure GPIO6 as input (UART Rx line)
     GPIO_EnableInput(GPIO_PIN_UART_RXD);
     // Disable pull down resistor
     GPIO_DisablePullDown(GPIO_PIN_UART_RXD);
     // Disable pull down resistor
     GPIO_DisablePullUp(GPIO_PIN_UART_RXD);
  }
}
```

In this example and in general, the GPIO re-configuration is only needed when not resuming from sleep, i.e. starting from a power ON or from a software reset.

In this example, the GPIO7 is configured as the TX signal for communication to send data over UART. Thus, the Tx line pull-up resistor is enabled.

The GPIO6 is configured to act as the Rx line for receiving data from the host computer.

Note how the function used to set the TxD and RxD lines to primary functions are named. The output pin function configuration is named `GPIO_SetOutputPinFunction()` while the input pin function configuration is named `GPIO_SetInputFunctionPin()`. The terms “Pin” and “Functions” are swapped. This does not really sound logical and will be hopefully improved in the future to have more harmonized naming rules.

9.3.3 GPIO default configuration at startup

By default, after the ROM has completed its start-up sequence and jumped to the main end user application, the GPIOs are in high impedance with no pull-up resistor configuration. They can be kept in this configuration if they are not needed, otherwise it is the main application’s duty to configure them as input/output and with pull-down/pull-up resistor according to the needs.

9.3.4 GPIOs and interruptions

Any GPIO can be used to trigger an interruption. However, the detection of such interruption is not done in the same way depending on the mode in which the system is working, either active (or running) or sleep.

When the system is running, i.e. the CPU is executing code, a leading edge or a falling edge triggers the interruption. The selection between leading or falling edge depends on how this detection is configured.

Unlike the active mode, when the system is sleeping, a level on a GPIO will trigger an interruption, either a high level or a low level depending on how it is configured.

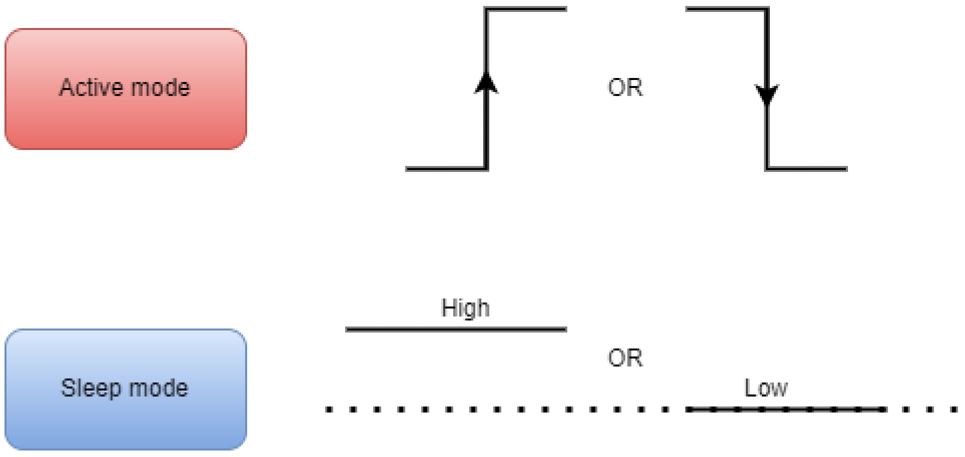


Figure 9-1 : GPIO interruptions vs device mode

One major point that must be considered is that if the goal is to wake-up the device using a GPIO when in sleep mode, then the following steps shall be taken before switching to the sleep mode:

1. configure the active level on the selected GPIO that will trigger the interruption: high or low
2. be sure that the GPIO is not in the active state
3. switch to sleep mode

If, when switching to the sleep mode, the GPIO has the value that has been selected to wake-up the device, then the sleep mode will not become active and the device will stay in active mode.

9.4 I2C MASTER

The I2C driver provides an abstracted way to control the communication over the I2C bus with connected devices. Using this driver, the EM9305 acts as an I2C master device.

In order to use this driver in an end user application, the dedicated module shall be listed in the list of libraries to link with as shown in the following example taken from the `CmakeLists.txt` file.

```
SET(${PROJECT_NAME}_LIBS
   ${NVM_COMMON_LIBS}
   i2c
}
```

To get access to I2C driver API, the following header files shall then be added in the end user application source file.

```
#include “i2c.h”
#include “i2c_hal.h”
```

Refer to the I2C Master documentation provided in the SDK to get all required technical details on how to use it in an end user application.

9.5 PML

The PML driver provides an interface to the power management logic IP from a software application.

Through the provided set of functions, an application can do the following:

- configure the CPU core frequency
- set the clock source
- configure the clock calibration
- manage the power supply for RAM blocks for data retention
- configure and manage the sleep modes

The PML configuration structure and content is given in Table 9-5 for informational purpose.

Field	Type	Default value	Description
lfClkFreqRatioHf	uint32	589030	LF XTAL frequency expressed as a ratio HF/LF clock – Numerator
lfClkFreqRatioLf	uint32	1023	LF XTAL frequency expressed as a ratio HF/LF clock – Denominator
lfClkSourceAccuracy	uint16	500	LF clock source accuracy as ppm
lfClkSourceType	uint8	Internal LF RC clock	Low frequency clock source selection: • PML_LF_CLK_XTAL_DIS (use internal LF RC clock, external LF XTAL disabled – default -) • PML_LF_CLK_XTAL_EN (use external LF XTAL) • PML_LF_CLK_XTAL_SQ (use external square wave) • PML_LF_CLK_XTAL_SINE (use external sine wave)
rcCalibSkip	Boolean	False	Skip the RC calibration. Use constant values instead.
rcCalibHfLimit	uint32	0	The RC high frequency limit for the calibration.
rcCalibLfLimit	uint32	48	The RC low frequency limit for the calibration.
rcCalibPeriod	uint32	20600	The period at which the RC shall be calibrated. The default value at a frequency of 500kHz is ~500 ms
rcCalibCorrection	int16	0	Static correction applied to the RC calibration results in ppm. A positive value extends the sleep interval.
hfClkFreq	uint16	24	HF clock frequency in MHz (at startup, the core runs at 24 MHz). The switch to 48 MHz is done prior to executing the end user application.
overheadBootTime	uint32	0	The estimated boot time overhead duration in µs
overheadSleepTime	uint32	300	The estimated overhead sleep duration in µs.
minimalSleepTime	uint32	15000	The minimal sleep time duration in µs below which the sleep manager will not actually configure the system to go to sleep.

sleepModeForbiden	uint8	False	By setting this Boolean variable to true, the sleep manager is instructed not to switch to sleep mode.
bypassSwitchEnable	Boolean	False	This Boolean can be used to enforce bypassing the switch enable in sleep mode.
useLdoVccDuringCalib	Boolean	False	This Boolean flag can be used to enforce using the LDO VCC during the RC calibration process.

Table 9-5: PML global configuration structure

Note that disabling sleep mode has a direct impact on the power consumption.

Integrating the PML driver in an end user application can be simply done by adding the driver in the list of drivers to link with in the `CMakeLists.txt` file like in the following code snippet:

```
SET( ${PROJECT_NAME}_LIBS
   ${NVM_COMMON_LIBS}
   pml
}
```

Then, in the C source file(s) of the end user application, the PML header file shall be first included prior to using any function the driver provides.

```
// Get access to the PML driver API
#include "pml.h"
```

Refer to the PML SDK documentation for a complete description of the PML driver and what it can achieve.

Including the file `pml.h` file in an application source file gives access to the `gPML_Config` structure which contains all the fields defined in Table 9-5. There is no need to explicitly declare an external reference to this structure. The structure pointer can be directly used like in the following example:

```
gPML_Config.sleepModeForbiden = true;
```

9.6 QDEC

The QDEC driver provides an abstraction layer to use the quadrature decoder.

To be able to use this driver, the application `CmakeLists.txt` file shall have this driver listed in the list of libraries.

```
SET(${PROJECT_NAME}_LIBS
   ${NVM_COMMON_LIBS}
   qdec
}
```

Then, in the main C source file of the application, the following headers can be included prior to using SPI functions like in the following code snippet:

```
#include "qdec.h"
```

Refer to the QDEC documentation provided in the SDK to get all required technical details on how to use it in an end user application.

9.7 RTC

The RTC (Real Time Clock) is a functionality provided within the SDK and relies on the sleep timer that helps maintaining the system time over the product lifetime. Once configured, the RTC maintains a timer that continuously counts from the reference time set to the 1st of January 2000. At any moment, this counter can be converted into another time representation like a date (day, month and year) and a time (hour, minutes, seconds).

To use this functionality, the end user application `CmakeLists.txt` file shall list the RTC library in the definition of libraries to use as shown in the following code snippet:

```
PROJECT(my_rtc_app C)
SET(${PROJECT_NAME}_SRCS
      nvm_main.c
)
SET(${PROJECT_NAME}_LIBS
      ${NVM_COMMON_LIBS}
      rtc
}
```

At this point, the end user is now ready to use the RTC.

Refer to the RTC documentation provided in the SDK to get all required technical details on how to use it in an end user application.

9.8 SPI Master

The SPI master driver provides an abstraction layer to drive a SPI communication to SPI slave devices connected to the EM9305.

To be able to use this driver, the application `CmakeLists.txt` file shall have this driver listed in the list of libraries.

```
SET(${PROJECT_NAME}_LIBS
   ${NVM_COMMON_LIBS}
   spi_master
}
```

Then, in the main C source file of the application, the following headers can be included prior to using SPI functions like in the following code snippet:

```
#include "spi_master.h"
```

Refer to the SPI Master documentation provided in the SDK to get all required technical details on how to use it in an end user application.

9.9 SPI SLAVE

The SPI slave driver provides an abstraction layer for an end user application running on the EM9305 to communicate with a SPI master device.

This driver comes in two flavours:

- SPI slave driver without DMA (`lib_spi_slave.a`)
- SPI slave driver with DMA (`lib_spi_slave_dma.a`)

Refer to §9.1.3 to check if the DMA version can be used along with other drivers DMA versions since the SPI slave driver shares the DMA channel 4 with the UART.

Selecting the right driver flavour is done in the `CmakeLists.txt` file of the end user application when specifying the list of libraries to link with, like shown in the following example.

For using the version without DMA:

```
SET(${PROJECT_NAME}_LIBS
   ${NVM_COMMON_LIBS}
   spi_slave
   …
}
```

For using the DMA version:

```
SET(${PROJECT_NAME}_LIBS
   ${NVM_COMMON_LIBS}
   spi_slave_dma
   …
}
```

Then, in the C source file of the end user application, the following header file shall be included:

```
#include "spi_slave.h"
```

The choice between these two versions depends on the amount of data to be transferred. Better performances can be achieved with the DMA version when a big amount of data is to be transferred since the overhead due to DMA configuration prior to the transaction is low. On the other hand, for transferring a small amount of data (like a few bytes), the non-DMA version would be preferred since the execution of code for DMA configuration prior to the transaction might take as much time as requested for the transaction itself. Moreover, it should be noted that once the SPI slave library version has been selected, it is part of the application and cannot be changed at run time. Thus, these two versions cannot be embedded in the same application. Consequently, the right version of this library to use shall be selected with care.

Refer to the SPI Slave documentation provided in the SDK to get all required technical details on how to use it in an end user application.

9.10 UART

The UART driver is used when a serial communication is needed. Depending on the requested bandwidth, the UART comes in two flavours:

- UART driver without DMA
- UART driver with DMA

The right UART driver version shall be added in the libraries list to be linked with in the `CmakeLists.txt` file of the application, like in the following code snippet for integration of the non DMA UART driver:

```
SET( ${PROJECT_NAME}_LIBS
   ${NVM_COMMON_LIBS}
   uart
}
```

Otherwise, integration of the DMA flavour is done like shown below:

```
SET( ${PROJECT_NAME}_LIBS
   ${NVM_COMMON_LIBS}
   uart_dma
}
```

Both the UART with DMA and w/o DMA have the same API so they can be interchanged easily.

Refer to the UART SDK documentation for a complete description on how to configure and use the UART driver API.

9.11 WATCHDOG

Unlike the other drivers that come along with a library and several header files, the watchdog driver only contains a header file that defines inline functions that can be directly called from the end user application.

These functions are simply helper function to control the hardware watchdog part of the EM9305 device.

Thus, there is no need to list the watchdog driver in the list of libraries to be linked with the end user application in the application `CmakeLists.txt` file.

Using the watchdog is just a matter of including the right header in the application source file:

```
// The watchdog header file contains the very simple watchdog API.
#include “watchdog.h”
```

And that’s it!

The `nvm_watchdog_example` is a very simple application that showcases how to use the EM9305 watchdog.

Refer to the Watchdog SDK documentation for the details of the watchdog API.

10. INTERRUPT AND EXCEPTION MANAGEMENT

10.1 INTERRUPTIONS MANAGEMENT

10.1.1 Overview

When the device starts-up, the ROM software configures the interrupt priority management according to the scheme defined in Table 10-1. This table shows the interruptions list from the highest priority at the top (priority equals to 0) to the lowest priority at the bottom (priority equals to 9).

Priority (0 ➡ highest, 9 ➡ lowest)	IRQ #	Interrupt source
0	20	Radio Tx
	21	Radio Rx
1	5	Protocol timer out comparator 0
	6	Protocol timer out comparator 1
	7	Protocol timer out comparator 2
	8	Protocol timer out comparator 3
	9	Protocol timer out comparator 4
	10	Protocol timer out comparator 5
	11	Protocol timer out comparator 6
	12	Protocol timer out comparator 7
	13	Protocol timer full value
	14	Protocol timer synchronization
2	15	Sleep timer out comparator 0
	16	Sleep timer out comparator 1
	17	Sleep timer out comparator 2
	18	Sleep timer out comparator 3
	19	Sleep timer full value
3	31	RC calibration
	34	PML clock
4	22	SPI slave Tx
	23	SPI slave Rx
	24	UART Tx
	25	UART Rx
5	0	ARC timer 0
	1	ARC timer 1
	2	ARC watchdog
	3	ARC DMA done
	4	ARC DMA error

	33	PML SVLD
6	26	GPIO
	32	ADC
	38	Crypto unit
	39	I2S
7	27	Universal timer 2
	28	Universal timer 3
	29	SPI master
	30	I2C master
	35	NVM
	36	QDEC
	37	USB
8	41	SWI1
	42	SWI2
	43	SWI3
	44	SWI4
	45	SWI5
	46	SWI6
	47	SWI7
	48	SWI8
	49	SWI9
9	40	SWI0

Table 10-1: Interrupt priorities configuration at start-up

The interrupts are natively managed by the drivers through their own defined interrupt handlers. However, an end user application code can register user defined callback functions for some of these interruptions to achieve specific processing at user level. Once defined, such user callback functions are called whenever an ISR occurs.

The internal ISR handler code handles and manages the ISR, then clears it and finally calls the callback. Consequently, there is no need for the end user application to clear or acknowledge an ISR in the user callback.

10.1.2 Connecting a user callback

There are several functions to set a callback for the following ISR:

- SWI (Software Interruptions)
- GPIO
- Sleep Timer
- Protocol Timer
- ARC Timers

They are documented in the SDK's documentation in "Interrupt management" part.

Anyhow, the GPIO interrupt management is taken here as an example to illustrate how things work.

If the user application needs to be informed when a GPIO interrupt is triggered, a callback shall be registered in the end user `NVM_ApplicationEntry()` using the following function:

```
void IRQ_SetIRQUserHandlerGPIO(void* pIRQUserHandlerGPIO);
```

The prototype of the GPIO user callback shall be compliant with the following:

```
void IRQUserHandler_GPIO(uint32_t gpio);
```

Here is a code snippet to show how to use it.

```
#include “irq_user_isr.h”

void IRQUserHandler_GPIO(uint32_t gpio)
{
  // User handler for GPIO ISR
}

void RegisterGPIO_ISR_Callback(void)
{
  IRQ_SetIRQUserHandlerGPIO(IRQUserHandler_GPIO);
}
```

The user defined function is a callback function. It will be called by the actual GPIO interrupt handler which is declared with the `_Interrupt` keyword. Consequently, there is no need to use such keyword for defining the user callback.

For information, here is how the internal interrupt handler is declared:

```
void _Interrupt IRQHandler_GPIO(void)
{
      // Read the GPIO interrupt status register
      uint32_t status = IRQ_GetStatus(IRQ_GROUP_GPIO) ;
      // Identify which GPIO has triggered the interruption
      uint8_t gpio = 0;
      while(!(status & (1u << gpio)) gpio++;

      // Acknowledge the GPIO interrupt
      IRQ_Clear(IRQ_GROUP_GPIO, IRQ_GPIO_EN_CLR(1u << gpio)) ;
      BOOT_IrqActionRoutine((uint32_t)SET_BOOT_ACTION_FLAGS_GPIO(1u << gpio)) ;

      // If a user defined callback has been previously registered, then call it!
      if(gISR_pIRQUserHandler_GPIO != NULL)
      {
            // Call user defined callback
            gISR_pIRQUserHandler_GPIO(gpio) ;
      }
}
```

The `_Interrupt` keyword in this code snippets obviously designates the function handler to be an interrupt handler. It instructs the compiler to generate code to save the interrupt register state before entering the handler and to output the `rtie` instruction to exit this handler unlike normal functions that would use the `ret` instruction. This macro expands to `__attribute__((interrupt))`.

This handler reads the GPIO interrupt status register and loops through the 11 interruption sources to get the first GPIO that has triggered the interruption.

Once identified, the interruption is acknowledged, and the user defined callback function is called (if it has previously been registered).

It is important to mention that the user callback is called in interruption context and shall be as short as possible. This means that the following rules shall be followed:

- no active wait
- no recursivity
- be careful when dealing with global variable at interrupt execution level to avoid any concurrent access

As said earlier, it is strongly recommended to be as fast as possible in the user define interrupt callback, to avoid introducing too much latency that could break the real time schedule. As a matter of fact, it is recommended not to extend the callback function duration execution beyond **2.5 milliseconds**.

If the application uses QP/C, then in case of a GPIO interruption, an event shall be posted to the task that will process the interrupt through a deferred execution out of the scope of the interrupt context. Here is an example of what such an end user callback would look like in a QP/C application:

```
static void IRQUserGPIO_ISR_Callback(uint32_t gpio)
{
      // Increment the nested calls counter
      QK_ISR_ENTRY();

      // Post a user event to the task. It shall have been defined earlier.
      QeventParams* pEvent = (QeventParams*)Q_NEW(QeventParams, SIG_GPIO_ISR);
      QACTIVE_POST(&AO_Task.super, (Qevt*)pEvent, NULL);

      // Decrement the nested calls counter and check if it is equal to 0.
      // If this counter reaches 0, then reactivate the kernel scheduler, otherwise
      // we are still in interrupt context.
      QK_ISR_EXIT();
}
```

The `QK_ISR_ENTRY` and `QK_ISR_EXIT` macros are used to count the number of nested interruptions. The `QK_ISR_ENTRY` macro simply increments this counter while the `QK_ISR_EXIT` macro decrements it.

Then, the `QK_ISR_EXIT` macro decrements this counter and checks if it is equal to 0 meaning that there is no more nested call.

In case this counter is greater than 0, this means that the process is still in interrupt context.

Using these macros ensures the proper operation of the QP/C kernel scheduler when leaving the interrupt context. When the counter reaches 0, then the kernel scheduler is reactivated, and the application tasks can be scheduled again.

The code in between creates an event variable on the memory stack and posts this event to the active object task. If this task is the highest priority one that is pending, it will then process the event and finalize the actions to be done.

More detailed information can be found in §16.

The functions listed in Table 10-2 can be used by an end user application to setup callback functions each time a related interruption is triggered.

Function	Description
`IRQ_SetIRQUserHandlerSW1()`	Set one single callback function for all eight software interruptions that can be triggered. The callback parameter contains the source of the interrupt ranging from 2…7. Note that interrupts 0 and 1 are reserved for QP/C's own usage. The software interrupts have the lowest priority.
`IRQ_SetIRQUserHandlerGPIO()`	Set a callback that is executed just after a GPIO signal configured as an interrupt signal is asserted.
`IRQ_SetIRQUserHandlerSleepTimerOutCmp()`	Triggered when the system resumes from a sleep period.
`IRQ_SetIRQUserHandlerSleepTimerFullValue()`	Triggered when the sleep timer counter reaches its maximal value, whatever is the actual timer configured threshold (2 hours 22 minutes 48 seconds with the LF RC clock running at 500 kHz).
`IRQ_SetIRQUserHandlerProtoTimerOutCmp()`	Triggered when the protocol timer reaches its threshold value.
`IRQ_SetIRQUserHandlerProtoTimerSync()`	Triggered when the protocol timer starts counting and when it stops. It can be used to get synchronized with this timer.
`IRQ_SetIRQUserHandlerARCTimer0()`	Triggered when the ARC timer 0 has elapsed.
`IRQ_SetIRQUserHandlerARCTimer1()`	Triggered when the ARC timer 1 has elapsed.

Table 10-2: API to user defined IRQ handlers

10.1.3 List of interruptions handlers

The list of interruptions handlers that are declared as weak functions and that can be overridden by user defined handler are given in Table 10-3. Note that this table does not cover the exceptions that are discussed in §10.2.

Interruptions	Description
IRQHandler_ArcWatchdog	In case the watchdog counter reaches 0, an interruption is triggered.
IRQHandler_ArcDmaDone	End of a DMA operation.
IRQHandler_ArcDmaError	DMA operation ended with an error.

IRQHandler_ProtocolTimerOutCmp0	Protocol timer reached limits that were previously defined.
IRQHandler_ProtocolTimerOutCmp1	
IRQHandler_ProtocolTimerOutCmp3	
IRQHandler_ProtocolTimerFullVal	Protocol timer full value reached.
IRQHandler_RadioTx	The radio IP triggers an interruption at the end of a transmission operation.
IRQHandler_RadioRx	The radio IP triggers an interruption when the Rx windows is closed.
IRQHandler_SpiSlaveTx	The SPI slave block triggers an interruption on one of the following conditions: • byte sent. • Tx FIFO empty: bytes have been sent but the FIFO runs out of data to send. • Tx FIFO limit reached: the number of remaining bytes in the FIFO is below the user defined limit. • Tx FIFO underflow: the FIFO is empty at the time the SPI transaction has started.
IRQHandler_SpiSlaveRx	The SPI slave block triggers an interruption on one of the following conditions: • byte received. • Rx FIFO full: the whole FIFO has been filled by received bytes. • Rx FIFO limit reached: the user defined number of bytes has been reached in the Rx FIFO. • Rx FIFO overflow: a new byte is received while the Rx FIFO is full of unprocessed data. • Protocol error: first byte LSB and MSB are swapped on the received byte.
IRQHandler_UartTx	The UART Tx block triggers an interruption on one of the following conditions: • byte sent. • Tx FIFO empty • Tx FIFO limit reached: the number of remaining bytes to send has just gone below the user defined limit.
IRQHandler_UartRx	The UART Rx block triggers an interruption on one of the following conditions: • byte received. • Rx FIFO full: the Rx FIFO is full of received bytes. • Rx FIFO limit reached: the number of received bytes stored in the FIFO has just gone above the user defined limit. • Rx FIFO overflow: the FIFO is full but more bytes are arriving. • Wrong stop bit detected • Parity error detected

IRQHandler_UniversalTimer2	The universal timer throws an interrupt on the following triggers: • the timer count value has reached its limit value • the timer count value has reached the comparator values (0…3)
IRQHandler_UniversalTimer3	Same as above
IRQHandler_SpiMaster	The SPI master block triggers an interruption at the end of a SPI transaction.
IRQHandler_I2cMaster	The I^{2}C block triggers an interruption at the end of an I^{2}C transaction.
IRQHandler_RcCalib	The RC calibration interrupt is generated when an RC calibration operation is completed. Then, a user defined callback (if any) is called. By this way, the end user application can be synchronized with this operation if some specific processing needs to be done.
IRQHandler_ADC	The ADC block triggers an interruption at the end of one single ADC conversion.
IRQHandler_PmlSvld	The supply voltage level detector (SVLD) has detected failure conditions and triggers an interruption. One of its main goals is to warn the end user application when a brown-out condition is detected regarding the reference voltage of 1.65 volts. It is also used to monitor the battery voltage or an external DCDC power supply.
IRQHandler_Qdec	The QDEC interrupt handler is in charge of updating the following information: • a wheel movement has been detected • an accumulator overflow has been detected • a double movement error has been detected Then, it calls the user defined callback (if any) passing the above mentioned information for user defined processing.
IRQHandler_USB	USB endpoint interruption generated.
IRQHandler_CryptoUnit	The crypto container block triggers an interruption an AES operation is completed.
IRQHandler_I2s	The I2S block throws interruptions in the following cases: • RX FIFO overflow • RX FIFO underflow • RX FIFO limit has been reached (prior to having a FIFO full) • RX new frame • TX FIFO overflow • TX FIFO underflow • TX FIFO limit has been reached • TX FIFO empty • TX new frame
IRQHandler_SWI0	Software interruptions handlers.

IRQHandler_SWI1	
IRQHandler_SWI8	
IRQHandler_SWI9	
IRQHandler_Timer0	Handler of the timer 0 interruption (#0)
IRQHandler_Timer1	Handler of the timer 1 interruption (#1)

Table 10-3: List of supported interruptions

Each of the above-mentioned handler has the following prototype:

```
void _WeakInterrupt(“NullHandler”) IRQHandler_xxx(void);
```

The “xxx” term is to be replaced by the actual function, like `UartTx` for example.

Redefining a strong symbol with the exact same name will override the default interruption handler. In such a case when the end user defines his/her own interruption handler, he/she is responsible to write the relevant code to acknowledge the interruption.

This can be achieved by clearing the interrupt bit in the relevant interruption group register.

As an example, to acknowledge a specific GPIO interruption, the following line shall be part of the interrupt handler:

```
IRQ_Clear(IRQ_GROUP_GPIO, IRQ_GPIO_EN_CLR(1 << gpioID));
```

The Figure 10-1 shows the bits to use to clear the interrupts (acknowledgement).

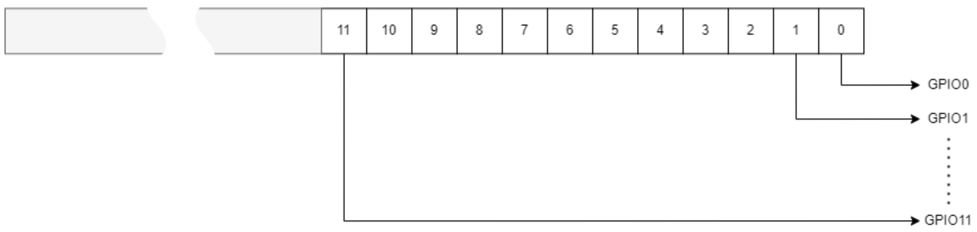


Figure 10-1: GPIO interrupt acknowledgement register

This function will write ‘1’ to the bit identified by the value `gpioID`. This action will clear the status bit for this GPIO interruption. Thus, only writing a value of ‘1’ will have effect on the register. Writing ‘0’ will have no effect.

The first parameter of this function specifies the interrupt group to be used. There are 18 groups which are the following:

1. protocol timer
2. sleep timer
3. RF
4. SPI slave
5. UART
6. GPIO
7. universal timer
8. SPI master

9. I2C master
10. RC calibration
11. ADC
12. PML
13. NVM
14. QDEC
15. USB
16. Crypto unit
17. I2S
18. SWI

The sequence order of these groups is important to be able to get an access to them as we'll see below.

Each group has 9×32 bits register which are all mapped sequentially with no gap in between. All these registers are the same for each feature (so they are repeated 18th times, one set of registers per feature). Here is the list of 9 registers for the first feature which is the protocol timer:

The Table 10-4 list the content of the interruption management group for the protocol timer functionality.

IRQ register	Description
IRQProtTimEn	Controls interruption enable and disable
IRQProtTimEnSet	Controls interruption enable only (only bits set to 1 have effects)
IRQProtTimEnClr	Controls interruption disable only (when disabled, an incoming event that would trigger the interruption is never propagated to the CPU. It is lost).
IRQProtTimMsk	Controls interruption masking (when masked, an incoming event that would trigger the interruption is stored and propagated to the CPU only when it is unmasked)
IRQProtTimMskSet	Controls setting the interruption mask only (only bits set to 1 have effects)
IRQProtTimMskClr	Controls unmasking the interruption (only bits set to 1 have effects)
IRQProtTimSts	Register giving the status of the interruptions, either they are enabled or disabled
IRQProtTimStsSet	Controls setting the flags of the status register (only bits set to 1 have effects)
IRQProtTimStsClr	Controls clearing the flags of the status register (only bits set to 1 have effects)

Table 10-4: Set of interrupt management registers for the protocol timer functionality

After the last register in this list (`IRQProtTimStsClr`) comes the IRQ enable register for the second feature which is the sleep timer. Then, the 9 registers for the sleep timer are exposed sequentially in the same order. And so on for all the 9 registers of each of the remaining features (RF to SWI).

The Figure 10-2 depicts this organization showing the first two groups of interruption registers (protocol timer and sleep timer). Other groups (RF…SWI) are not represented here. As shown in this picture, the relative offset address of a register compared to the previous one is incremented by 4.

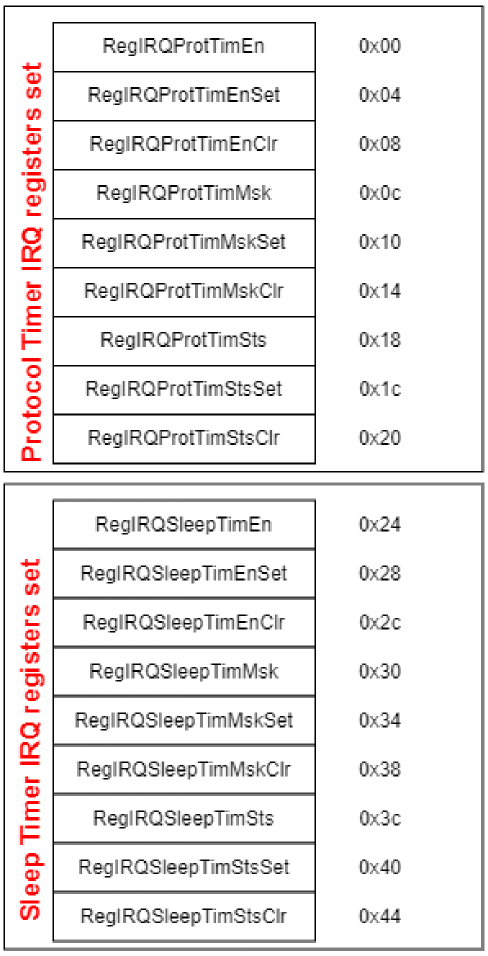


Figure 10-2: Sequential organization of the interrupt registers

Consequently, accessing a specific register within a group can be done by starting from the targeted register (*En*, *EnSet*, *EnClr*, *Msk*, *MskSet*, …) in the first group (protocol timer) and by adding an offset that is computed with the following formulae:

$$address = IRQProtTim < register > \; + group * (9 * 4)$$

The term `register` here shall be one of the following:

• `En`	(enable ➨ bits set to 1 will enable, bits set to 0 will clear)
• `EnSet`	(enable set ➨ only bits set to 1 will enable, bits set to 0 will have no effect)
• `EnClr`	(enable clear ➨ only bits set to 1 will clear, bits set to 0 will have no effect)
• `Msk`	(mask ➨ bits set to 1 will unmask, bits set to 0 will mask)
• `MskSet`	(mask set ➨ only bits set to 1 will mask, bits set to 0 will have no effect)
• `MskClr`	(mask clear ➨ only bits set to 1 will unmask, bits set to 0 will have no effect)
• `Sts`	(status)
• `StsSet`	(status set ➨ only bits set to 1 will have a set effect)
• `StsClr`	(status clear ➨ only bits set to 1 will have a clear effect)

For example, if the `IRQGPIOStsClr` register needs to be accessed, then an access pointer is computed like this:

$$IRQGPIOStsClr = IRQProtTimStsClr + 5 * (9 * 4)$$

The GPIO set of registers is located at group position 5 in the registers mapping. We then multiply 5 by 9 (because of 9 registers per feature) time 4 (because of 32 bits = 4 bytes). Finally, we add the result to the address of the same register in the first group (`IRQProtTimStsClr`). And here we go, we get the address of the `IRQGPIOStsClr` register.

The second parameter of the `IRQ_Clear()` function acts as a bit mask that is used to modify the content of the register. Depending on the targeted register, bits 1 and 0 can have both an impact (set and reset) while on some other registers (which is the case here for this function), only bits set to 1 will modify the register (bits equals to 0 have no effects).

The `IRQ_Clear()` function encapsulates the use of the relevant macros to issue the following statement at the end:

```
(IRQ->RegIRQProtTimerStsClr + IRQ_GROUP_GPIO * 0x24)->r32 = bitmask;
```

In this example, the `StsClr` register is written. Only bits equal to 1 will actually have an effect. Other bits set to 0 will have no effect.

If the source of the GPIO interrupt comes from `GPIO0`, then writing 1 will clear and acknowledge this interruption.

10.1.4 Critical sections and interrupts

The SDK provides a set of drivers that might execute critical sections which must not be interrupted during their execution, mainly to fulfill timing constraints and to avoid concurrent access to a single resource. This is why interruptions are masked at the beginning of such sections and unmasked when completed.

However, for the sake of keeping any on-going Bluetooth LE connection and limiting the timing impacts on the radio IP, the radio driver has the highest priority level compared to other drivers. Consequently, a critical section in a driver will mask all interruptions except the ones that are of the highest priority level, that is the radio driver interruptions.

EM highly recommends not to change the radio driver priority level and not to push another driver than the radio driver at this higher priority level as well. In such a case, EM does not guarantee a nominal behavior of the device.

The Table 10-1 shows that the radio interruptions have interruption priority level 0 which is the highest. To ensure a normal behavior of the device during Bluetooth LE operations, this configuration must not be changed.

When a critical section shall be executed without being interrupted, then it shall be wrapped between statements that will mask and then unmask the undesirable interruptions.

Interruptions can be masked and in such a case, they are recorded but not propagated to the CPU until they are unmasked. In this case, their processing is deferred.

They can also be disabled. In this case, the source of the interruption can trigger one interruption, but from the interruption controller standpoint, the interruption will be lost and will never reach the CPU. Consequently, it will never be processed.

Moreover, there is no way to mask/disable all interruptions with one single instruction, and it is not a good practice to act like this. Instead, a good software architecture design would define accurately the interruptions that would be masked for the critical section and that would prevent this critical section to work properly in case they occur. And then, only these identified interruptions should be masked/disabled.

Here is an example of a critical section in which the interruptions triggered by GPIO0 and GPIO1 are masked.

```
IRQ_Mask(IRQ_GROUP_GPIO, 3);
// Critical section starts here
// …
//    Your code
// …
// Critical section ends here
IRQ_Unmask(IRQ_GROUP_GPIO, 3);
```

The Figure 10-1 shows how the GPIO mask shall be built. The bit 0 identifies the GPIO0, the bit 1 identifies the GPIO1 and so on until the bit 11 that identifies the GPIO11. All these bits can be OR'ed together to build the final mask that is passed to the function `IRQ_Mask()` for the GPIO group. In this example, the parameter 3 indicates that the interruptions related to GPIO0 and GPIO1 are masked.

It shall be noted here that if a previously masked GPIO interrupt is triggered during the execution of the critical section, its processing will be deferred after the GPIO interruptions are unmasked. Moreover, in the critical section, it is possible to know if a masked interruption has occurred by calling the function `IRQ_GetStatus(…)`. This lets the user behave differently if the interruption has been triggered or not. But it does not remove the interruption processing that will be done when the interruption is unmasked.

If the critical section completely gets rid of the masked interruptions, then another possibility is to disable such interruptions, and enable them again after the critical section, as shown in the following code snippet.

```
IRQ_Disable(IRQ_GROUP_GPIO, 3);
// Critical section starts here
// …
//    Your code
// …
// Critical section ends here
IRQ_Enable(IRQ_GROUP_GPIO, 3);
```

When doing so, it is not possible to be informed of the presence of the interruption by using the function `IRQ_GetStatus(…)`.

10.2 EXCEPTIONS MANAGEMENT

10.2.1 Overview

An exception is similar to an interruption in the fact that it redirects the CPU to a specific handler in case an error occurs. However, unlike the interruption, the exception is representative of an error that might prevent the software to keep running in a nominal way. Moreover, an exception is always synchronous to the execution of an instruction while an interruption can be triggered at any time, i.e. asynchronous.

All exceptions that can be processed have a default handler that is called whenever the related exception occurs.

These exception handlers are declared as weak interrupt handlers and then can be overridden by user defined handlers with the same prototype. User defined handlers will then be considered as strong handlers and take precedence over the default handlers.

The default handler that is used to populate the exception vector table simply resets the CPU. Its code is the following:

```
void _Exception NullHandler(void)
{
     // Reset on an unhandled interrupt or exception.
     ResetCPU();
}
```

As mentioned in this code snippet, the `_Exception` keyword specifies that the function is an exception handler. It instructs the compiler to save the interrupted register state before entering the handler and to output the `rtie` instruction in the binary code to return. This keyword expands to `__attribute__((exception))`.

All other exception handlers are function names aliased to the above-described default exception handler. However, by defining a specific strong handler symbol on user side, it is possible to override the default handler.

As an example, let's consider the trap exception that occurs whenever the `trap_s` instruction is executed.

This handler is declared as follow:

```
void _WeakInterrupt("NullHandler") EV_TrapHandler(void);
```

It is an alias to the default so called `NullHandler()` handler.

Like the interrupt handlers, an exception handler is executed at interrupt level. Most of the interruptions are masked before calling the handler so only very fast and simple operations shall be done within the handler.

10.2.2 User defined exception handlers

10.2.2.1. Trap handler

In order to overwrite the default handler by a custom one, a new handler shall be defined at end user application level as shown in the following example:

```
// User defined trap handler somewhere in the end user application
void _Exception EV_TrapHandler(void)
{
     // Your code here that will be executed whenever the trap_s
     // instruction is executed.
}
```

Once the end user application is built, the address of this handler in the resulting symbol file will be the user defined one.

10.2.2.2. Integer division by zero handler

Another example is the integer division by zero which leads to an exception management when it happens.

As usual now, a user defined exception handler shall be defined like this:

```
void _Exception EV_DivZeroHandler(void)
```

```
{
      // Log information on exception that just occurred by toggling a dedicated GPIO twice.
      GPIO_Toggle(GPIOx);
      GPIO_Toggle(GPIOx);

      // Then restart the CPU.
      ResetCPU();
}
```

It will override the default one.

To trigger this exception, the following code snippet can be used:

```
int result_i;

void IntegerDivByZeroTest(void)
{
      for(int i = 3; i >= 0; i++)
      {
            result_i = (i + 200) / i;
            printf("Result = %d\r\n", result_i);
      }
}
```

In this loop, the exception is triggered at the end of the loop when `i` equals to 0. The CPU then branches to the exception handler `EV_DivZeroHandler()` in which it is possible to log post-mortem information. This can be activating a signal (GPIO) and/or storing a value into a retention RAM for example.

The exception handler is executed at interrupt level. Thus, logging post-mortem information shall be straightforward, like storing information in retention RAM that is read back after CPU reset. This can be a good way to troubleshoot this type of failure.

10.2.2.3. Floating point division by zero handler

It is also possible to catch a floating-point division by zero exception by overriding the `EV_ExtensionHandler`. This exception is not the same as the integer division by zero, because the floating-point unit is an extension (CRC, AES, JLI are other example of extensions). Depending on the CPU implementation, it might not be present. However, if it is there (which is the case for the EM9305), floating-point division by zero exceptions will come from this extension. This explains why it is a different handler than the one to process the integer division by zero exception.

The handler prototype that must be defined at user application level shall be the following:

```
void _Exception EV_ExtensionHandler(void)
{
      // Log floating point division by zero exception
}
```

An example of code that is able to trigger such exception is given in the following code snippet. This function can be called from the `NVM_ApplicationEntry()` function.

```
01:  float FloatDivByZeroTest(void)
02:  {
03:        float result_f;
04:
05:        for(int i = 3; i >= 0; i++)
06:        {
07:              if(i == 1)
08:              {
09:                    // Trigger the floating point division by zero exception when i = 1
10:                    result_f = ((float)i + 200.0f) / (float)((int)i - 1);
11:              }
12:              else
13:              {
14:              // When i = 0, its conversion to a floating point value does not strictly
15:              // equals 0. Consequently, the division by i will lead to value so big
16:              // that it is considered as infinity. But no exception will be raised.
17:              result_f = ((float)i + 200.0f) / (float)i;
18:              }
19:        }
20:  }
```

This code is a countdown loop, and the statement that triggers the exception is on line 10. It occurs when the variable `i` equals to 1. Casting the `i - 1` to int will provide a strict null floating-point value. Dividing by this value will actually lead to trigger a div by zero exception.

At first glance, it could have been much simpler to directly apply the statement on line 17 waiting for `i` to reach the null value to trigger the exception. It turns out that doing so will not trigger an exception. Instead, the division will be done but the result will be considered as infinity because casting `i` to a floating does not provide a strict floating point null value. It can be something like 0.000000001 for example, a very small value. Consequently, dividing by this value will give a very big result. Even if this result is too big to be stored within a 32 bits words, it is still valid so no exception is triggered in this example.

10.2.3 List of supported exceptions

The list of supported exceptions is given in Table 10-5.

Exception	Description
MemoryErrorHandler **(0x01)**	Exception triggered in case an error is raised when the CPU access the memory: • Bus error from instruction and data memory • Instruction/data fetch spanning multiple instruction/data memory targets • Memory address out of range • Illegal target memory • …
InstructionErrorHandler **(0x02)**	Exception triggered in case an unknown instruction code/sequence is executed.
EV_ProtVHandler **(0x06)**	Memory access and stack check violation exception. The memory access violation is triggered when a user-mode process (with low privileges) tries to access a reserved high privilege memory area. However, since the

	user application is executed in kernel mode, this exception cannot be triggered, the user application having access granted on all memory areas. Consequently, only the stack checking violation can be triggered.
EV_SWIHandler **(0x08)**	Exception triggered when executing the `swi` instruction. It is used by the debugger for setting up a breakpoint during debugging sessions (in this case, the debugger inserts one `swi` instruction at the memory location of the specified breakpoint). There is no operational reason to explicitly configure an exception handler if not in a debugging session.
EV_TrapHandler **(0x09)**	Exception enforced when executing the `trap_s` instruction. Using this instruction and a connected exception handler is application dependent.
EV_ExtensionHandler **(0x0a)**	Generic exception triggered by multiple sources. In the EM9305 implementation, it is triggered if a floating-point exception is detected (division by zero, …).[8]
EV_DivZeroHandler **(0x0b)**	Exception triggered when an integer division by 0 is detected.
EV_MalignedHandler[9] **(0x0d)**	Exception triggered when a data memory access is done with a misaligned address. This is the case when the address is not an integer multiple of the operand size.

Table 10-5: Supported exceptions list

All these above listed functions can be overridden by a homebrewed user function handler. Note that it is not possible to replace the default handler for all exceptions that could happen in one shot simply by redefining the `NullHandler`. In all cases, the `NullHandler` handler cannot be overwritten. Instead, all exception handlers have to be individually overwritten by a user defined version.

10.2.4 Exceptions management good practices

Unlike interruptions that are very common and that occur all the time asynchronously, an exception is relevant to an unusual situation that is not expected to happen in nominal situations.

In general, an exception handler is the last possibility for the end user to log post-mortem information somewhere (like in retention RAM) for further investigation. Once done, it is recommended to reset the CPU and to restart the end user application.

It is not really recommended to try to resume from the exception like if nothing specific happened because once the exception has been triggered, it is difficult to figure out if the software (or maybe the whole system) is in a stable configuration. For example, is it really worth trying to resume from a “memory misalignment” exception? Isn’t it too dangerous to keep executing the application knowing that such an error occurred?

Consequently, it is recommended to complete the following operations within an exception handler:

- log post mortem data (in RAM retention)
- trigger a GPIO (e.g. to warn an external device)
- reset the CPU

[8] The EM9305 contains the following hardware extensions: CRC, AES, Log2, JLI and FPU. However, only the FPU can generate an exception related to any floating point operations.

[9] The wording `EV_MalignedHandler` can be a little bit confusing here. It should be read `EV_MisalignedHandler` to highlight the fact that this exception occurs with a misaligned data memory address.

11. TIMERS

11.1 OVERVIEW

There are two timer types available in the SDK. While the protocol timer is intended to be fully used by the Bluetooth LE link layer, the universal timer is the one that an end user application can use for its own needs.

11.2 PROTOCOL TIMER

The protocol timer is designed to fulfil timing constraints imposed by the Bluetooth standard at link layer level. It is not expected to be directly handled by the end user application. Doing so might interact with the Bluetooth LE link layer and might end up in connection loss or unexpected failures. However, it is possible to read its value at any time while it is running and use it to measure elapsed time. Since it is driven with the 24 MHz clock, one LSB of this timer counter represents 42 nanoseconds of duration.

Refer to the SDK documentation for a description of the protocol timer. This documentation is provided for informational purpose only.

11.3 UNIVERSAL TIMER

This timer, also summarized as the “unitimer”, can be seen as a general-purpose timer. It provides different ways of using it depending on the end user application’s needs. It can be used as a single one-shot timer, as an expresso timer or as a periodic based timer.

The SDK’s documentation provides a complete documentation with examples on how to use the different flavours, along with the timer driver API. Refer to this documentation for further detailed information.

11.4 CPU TIMERS

The CPU contains two general purpose timers named timer #0 and timer #1.

These timers are continuously counting at a frequency of 48 MHz (period of 20.8 nanoseconds) which is the frequency of the core CPU. When they reached their maximum value, they roll over and restart from 0.

They can be accessed using the auxiliary register access functions. For example, the counter #0 can be read:

```
uint32_t count = ReadAUX(EM_REG_COUNT0);
```

Its value can also be written, for being reset for example:

```
WriteAUX(0, EM_REG_COUNT0);
```

There are two ways of using this type of timers. The first way is the invasive method in which the timer value is forced to 0 when it comes to count for a delay, like shown in the following code snippet:

```
uint32_t count;
WriteAUX(0, EM_REG_COUNT0);
while(ReadAUX(EM_REG_COUNT0 < delay_in_microseconds * 48);
```

With this method, the user shall ensure that nothing else is currently using the timer because its value is enforced to 0.

The other way to use this timer is to use its current value as a start value like shown in the code snippet below:

```
uint32_t start_value = ReadAUX(EM_REG_COUNT0);
while(ReadAUX(EM_REG_COUNT0 - start_value < delay_in_microseconds * 48);
```

The auxiliary registers are listed in the following file provided in the SDK:

`<SDK>/common/9305/includes/aux_registers.h`

It is also the file where the inline functions to read from/write to the auxiliary registers are defined as well.

Since these timers are featured by the ARC processor, the document [9] describes them in detail. It is then suggested that the reader reads the chapter 30 of this document to get a good understanding of these timers.

12. SLEEP MODE

12.1 OVERVIEW

For being able to have a very low power consumption, the EM9305 device can switch off almost all of its subparts. This covers the CPU, most of the RAM blocks (except one for data retention, see §8.2), the NVM, and many other internal blocks.

Only the PML block at hardware level is never powered off during sleep mode.

Switching to sleep mode is relevant when there is no specific action to be done. The CPU can then sleep for a certain amount of time, and this time can be configured by the end user application. When the application does not have anything else to do, a trigger to sleep transition is started.

Note that the system will switch to sleep mode if the foreseen duration is greater than 15 milliseconds, otherwise the system will not go to sleep and will instead enter an active loop waiting for the requested delay.

12.2 DIFFERENT SLEEP MODES

The Table 12-1 lists the different modes in which it is possible to save energy, from the lowest energy saved to the highest energy saved. These modes can be considered as sleep modes, with the sleep wording being understood as a mode where the CPU, and the system in general, does nothing in order to save power.

Mode	Description
CPU Halt	The CPU is powered but executes the “sleep” instruction (on some other CPUs it is also called “halt” instruction). While it is still using energy, its power consumption is reduced compared to a normal working state. In this mode, all gated clocks are disabled however the processor core is not stopped. But the CPU does not fetch any instructions. Exiting from this mode can be done through the debug interface (if previously enabled), from a POR or on interrupt trigger coming from a GPIO or from the PML for example.
Sleep mode	The CPU is powered OFF, along with other blocks. Only the PML is maintained ON to be able to wake up the whole system later on (CPU, …). In this stage more energy is saved compared to executing the “sleep” instruction (see §8.2.2). RAM blocks configured for data retention are also kept powered ON. All other RAM blocks are powered OFF.
Deep sleep mode	The CPU is powered OFF. All RAM blocks are also powered OFF except the blocks that are configured for data retention. From a functional standpoint, this mode is not different from the above mentioned “sleep” mode. However, from a performance standpoint, the deep sleep mode uses a very low frequency clock which is not suitable for Bluetooth LE operations. This mode is the one in which it is possible to save the maximum energy compared to the previous ones.

Table 12-1: Energy saving modes

12.3 HOW THE SLEEP MODE WORKS

Going to sleep mode is done for a specific period of time and the sleep timer controls this behavior. However, the user application has to select either the sleep mode or the deep sleep mode depending if Bluetooth LE operations have to be maintained or not. Doing so will select the clock to be used as summarized in Table 12-2.

Clock	Sleep mode	Operations
LF RC 500 kHz	Regular sleep (so called “sleep mode”)	Suitable for Bluetooth LE operations
LF RC 100 kHz	Deep sleep	Not suitable for Bluetooth LE operations

Table 12-2: Sleep mode vs deep sleep mode clocks

Deep sleep mode provides a lower power consumption, or a better power saving compared to regular sleep mode. The downside is that as mentioned in Table 12-2, this mode is not suitable for BLE operations.

Before switching to the sleep mode, the SW computes the exact time during which the sleep shall occur minus the time needed by the start-up sequence when the CPU will resume from sleep. The goal is to get a very accurate duration of sleep that is used to configure the sleep timer to be very close to the requested duration. This accuracy is mandatory in case Bluetooth LE operations are on-going while periodically switching to sleep mode for saving power. This is required because of the very stringent timing constraints coming from the Bluetooth LE link layer.

If the sleep computed time is less than 15 ms (default value), then the EM9305 will not actually switch to sleep mode. Instead, the CPU will halt by executing the ‘sleep instruction’ (equivalent to ‘halt’ instruction on other CPUs).

Roughly speaking, the time needed for the system to resume from sleep represents a couple of milliseconds (see §12.4), but it more accurately depends on the exact conditions that prevailed when the system switched to sleep mode which can be:

- power supply configuration (voltage multiplier, step-down, …)
- memory configuration and initialization (persistent vs non persistent)
- FW image header CRC integrity check

Once the time is determined, the sleep timer is configured accordingly. Then the SW issues a request to the PML to switch to sleep mode.

The latter will then switch off the CPU, the RAM blocks and all other elements that must be switched off with the exception of the RAM blocks previously configured as persistent memory.

In a bare metal like application i.e. not relying on QP/C, an application is responsible to explicitly request a sleep mode switch by calling the function `SLEEP_MANAGER_GoToSleep()`. This function verifies if switching mode is allowed by looking at the following points:

- sleep mode has not been forbidden by the end user application
- sleep mode duration is greater than the minimal sleep duration (by default 15 ms)
- there is enough time before the next planned wake-up event
- there is no active transport layer action on-going

Otherwise, in a QP/C based application, the user can just get rid of switching to sleep mode since if there are no events to process, the QP/C port will decide by itself if it switches to sleep mode.

Beyond this normal behavior, there are two possibilities to modify the sleep manager behavior by applying different values to the fields listed in Table 12-3 part of the `gPML_Config` structure stored in persistent memory.

Parameter	Description
gPML_Config.minimalSleepTime	Defaulting to 15 milliseconds, it can be changed based on application’s needs
gPML_Config.sleepModeForbiden	Defaulting to false (sleep mode allowed), it can be changed to true to prevent switching to sleep mode

Table 12-3: Parameters that have impact on the sleep mode

These parameters also apply when entering deep sleep mode if requested. Since there is no automatic switch to this mode, it is the user’s responsibility to activate this mode by explicitly calling the function `PML_SystemShutdown()`. This function will manage all operations before switching to deep sleep mode.

In all cases, switching to sleep mode can be enforced by calling the function `PML_GoToSleepWithoutChecks()`. In such a case, the end user application is fully responsible for doing relevant checks. For example, if data is to be sent over the transport layer, enforcing the sleep mode can lead to losing data.

When required, it is recommended to modify the sleep mode behaviour in the `NVM_ConfigModules()` function like it is shown in the following code snippet where the sleep mode is forbidden.

```
Extern PML_Configuration_t gPML_Config;

void NVM_ConfigModules()
{
   // Prevent switching to sleep mode
   gPML_Config.sleepModeForbiden = true;

   // Do other initialization actions here
   …
}
```

The §18 introduces the EM-System by showing an example of bare metal application in which going to sleep mode is explicitly requested.

In case the end user application relies on QP/C, such request is fully handled by the EM QP/C port when in idle task and there is no signal to process. The application example described in §16.4 shows a QP/C based application without any explicit request to enter sleep mode. It is only when the QP/C idle task is called that the sleep mode is requested, and this operation is fully transparent for the application.

12.4 SLEEP MODE WAKE-UP TIME

Resuming from the deep sleep mode and from the sleep mode takes some time with a duration that is less than 2 ms. The various measurements gave a more accurate value around 1.5 ms which may vary a little bit.

The start-up sequence phases are exposed in Figure 12-1 showing what this maximum time of 2 ms actually represents.

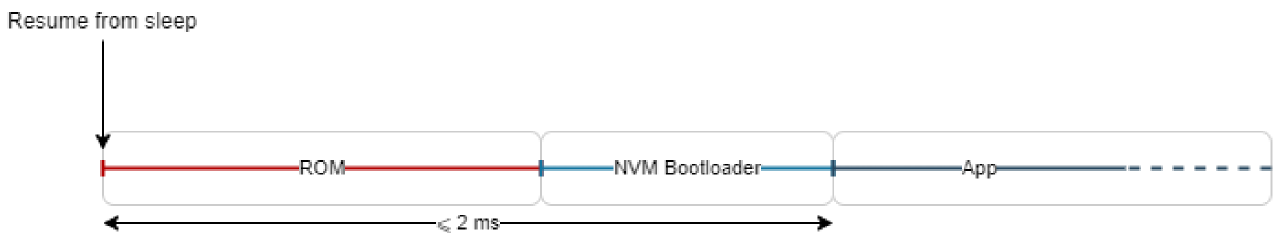


Figure 12-1: Start-up sequence phases and maximum duration

This measurement has been done using the HRS example application from the start of the CPU activity to the first advertising packet generated. This HRS example is represented by the “App” block on the figure. It initializes the Bluetooth LE stack before it can actually advertise. So the time measurement also includes the Bluetooth LE stack initialization.

But for the sake of simplicity, just consider that this maximum start-up time is less than 2 ms.

13. LOCKING SUB-PARTS

13.1 OVERVIEW

This chapter gives a quick overview on the lock bits definition and management so an end user can quickly get started with this concept and can start handling these bits to finalize the device configuration. However, a more complete technical description is given in [4].

The EM9305 has the capability to lock some features and memory areas. The goal is to prevent the device hijacking or tampering. Another goal is to prevent the end user application to be disclosed due to some form of hijacking.

Thus, locking a specific functionality (like the USB for example) is a way to prevent an end user from using it on devices which are not intended to be used in this way.

The locking mechanism is done by the hardware but under control of the software. This means that when a feature lock is activated, any attempt to enforce a forbidden operation by the software will fail. For example, if a NVM page is locked, it is not possible then to erase it or to overwrite it even if the software issues the right erase command. The hardware lock cannot be subverted by any means at software level.

Regarding some specific functionalities, it works in the same way. For example, once the +10dBm is locked, it means that it will never be possible to configure the device to reach a transmit power of +10dBm.

The Table 13-1 lists the EM9305 subparts (features and memory areas) that can be locked.

Subpart	Description	Size *(bits)*
JTAG	Controls if the JTAG I/F is enabled or disabled (default is disabled, not locked).	1
Max Power	Contains the maximum transmit power that can be configured in the radio module. This value cannot be modified and the actual configured power cannot exceed this one.	4
Test mode	Control the test mode lock (locked by default)	1
USB	Controls the USB interface usage if it is enabled or disabled. If this bit is set, the USB interface does not receive any clock	1
Erase Full	Controls whether the complete NVM can be erased (default is NVM can be erased). Full NVM is Main NVM + info pages.	1
Erase Main	Controls whether the main NVM part can be erased (default is NVM can be erased). NVM main part excludes info pages.	1
NVM Main pages	Controls which pages are locked or not (default is no page locked).	64
NVM Info pages	Controls which info pages are locked or not (default is EM Info page locked, user info page is unlocked).	4
NVM Info Page 0 Write	Controls whether writing info page 0 (key container) is allowed or forbidden.	1
NVM Info Page 0 Erase	Controls whether erasing info page 0 (key container) is allowed or forbidden.	1

Table 13-1: Subparts locking control

The JTAG function can be enabled in one out of two possible configurations as summarized in Table 13-2. These two bits are null by default and can be set in case the JTAG function needs to be used. However, if the JTAG lock bit exposed in Table 13-1 is set to 1, these two enable bits will have no effect since the JTAG function would be locked.

JTAG	Enabler	
Jtag2Wires	2 wires protocol enable bit (disabled by default, not set)	1
Jtag4Wires	4 wires protocol enable bit (disabled by default, not set)	1

Table 13-2: JTAG enable bits

These two bits can be used to enable or disable the JTAG function as much as a user need until the function is locked by using the JTAG bit.

When the JTAG lock bit is set to 1, then the JTAG function is automatically disabled whatever is the JTAG 2/4 wires configuration bits.

Having the JTAG not locked by default is helpful for a customer to use it for software debugging purposes. Then, EM highly recommends locking this feature before the complete device is sent to the final end user on the field.

13.2 HOW THE LOCKING PROCESS WORKS

The EM9305 integrates a set of registers which control the locking of some parts and features. They are part of the PML block which is always powered ON unless the battery is removed. By default, in these registers, all bits are set to 0 meaning that nothing is locked.

One example is the PML Lock register that defines the bits to be set to 1 to lock the following features:

- Maximum transmit power
- USB
- JTAG

A zero bit can be set to '1' to lock the corresponding feature. However, it is not possible to set the bit back to zero once it has been set. The only way to achieve this is to do a power off-on cycle which means removing the battery which is not always possible on the end user product.

Beyond simple writing these lock bits in the PML registers, the configuration to be applied can also be permanently written in the NVM in the information pages (the upper part of the NVM). Such configuration is then read by the ROM boot sequence each time the device starts. Once read, it is applied by writing this configuration into the PML lock registers. The Figure 4-2 shows the device start-up sequence executed by the ROM in which at some stage, the lock bits are read from the NVM and applied to actually lock the device according to the requested configuration.

As said earlier, the lock bits are stored in the information pages at the end of the NVM and since there are two configuration space (EM reserved and user reserved), there are two lock configurations.

The first configuration is applied by EM after the production process and before the chip is delivered to the customer. It usually is a configuration agreed with the customer but most of the lock bits are in an opened state. It is then up to the end user to lock the device features according to their plans.

The user still has the possibility to lock other features depending on his need. To achieve that, the user information page shall be used.

Regarding the lock bit mechanism, the user information page contains the same content than the EM information page. However, if the end user has the possibility to lock a feature that is not locked in the EM page (lock bits in user page overwrite unlocked features in EM page), unlocking a feature in the user page that is already locked in the EM page is not possible. The lock bits set in EM page take precedence because in the device start-up sequence these lock bits are applied first (see §4.2).

13.3 READING INFO PAGES

The information pages can be read, and it is the first step to be done prior to making any modification.

This is achieved by executing the `blengine_cli.py` command from the `tools` folder part of the SDK as shown in the following example:

```
# python blengine_cli.py -port COMXX -debug run nvm_read -p em -out_file em_info_page.json
```

Just replace the port identifier by the actual COM port created on your system.

In this example, the EM reserved page is read. It is the default page if the page is not specified. To indicate that the user page shall be read, the `-p em` parameter shall be replaced by `-p user`.

The file introduced by the `-out_file` parameter will be created and will be populated by the values found in the related NVM info page.

The JSON file contains a section dedicated to lock bits. An example of such section is given below:

```
"Lock Bits": {
      "data": {
            "RegNvmLockMain0": {
                  "RegNvmLockMain0": null
            },
            "RegNvmLockMain1": {
                  "RegNvmLockMain1": null
            },
            "RegNvmLockInfo": {
                  "NvmEraseLockInfoPage0": 0,
                  "NvmWriteLockInfoPage0": 0,
                  "NvmLockInfoPage": 8
            },
            "RegNvmLockMaster": {
                  "NvmLockMaster": null,
                  "NvmLockRedund": null,
                  "NvmLockEraseFull": null,
                  "NvmLockEraseMain": null
            },
            "RegNvmKcLockKey": {
                  "NvmKcLockKey": null
            },
            "RegPmlLockBits": {
                  "PmlTxPaPwrLock": 61,
                  "PmlLdoAntTrimLock": 7,
                  «PmlTmLock»: 1,
                  «PmlUsbLock»: 1,
                  «PmlJtagLock»: 0,
                  "PmlJtag2wEn": 0,
                  "PmlJtag4wEn": 0
            }
}
```

In this example, all lock bits are null, the device from which this configuration has been extracted is fully opened. Nothing is locked.

To read the user info page, a very similar command is executed. Note that the page parameter is changed from `em` to `user` and the json file name is also changed from `em_info_page.json` to `user_info_page.json`.

```
# python blengine_cli.py -port COMXX -debug read_nvm_info -page user -file user_info_page.json
```

The extracted file also contains a lock bit section that the user can modify. Once modified, this new configuration is written into the user information page by executing the following command:

```
# python blengine_cli.py -port COMXX -debug write_nvm_info -page user -file
user_info_page.json
```

13.4 SETTING USER LOCK BIT CONFIGURATION

For lock bits that have not been set in the EM information page, the user can change their equivalent values in the USER information page to lock the associated feature.

For example, if the JTAG lock bit is not set in EM information page, then the user has the possibility to activate this lock in the USER information by setting the corresponding bit to 1 (remember, switching from 0 to 1 can be done).

If this bit has already been set to 1 in EM information page, then the user would not have had any chance to unlock this feature (remember, switch a lock bit from 1 to 0 is not possible and has no effect on the lock configuration).

If the end user wants to lock the JTAG feature, the JSON file has to be modified like this:

```
"RegPmlLockBits": {
      "PmlTxPaPwrLock": 63,
      "PmlLdoAntTrimLock": 15,
      "PmlTmLock": 0,
      "PmlUsbLock": 0,
      "PmlJtagLock": 1,
      "PmlJtag2wEn": 0,
      "PmlJtag4wEn": 0
}
```

The 'null' (or 0) statement is replaced by 1 to enable the JTAG lock (check the "PmlJtagLock" keyword in this example). This lock is easy to handle because it has a 1 bit size.

Save the file and execute the following command to write this updated configuration into the user information page:

```
# python blengine_cli.py -port COM5 -debug write_nvm -page user
                                                            --in_file user_info_page.json
```

After this command is completed, reading back the content of the user information page will show up a 1 related to the JTAG lock feature.

At this stage, it is still possible to reset all lock bits in EM information page because this page is not locked. So erasing and rewriting this page can still be done.

All EM9305 chips are shipped to customers with the EM information page in NVM locked to prevent any alteration of this part by the customer or by any other third party.

In the JSON file, the `NvmLockInfoPage` field is used to indicate which information to lock according to Figure 13-1.

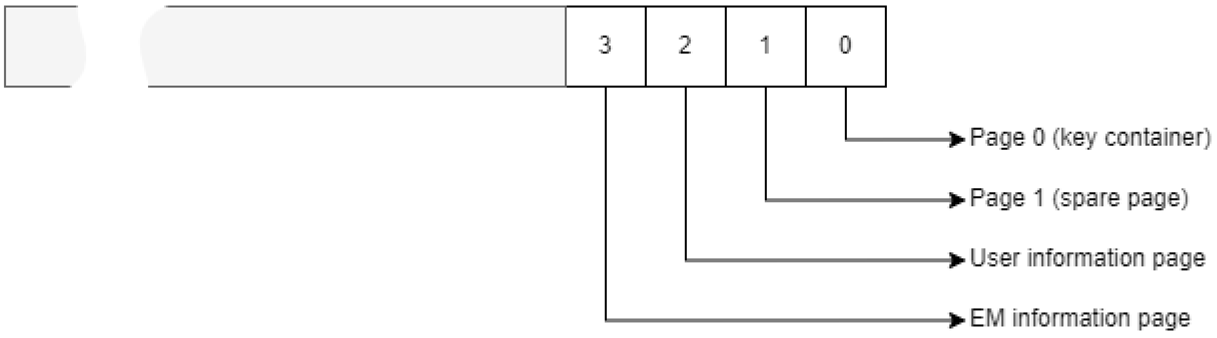


Figure 13-1: Information page lock bits

To lock the user information page, the `NvmLockInfoPage` field shall contain 0x4 according to Figure 13-1. For a device that outgoes the production stage, it is expected that bit 3 is already set to 1 (EM information page locked) and writing 0x4 will have no effect on the lock status for this page. This means that the bit 3 will not be reverted to 0.

14. FIRMWARE UPDATE PROCESS

The firmware update process is introduced with the em|bleu SDK 3.2.

It becomes possible to update any firmware image stored in the non volatile memory by transmitting it to the EM9305 SoC using a Bluetooth LE connection.

The SDK's documentation provides detailed information on the firmware update, how it works and how to set it up. It also covers the process by which a packed firmware image is created before it is uploaded to the device. Please refer to this documentation for further information.

15. OPTIMIZATIONS AND PERFORMANCES

15.1 OVERVIEW

This chapter deals with topics related to power consumption and performance. It provides some information and some clues to improve either performance and/or power consumption.

15.2 MEMORY CRC COMPUTATION

When computing a CRC over a 100 KB of flash memory data set, the execution time is expected to be a constant number of clock cycles, assuming the CRC computation is not preempted. Wherever the CRC computation is performed from the ROM or from the NVM at start-up, the core CRC function is executed from the ROM where it is stored. During this step, it is assumed that the CPU is already running on the high precision HF XTAL clock source.

Based on the ROM versions (V2.0 or V3.0) and the assumption of no preemption, the worst-case execution time to compute a CRC32 over a 100KB data set is expected to be 8 milliseconds. This calculation assumes that the CRC computation is not preempted during the boot sequence.

However, it’s important to note that the CRC calculation function might be used by applications in other contexts where it could potentially be preempted (e.g., in real time multi-tasking environment). In such cases, the execution time may vary depending on the specific circumstances.

16. QUANTUM LEAP QP/C

16.1 OVERVIEW

QP/C (*Quantum Platform in C*) is a lightweight open-source real-time embedded framework and fast context switching RTOS/scheduler. It manages the creation and the scheduling of tasks and provides events and signals processing, which are the primary means of communication between tasks and interrupt handlers. QP/C also provides an integrated state-machine framework.

The QP/C framework can be cooperative or pre-emptive depending on the build configuration. QP/C is cooperative in that a task runs to completion before a scheduled lower priority task can be executed. It can also be pre-emptive, so a lower priority task can be interrupted by the scheduler itself to allow a higher priority task to execute when this higher priority task becomes ready.

In pre-emptive mode, the running application can be interrupted so it can safely enter sleep mode or enter into an endless loop.

A task terminates its execution or run to its completion when the active or executing function returns and gives control back to the scheduler. Once the active function has returned, QP/C will determine if there are any outstanding events requiring service or task scheduled to execute and will take appropriate action. If there are no further actions immediately required, QP/C will enter the idle stated and determine if the EM9305 can enter into sleep mode. If all conditions are met, sleep mode is entered[10].

16.2 STATE MACHINE DIAGRAM

QP/C has a built-in state-machine mechanism by which it is able to trigger tasks switching. This state machine is transparent, but it is always present when a QP/C based application is running.

Based on that, the general structure of applications that are provided in the SDK are the same. Such structure is exposed on Figure 16-1.

At the beginning of the application, one or more tasks are created and activated. Each task enters in its initial state in which some initialization statements can be implemented.

Then, a transition to the idle task is triggered. It is where all signals processing is done. This task is called by the QP/C state machine while there is no other task activation. So it can be considered that the idle task is called in loop.

The idle task body contains a big switch case statement in which some signals can be processed. These signals can be QP/C reserved signals or user defined signals.

Among the QP/C reserved signals, the following ones are processed:

- Q_INIT_SIG
- Q_EXIT_SIG
- Q_ENTRY_SIG

These signals are generated only once and can be used to prepare the application for properly running.

In particular, the `Q_ENTRY_SIG` is emitted by QP/C when every QP/C internal initialization has been completed and then the user can start creating objects for his own purpose. This can be the activation of a timer which will send a user defined signal when it expires, configure GPIOs, register callbacks, …

When the idle ask does not have any signal to process, then the control is given to the QP/C HSM that will decide to switch to sleep mode. Note that this feature is part of the EM9305 QP/C port and is not part of the standard third party QP/C software.

The `basic_app_tutorial` that is described in more details in §16.4 is built like the description above. A more complete example can be found in the `Blinky` example of the official QP/C documentation. So for those who are interested to dive deeper into QP/C, it is recommended to read [7].

[10] Entering sleep mode is automatically achieved. The user application does not have to bother with that.

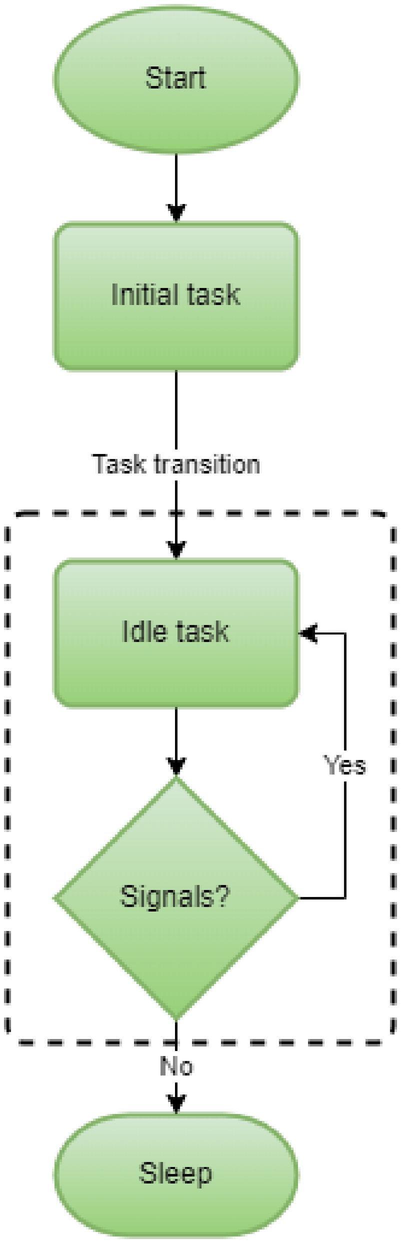


Figure 16-1: QP/C state machine with three states

16.3 HOW TO INTEGRATE QP/C IN AN APPLICATION?

From a general stand point, after a cold reset, the QP/C scheduler shall be initialized. This will populate and initialized QP/C internal structures stored in RAM.

Here is the QP/C function call sequence that shall be done on cold reset:

Function	Goal
QF_Init()	initialize QP/C (needs to be called at least once after cold reset)
Qactive_ctor()	create the idle task
QACTIVE_START()	start the QP/C main task

QACTIVE_POST()	macro used to post an event for further processing withing a task
QF_run()	run QP/C (it comes to life)
QF_bzero()	set the events pool content to null (full initialization)
QF_poolInit()	initialize the pool of events (remember, QP/C is an event driven scheduler)

Table 16-1: QP/C primitive call after a cold reset

Then, after resuming from a sleep period, since the QP/C status is maintained in RAM, there is no need to do another scheduler initialization. It just needs to be resumed. The sequence of QP/C functions call is given in Table 16-2

Function	Goal
QF_bzero()	Set the events pool content to null (full initialization)
QF_poolInit()	Initialize the pool of events since this pool might not be stored in persistent RAM, and thus the list of events has to be cleaned before proceeding.
QF_resume()	Resumes QP/C from where it was just before switching to sleep mode. It is assumed that the QP/C state is kept in persistent RAM.

Table 16-2: QP/C primitive call when resuming from a sleep period

Through QP/C, the application can define tasks (up to 8) that are executed based on events.

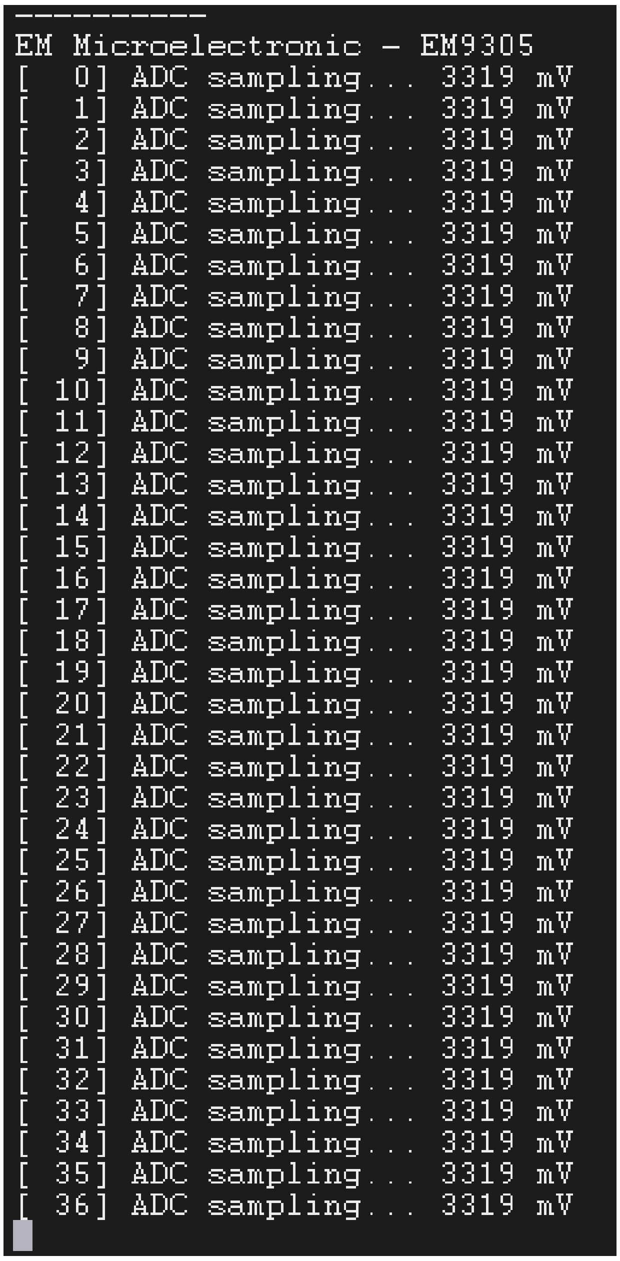
When not executing a task, QP/C puts the application into the idle state in which the device might go to sleep mode. As already said earlier, the EM port of QP/C manages switching to sleep mode by calling the `SLEEP_MANAGER_GoToSleep()` function in the `QK_onIdle()` function after the user idle function is called. So, if the end user application actually relies on the EM9305 QP/C port, the call to the sleep manager function is fully transparent. This is why EM recommends developing applications relying on QP/C.

16.4 EXAMPLE OF COMPLETE QP/C APPLICATION

16.4.1 Overview

This chapter covers a complete basic step by step tutorial on how to write an application that uses QP/C. This application is sampling the V_{bat1} voltage every two seconds using the ADC driver on a periodic basis that is provided by a periodic timer running at a frequency of 0.5Hz. The sampled value is converted to millivolts and is then sent to a host computer through the UART line with no flow control.

The Figure 16-2 shows a screenshot of the console output when this application is running. A measurement of the V_{bat1} is done every 2 seconds as mentioned earlier. The measured raw value is converted to millivolts and then displayed in the console.



```
-----------
EM Microelectronic - EM9305
[   0] ADC sampling... 3319 mV
[   1] ADC sampling... 3319 mV
[   2] ADC sampling... 3319 mV
[   3] ADC sampling... 3319 mV
[   4] ADC sampling... 3319 mV
[   5] ADC sampling... 3319 mV
[   6] ADC sampling... 3319 mV
[   7] ADC sampling... 3319 mV
[   8] ADC sampling... 3319 mV
[   9] ADC sampling... 3319 mV
[  10] ADC sampling... 3319 mV
[  11] ADC sampling... 3319 mV
[  12] ADC sampling... 3319 mV
[  13] ADC sampling... 3319 mV
[  14] ADC sampling... 3319 mV
[  15] ADC sampling... 3319 mV
[  16] ADC sampling... 3319 mV
[  17] ADC sampling... 3319 mV
[  18] ADC sampling... 3319 mV
[  19] ADC sampling... 3319 mV
[  20] ADC sampling... 3319 mV
[  21] ADC sampling... 3319 mV
[  22] ADC sampling... 3319 mV
[  23] ADC sampling... 3319 mV
[  24] ADC sampling... 3319 mV
[  25] ADC sampling... 3319 mV
[  26] ADC sampling... 3319 mV
[  27] ADC sampling... 3319 mV
[  28] ADC sampling... 3319 mV
[  29] ADC sampling... 3319 mV
[  30] ADC sampling... 3319 mV
[  31] ADC sampling... 3319 mV
[  32] ADC sampling... 3319 mV
[  33] ADC sampling... 3319 mV
[  34] ADC sampling... 3319 mV
[  35] ADC sampling... 3319 mV
[  36] ADC sampling... 3319 mV
```

Figure 16-2: Basic application with ADC conversion console output

Thus, using another timer, the application toggles some GPIOs that can be tracked using an oscilloscope or a logic analyzer. The Figure 16-3 shows the trace of the configured GPIOs toggling.

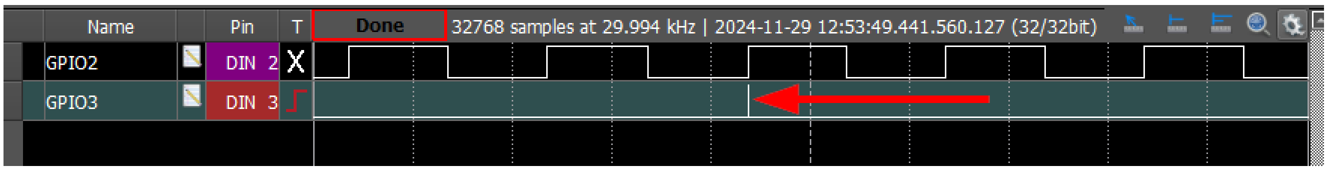

Name
Pin
T
Done
32768 samples at 29.994 kHz | 2024-11-29 12:53:49.441.560.127 (32/32bit)
GPIO2
DIN 2
GPIO3
DIN 3

Figure 16-3: GPIOs toggled by the sample application

This figure shows a pulse on GPIO3. The rising edge of this pulse is representative on a start of the ADC conversion, and the falling edge occurs when the conversion is complete.

As it is shown on Figure 16-4, the sampling duration is about 1 millisecond.

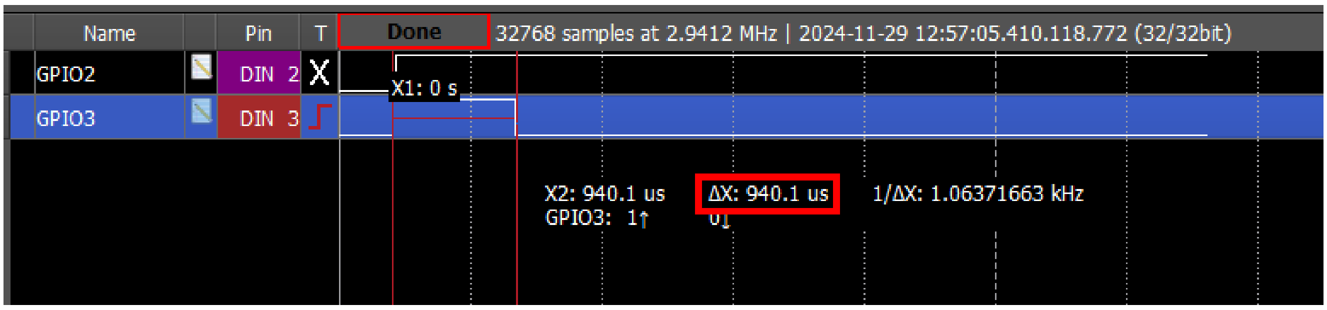


Figure 16-4: Duration of an ADC sampling acquisition

With this sample application, you will learn how to do the following tasks:

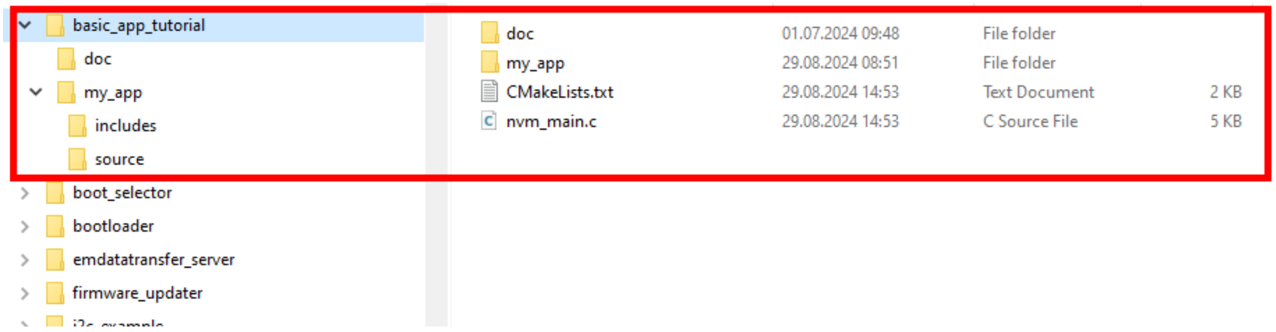
- Get a deep view on an application architecture.
- Create a sample application on a real use case.
- Create and start timers.
- Configure and toggle GPIOs.
- Start an ADC conversion and read the measured value.
- Deal with QP/C events management.
- Print messages to the console.

The complete source code of this example is provided in the SDK through the "basic_app_tutorial" application example. Only relevant code subparts are mentioned in this chapter.

16.4.2 Application structure description

This application structure will follow the one that is exposed in §7. The main goal here is to focus on QP/C primitives or functions to use to complete a real time multi-tasking application.

The `basic_app_tutorial` folder structure provided within the projects SDK's folder is depicted in Figure 16-5.



Name	Date	Type	Size
doc	01.07.2024 09:48	File folder	
my_app	29.08.2024 08:51	File folder	
CMakeLists.txt	29.08.2024 14:53	Text Document	2 KB
nvm_main.c	29.08.2024 14:53	C Source File	5 KB

Figure 16-5: The "basic_app_tutorial" folder structure

It contains the following files and folders:

Item	description

nvm_main.c	This the main file that contains both the `NVM_ConfigModules()` and the `NVM_ApplicationEntry()` functions.
my_app/includes	This folder contains all the local application header files, not to be confused with the header files provided in the SDK.
my_app/includes/my_app_signals.h	This header files defined the user defined signals that are processed by the application. These signals are gathered within the `MyApp_Signal_t` structure.
my_app/includes/my_app_task.h	This header file gives the forward definitions of the functions used to create tasks, start tasks and post events.
my_app/source	This folder contains all the local application source files.
my_app/source/my_app_task.c	This file contains the definition of the functions used to interact with QP/C.
my_app/source/my_app.c	This is the most important file in such it contains the user defined code that gives the actual behavior to the user application.
my_app/source/my_app.h	This file contains all definitions local to the my_app.c file.
CMakeLists.txt	This file is to be used by CMake to build the application example
doc	This folder contains the HTML documentation related to this sample application. This documentation is not as complete as the one exposed in this chapter but it gives a quick overview on the application. It can be read first and if more details are needed, then this chapter should complete the documentation.

Table 16-3: 'basic_app_tutorial' folder structure

Almost all the above-mentioned files can be copied and paste from another application. With the exception of the `my_app.c` file which has to be written for each new application, other files might require very minor modifications or customizations.

16.4.3 The NVM_ConfigModules() function

First the `NVM_ConfigModules()` function is defined. As discussed earlier, it contains the early initialization of the software items that are in use in this example application.

```
// An array in non persistent memory is declared to store events.
static SECTION_NP_NOINIT uint8_t gQpcEventPool[8 * sizeof(QeventParams)];

void NVM_ConfigModules(void)
{
   // Register modules to use
   UART_RegisterModule();
   Timer_RegisterModule();

   // The UART and the timer are enabled in the structure stored in persistent RAM.
   gUART_Config.enabled = true;
   gTimer_Config.enabled = true;
   // Do we resume from sleep period?
   If(PML_DidBootFromSleep())
   {
      // When resuming from a sleep period, we indicate to QP/C that there is no
      // events pool already initialized we could resume from.
      extern uint_fast8_t QF_maxPool_;
      QF_maxPool_ = (uint_fast8_t)0;
   }
   else
   {
      // Initialize the QP/C framework in a clean state by initializing all internal
      // variables. This is mandatory when resuming from a POR on a SW reset.
      QF_init();
   }
   // Initialize the pool events queue.
      MAIN_InitEventPool();
}
```

The UART and the Timer modules are first registered. This is mandatory to be able to use them whatever is the start conditions (cold reset – or POR, SW reset – or warm reset, or resuming from sleep). Note that the GPIO's registration is not needed since it is automatically registered at start-up.

Then both UART and timer modules are enabled in their respective structure stored in persistent RAM. By doing that, they will be enabled again when resuming from sleep. So, in such a case, there is no need to reactivate them.

Since QP/C works with events to trigger pre-programmed actions later on in the idle task, an event pool has to be defined and allocated to handle the events. Since this pool is stored in non-persistent RAM by using the `SECTION_NP_NOINIT` statement, it has to be initialized regardless of the previous state (starting from POR or resuming from sleep). This is done with a call to `QF_bzero()` and `QF_poolInit()` functions in the `MAIN_InitEventPool()` function defined as follow:

```
static void MAIN_InitEventPool(void)
{
   QF_bzero(&gQpcEventPool[0], sizeof(gQpcEventPool));
   QF_poolInit(&gQpcEventPool[0], sizeof(gQpcEventPool), sizeof(QEventParams));
}
```

It shall be noted that this function is declared `static`. Its scope is limited to the `nvm_main.c` file.

And the last thing to do when starting from a POR is to initialize the QP/C framework by calling the `QF_init()` function. Then, the QP/C based scheduling is up and running.

16.4.4 The NVM_ApplicationEntry() function

Then comes the `NVM_ApplicationEntry()` function which is the actual example application entry point.

```
NO_RETURN void NVM_ApplicationEntry(void)
{
   // Initialize the QP/C BSP module.
   BSP_Init() ;

   // Enable interruptions.
   IRQ_EnableInterrupt() ;

   if(PML_DidBootFromSleep())
   {
      // Restart the timers
      Timer_Resume();
      // Restart the ADC
      ADC_Resume();
      // Resume QP/C so it restarts from the previously saved state in persistent RAM.
      (void)QF_resume();
   }
   else
   {
      …
   }
}
```

The first two lines in the `NVM_ApplicationEntry()` function are for BSP initialization and interruptions activation. This has to be done in all cases.

Then, we check if we start this application by resuming from a sleep period (call to `PML_DidBootFromSleep()` function). If it is the case then the timers and the ADC have to be resumed.

And then, QP/C has to be resumed as well by calling the `QF_resume()` function. This function is part of the QP/C ARCv2 port.

When resuming from sleep, it is assumed that the QP/C tasks already exist and calling `QF_resume()` will bring them back up and running. So, in this case, there is no need to create them again.

However, if the application is starting from a cold reset or from a software reset, then the retention RAM does not contain any valuable information that could be resumed.

We then need to create the tasks and to activate them. This is done only once until a POR or a SW reset is triggered.

```
void NVM_ConfigModules(void)
{
      // See code above
}

NO_RETURN void NVM_ApplicationEntry(void)
{
   BSP_Init();
   IRQ_EnableInterrupt();

   if(PML_DidBootFromSleep())
   {
      …
   }
   else
   {
      // Configure the UART (at least Tx) to send messages over the serial line.
      MAIN_ConfigureUART();

      // Call the function that configures the GPIOs
      MAIN_SetupUartGpio();

      // This application will use the ADC so, it needs to be initialized.
```

```
        ADC_Init();

        // Set the clock source to HF crystal to have a better accuracy.
        PML_SetHfClkSourceNonBLocking(PML_HF_CLK_XTAL);

        // Create the user tasks (function defined in tasks.c file).

        MyAppTask_Create();

        // Start the user tasks (function defined in tasks.c file).

        MyAppTask_Start();

        // Start the QP/C scheduler.
        (void)QF_run();
    }
}
```

The first statement in the “else” block calls a user defined function that configures the GPIOs. In this example, the GPIO7 is configured as the UART Tx to send data to the console while some other GPIOs are actually configured as output GPIOs that are toggled in this application later on. These functions are defined within the `nvm_main.c` file and are declared `static` to limit its scope to the file. Their name is here `MAIN_ConfigureUART()` and `MAIN_SetupUartGpio()` but it is up to the end user to define it.

Then the ADC is initialized.

Once done, the source clock to be the HF crystal. Note that the function that is called here is the non-blocking version which instructs the hardware to actually switch to the HF crystal. It takes a little bit of time, but we can safely continue the initialization during this time.

Then, the tasks are created and started before starting the QP/C scheduler.

The last thing to add to the `NVM_ApplicationEntry()` function is an infinite loop after the call to the `QF_run()` function. If everything works in a nominal way, then this step will never be reached but in case of issue, we just halt the CPU. Thus, this infinite loop indicates to the compiler that the `NVM_ApplicationEntry()` never returns which is compliant with the `NO_RETURN` keyword used to declare the function. Otherwise, the compiler will issue an error.

```
NO_RETURN void NVM_ApplicationEntry()
{
    …

    // Entering an infinite loop in which the CPU is halted, only in case of error.
    // But this statement is never reached otherwise.
    While(true)
    {
        HaltCPU();
    }
}
```

And then we are done with the core of the example application.

Now, we can focus on the QP/C side by having a look at how tasks are created, started and how events are processed. All this part is gathered in the `my_app_tasks.c` file.

16.4.5 Tasks management

The `MyAppTask_Create()` function is straightforward. We simply call the QP/C task constructor to create a task with an initial state represented by the `MyAppTask_Initial()` function.

```
Qactive_ctor(&gMyAppTask_AO.super, Q_STATE_CAST(&MyAppTask_Initial));
```

The `activeTask` is an active object (“AO”) which represents a task. It is declared like this:

```
static MyUppTask_AO_t gMyAppTask_AO;
```

and the `MyAppTask_AO_t` type is defined as follow:

```
typedef struct { Qactive super; } MyAppTask_AO_t;
```

It simply contains a `Qactive` object defined by QP/C.

The `startupTask` parameter is the function to be called when the task becomes active. It is defined like this:

```
static Qstate MyAppTask_Initial(MyAppTask_AO_t* const me)
{
   return Q_TRAN(&MyAppTask_Idle);
}
```

When activated, this task simply triggers a transition to the idle task where events processing is taking place. However, some other actions can be done but, in this example, we simply trigger a transition to the idle task that will process the various signals.

Activating, or starting the task is done in the `MyAppTask_Start()` function which is defined like this:

```
void MyAppTask_Start(void)
{
   // Start the task represented by the activeTask. Assign to this task the
   // event queue declare before.
   QACTIVE_START(
      // active object pointer to start.
      &gMyAppTask_AO.super,
      // priority
      (uint_fast8_t)gNextAvailablePriority--,
      // storage for events
      gMyAppTask_EventsQueue,
      // size of events queue (max number of events)
      MY_APP_TASK_EVENTS_QUEUE_SIZE,
      // stack size
      0,
      // initial event
      0,
      NULL);
}
```

The first step is to get a reference to the active task to be started. It is used to tell QP/C which task has to be started through the following statement:

```
&gMyAppTask_AO.super;
```

The priority parameter is provided by QP/C through the `gNextAvailablePriority` variable that is decremented. Since there is only one task created here, there is no real need to decrease this value, but in case of more than one tasks, it shall be done to give different priorities to other additional tasks.

It has to be noted that an event queue is provided for the task so it can handle incoming events. The `taskEventsQueue` is declared earlier in the `tasks.c` file as an array of events with a predefined size. Here, it is defined with a size of up to 8 events. Thus, this array is stored in non-persistent memory and will have to be initialized each time the device will start-up wherever it comes from.

```
Static SECTION_NP_NOINIT const QEvt* gMyAppTask_EventsQueue[MY_APP_TASK_EVENTS_QUEUE_SIZE];
```

The `QEvt` type is a QP/C defined type for events.

For more information on the `QACTIVE_START()` function, refer to the QP/C documentation.

16.4.6 Events processing

The user idle task is handled by the `MyAppTask_Idle()` function which goal is to process events. Its structure is shown below.

```
static QState MyAppTask_Idle(MyAppTask_AO_t* const me, QEvt* pEvt)
{
   // Define a variable to store signal processing results, with a default value.
   QState status = Q_HANDLED();

   // The event is stored
   QEventParams *pEvent = ((QeventParams*)pEvt);
   // It it then processed.
   switch(pEvent->super.sig)
   {
      // The init signal is send when QP/C hierarchical state machine (Hsm) is ready
      // to start tasks transitions.
      case Q_EXIT_SIG:
      case Q_INIT_SIG:
         // No specific process needed here, just let the state machine process the signal.
         Status = Q_SUPER(&QHsm_top);
      break;

      case Q_ENTRY_SIG:
         // Initialize the application (process the INIT signal once at startup)
         MyAppTask_PostEvent(MY_APP_INIT_SIG, NULL);
      break;
      // Any other reserved QP/C signal processing is delegated to the state machine (Hsm).
      default:
         MyApp_EvtStates_t eventStatus = MyApp_HandleEvent((MyApp_Signals_t)pEvent-
>super.sig, pevent->params);
         if(eventStatus == MY_APP_EVT_ST_UNKNOWN)
         {
            // If the event is unknown, let QP/C handle it.
            qstatus = Q_SUPER(&QHsm_top);
         }
         else if(eventStatus == MY_APP_EVT_ST_ERROR)
         {
             // If an error occurred, then trigger a transition to error task
            qstatus = Q_TRAN(&MyAppTask_Error);
         }
      break;
   };

      return(qstatus);
}
```

The `Q_INIT_SIG` signal is sent by QP/C when it becomes possible to issue tasks transitions. In the above example, there is no specific process to be done. Consequently, its process is delegated to the QP/C hierarchical state machine (Hsm). This is the same for the `Q_EXIT_SIG`.

Note that since no user specific process is needed in this example, the explicit processing of these two signals exposed in this example can be removed. They will then be processed in the default statement in which the hierarchical state machine will actually do it.

The `Q_ENTRY_SIG` is an event sent when QP/C is ready. It is sent only once at start-up and its goal is to inform our software that we can start processing all the user defined actions.

When processing this signal, our software will do the following:

- Create the two periodic timers, one at 10 Hz for toggling GPIO9, and one at 1 Hz for starting the Vbat1 sampling using the ADC
- Enable the ADC and register the ADC callback when a sampling operation is completed

16.4.7 Application initialization and events handling

When QP/C is ready to run the user application, it sends the `Q_ENTRY_SIG` signal. This signal is processed within the `MyAppTask_Idle()` function that is the user defined idle function. On reception of this signal, we simply post a user defined initialization signal that is the `MY_APP_INIT_SIG` signal. Note that this signal is part of the enumerate definition located in `includes/my_app_signals.h`.

When control is given back to QP/C, it actually sends the signal `MY_APP_INIT_SIG` which will then be processes within the my_app_task.c idle function. Since this signal is a user define, we call then the `MyApp_HandleEvent()` with this signal as a parameter. This function is a user defined function in which the application initialization code is executed.

It is within this function that the timers are created and the ADC is initialized properly.

The Figure 16-6 shows the corresponding sequence diagram.

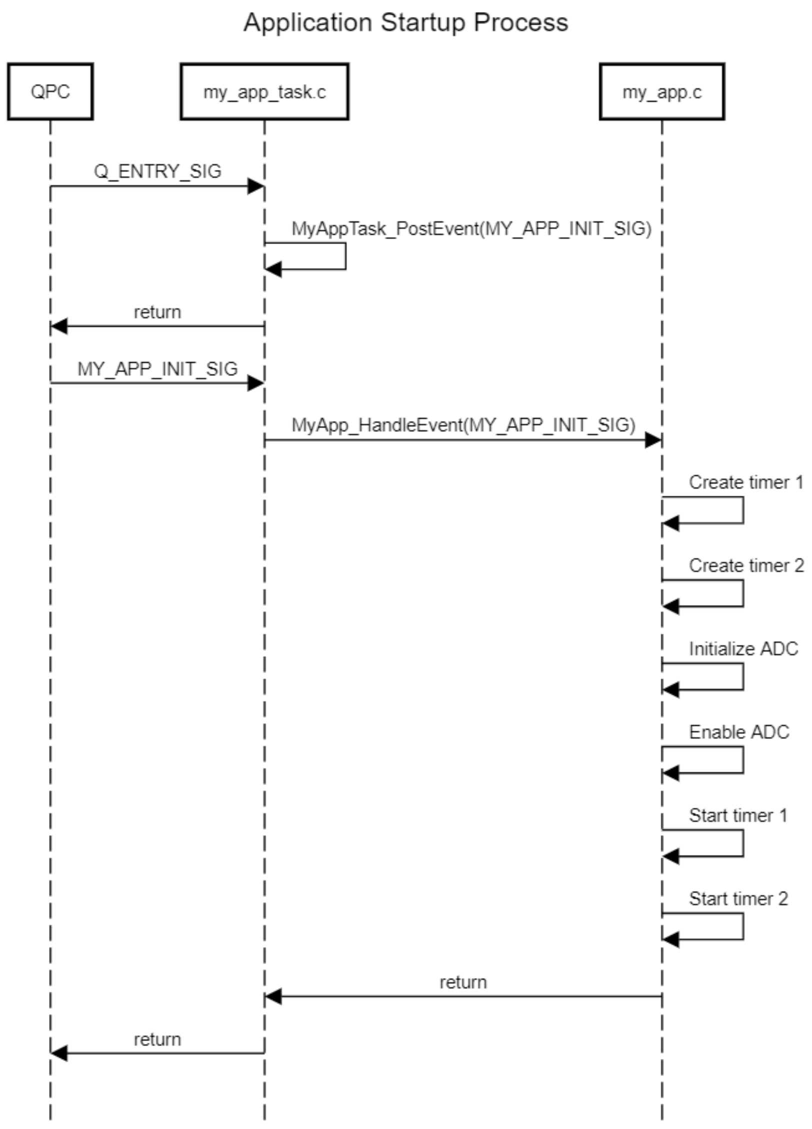


Figure 16-6: Sample application startup process

The code snippet below shows this process.

The `Q_ENTRY_SIG` signal is processed first in the `MyAppTask_Idle()` function when it is received. As already mentioned, the processing consists in posting the user defined `MY_APP_INIT_SIG` signal.

```
static QState MyAppTask_Idle(…)
{
   case Q_ENTRY_SIG:
      // Send the INIT_SIG signal. No parameters are needed.
      MyAppTask_PostEvent(MY_APP_INIT_SIG, NULL);
      break;
```

Then control falls back to QP/C.

In case the signal is not the `Q_ENTRY_SIG` (in fact for any other signal like our user defined `MY_APP_INIT_SIG` signal), we call our user defined event handling function.

```
   default:
      // Call our user defined signal handling function so these signals have
      // a chance of being processed.
      MyApp_EvtStates_t eventStatus = MyApp_HandleEvent((MyApp_Signals_t)pEvent->super.sig,
pEvent->params);
```

This function (we will see it later) will process all signals it knows, or all user defined signals.

In case it receives an unknown signal, it will return `MY_APP_EVT_ST_UNKNOWN`, meaning that this signal will have to be processed by QP/C itself. In this case, we call the `Q_SUPER(&QHsm_top)` function to give control back to QP/C hierarchical state machine for this event processing.

```
if(eventStatus == MY_APP_EVT_ST_UNKNOWN)
{
   // Let QP/C process this signal
   qstatus = Q_SUPER(&QHsm_top);
}
```

Sometimes, however, an error can occur when the user defined events handling function detects an error. In such a case, it returns the `MY_APP_EVT_ST_ERROR` which means that we require to enter error mode.

```
else if(eventStatus == MY_APP_EVT_ST_ERROR)
{
   // Trigger a transition to the error task
   qstatus = Q_TRAN(&MyAppTask_Error);
}
```

As already mentioned, the function `MyApp_HandleEvent()` is where almost all the magic takes place. This is where the user writes his/her own code. This is done by writing a function that processes all user defined signal, and maybe some system signals depending on the goal of the application.

```
MyApp_EvtStates_t MyApp_HandleEvent(MyApp_Signals_t signal, void *pParams)
{
      switch(signal)
      {
           case MY_APP_INIT_SIG:
           // Processing of the user defined initialization signal
```

```
            break;
            case MY_APP_TIMER_TICK:
            // Processing of the timer tick that occurs 10 times per seconds
            break;
            case MY_APP_ADC_TIMER_TICK:
            // Processing of the timer running at 1Hz that triggers one ADC
            // conversion (one per second).
            break;
            case MY_APP_ADC_CONVERSION_DONE:
            // ADC conversion is complete. Read the measured value.
            ADC_GetValue(&value);
        };
}
```

It is under the `MY_APP_INIT_SIG` that all initializations take place. This concerns the timers and the ADC.

The code snippet below shows the steps used to initialize the timer 1, the timer 2 and the ADC.

```
MyApp_EvtStates_t MyApp_HandleEvent(MyApp_Signals_t signal, void *pParams)
{
   uint8_t Timer1, Timer2;
…
   case MY_APP_INIT_SIG:
      // Create the timer at 10 Hz.
      Timer1 = Timer_SchedulePolling(10.0f, (Driver_Callback_t)timerCallback, NULL);
      // If timer creation is successful, enable it.
      if(timer1 >= 0)
      {
         Timer_Enable(timer1);
      }
      else
      {
         // It has not been possible to create the timer. Post a user defined
         // timer error event for further processing.
         MyAppTask_PostEvent(MY_APP_TIMER_FAILURE, NULL);
      }
      // Create the timer at 1 Hz.
      Timer2 = Timer_SchedulePolling(1.0f, (Driver_Callback_t)adcTimerCallback, NULL);
      // If timer creation is successful, enable it.
      if(timer2 >= 0)
      {
         Timer_Enable(timer2);
      }
      else
      {
         MyAppTask_PostEvent(MY_APP_TIMER_FAILURE, NULL);
      }

      // Configure the ADC source.
      ADC_ConfigSourceVBAT1();
      // Enable the ADC, however no sampling is started here!
      ADC_Enable();
```

```
        // Register the function that will be called once one single sampling is completed.
        ADC_RegisterCallback(ADC_Conversion_Callback);

break;
```

Note that when a timer is enabled, it starts counting.

It shall be noted in this example that a user defined error event is posted in case a timer cannot be created for any reason. This event will be processed at the next iteration of the function `MyApp_HandleEvent()`. It is a good practice to process any error that could occur.

After the timers have been created, the ADC is configured and prepared for being used. Note that no ADC conversion is started unless explicitly requested for.

16.4.8 Timers event processing

As you might have seen in the example application, user callback functions are provided. This is the place where the user code has to be written when the timers expire.

The dedicated `Timer_Callback()` function is defined for handling interruptions coming from the timer 1. The goal of this function is to send the user defined event `MY_APP_TIMER_TICK` which process is deferred into the `MyApp_HandleEvent()` function already exposed earlier.

The Figure 16-7 shows a sequence diagram that highlights how the timer tick signal is triggered and processed.

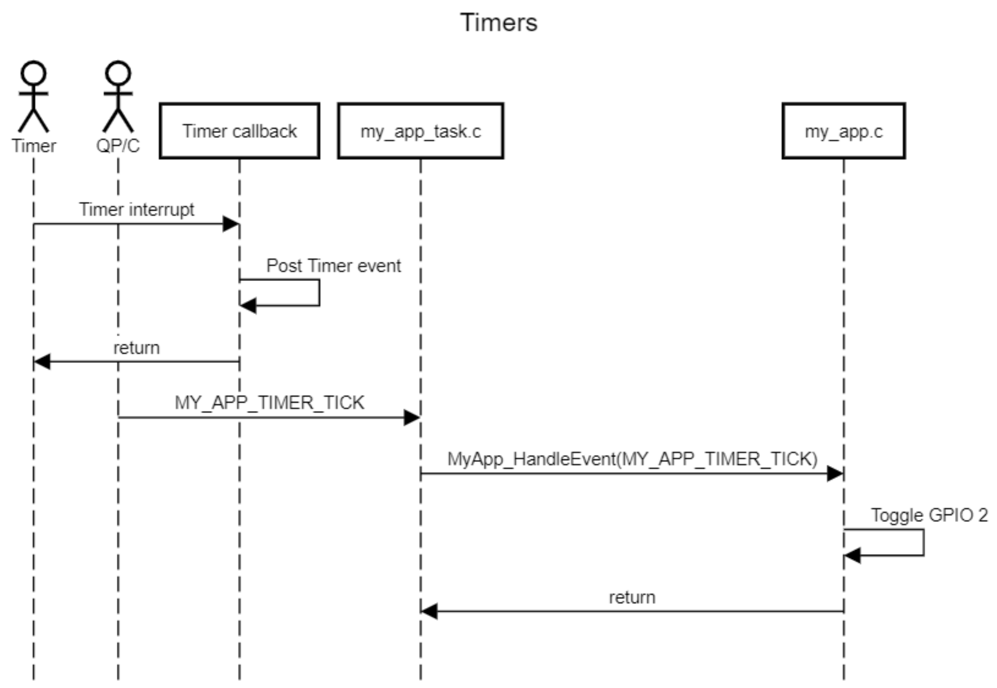


Figure 16-7: Timer callback mechanism

Once the timer has been created and started, the timer hardware triggers an interruption at the proper pace. For the first timer of the application, it runs at 10 Hz.

On timer interrupt, the timer callback is called.

This callback posts the event MY_APP_TIMER_TICK.

Once QP/C regain control, it processes the MY_APP_TIMER_TICK by calling the user defined function for handling events.

The user defined events handling function processes the timer tick by toggling a GPIO.

The code below shows the `Timer_Callback()` callback function that is sent every 100 ms (10 Hz) each time the timer hardware triggers an interrupt.

```
static void Timer_Callback(Driver_Status_t status, void* pUserData)
{
   // Check if internal interruption counter is not null.
   // Otherwise,Increment interrupt counter
   QK_ISR_ENTRY();
   // Send the event MY_APP_TIMER_TICK.
   MyAppTask_PostEvent(MY_APP_TIMER_TICK, NULL);

   // Decrement interrupt counter
   QK_ISR_EXIT();
}
```

The code snippet below shows the actions done on processing the timer tick signal.

```
MyApp_EvtStates_t MyApp_HandleEvent(MyApp_Signals_t signal, void *pParams)
{
      switch(event->super.sig)
      {
         …
         // Processing the user defined timer2 signal
         case MY_APP_TIMER_TICK:
         // Toggle GPIO 2 so we can know when a single sampling is started.
         GPIO_Toggle(2);
         break;
…
}
```

As you can see here, there is nothing fancy. But more complex processing can be done depending on the application’s goal.

The 1 Hz timer user defined callback function’s goal is to send another user defined signal which is `MY_APP_ADC_TIMER_TICK`. As above, this signal is handled in the `MyApp_HandleEvent()` function, and its goal is to start one single ADC sampling.

```
MyApp_EvtStates_t MyApp_HandleEvent(MyApp_Signals_t signal, void *pParams)
{
switch(event->super.sig)
{
   …
   // Processing the user defined timer2 signal
   case MY_APP_ADC_TIMER_TICK:
      // Then start the sampling.
      ADC_StartWithCallback();
   break;
…
}
```

16.4.9 Analog to digital sampling

The ADC sampling operation is an asynchronous operation. This means that the software instructs the hardware to start sampling a voltage value, but such operation takes some time.

Since starting one ADC conversion is not a blocking function, the software can keep running and doing something else while the sampling operation is in progress.

Then, when the ADC has completed the conversion, it triggers an interruption that is catched in a dedicated handler in the same way it is done for the timers interruptions.

As already stated, no ADC sampling operation is started when calling the `ADC_Enable()` function. Actual single sampling will be done later on a periodic basis from `timer2` trigger.

When the ADC sampling operation is complete, the user defined callback `ADC_Conversion_Callback()` function is called. Its goal is to send the user defined signal `MY_APP_ADC_CONVERSION_DONE signal` for being processed inside the `MyApp_HandleEvent()` function.

Sending the event is done in a similar way than sending the `MY_APP_TIMER_TICK` shown earlier. The only difference is the signal name.

Then, in the `MyApp_HandleEvent()` function, the ADC signal is processed like this:

```
MyApp_EvtStates_t MyApp_HandleEvent(MyApp_Signals_t signal, void *pParams)
{
   switch(signal)
   {
      …
      // Processing the user defined ADC signal
      case MY_APP_ADC_CONVERSION_NONE:
      // Declare a variable that will contain the sampled Vbat1 value
         uint16_t value;

         // Read the sampled value
         ADC_GetValue(&value);

      // Set the GPIO 9 to low to visualize when the ADC has completed his work
         GPIO_SetLow(9);

      // Send the value over the UART line after conversion to millivolts
      printf("V=%d mv\r\n", ADC_ValueToMillivolt(value));
      break;
…
}
```

The sampled value is retrieved from the ADC driver, converted to millivolts and sent over the serial line to the host computer to be displayed on the console.

16.4.10 Error processing

When an error is detected, an error signal is posted for further processing. This is the case for example if a timer cannot be created or used. In this case, the MY_APP_TIMER_FAILURE signal is posted.

Now what to do in such a case?

There several ways to handle an error. In this example, the processing is very simple and consists in achieving the following operations:

1. print a message over the console
2. wait one second
3. halt the CPU

However, simply halting the CPU might be a little bit brutal. If the CPU cannot be woken up from this state, then the device is simply lost.

Instead of halting the CPU within the MyApp_HandleEvent() function, another way of handling an error is to return an error status. When this value is received, a transition to an error dedicated task is done. This process is done in the `MyAppTask_Idle()` function.

```
static QState MyAppTask_Idle(…)
{
      // Process the incoming event
      MyApp_EvtStates_t eventStatus = MyApp_HandleEvent(…);

      // If an error has been detected, switch to the error task
      if(eventStatus == MY_APP_EVT_ST_ERROR)
      {
         qstatus = Q_TRAN(&MyAppTask_Error);
      }
      …
}
```

The error task function definition is up to the end user depending on the application’s goal. A very simple one can be the following:

```
static QState MyAppTask_Error(…)
{
      // An unresolvable error has occurred. Restart the CPU to see if it solves the issue.
      ResetCPU();
}
```

And that’s it on how to write an application that relies on QP/C for its scheduling and events management. However, this sample application does not deal with any Bluetooth LE communication. The reader is encouraged to have a deep look at some more examples that showcase how to implement some Bluetooth LE profiles and services, like the `nvm_emb_hrs` (heart rate sensor) and the `nvm_emb_fit` (fitness) for example.

This will introduce the em|bleu Bluetooth LE stack to the user.

16.4.11 User defined signals

This example heavily relies on the use of signals. There are QP/C reserved signal and user defined signal. Since the QP/C reserved signal are already defined, the user signals need to be defined as the `MyApp_Signals_t` enumerated type. This is done in the my_app_`tasks.h` file.

```
typedef enum
{
    // Signals for the application task.
    MY_APP_INIT_SIG = MY_APP_TASK_SIG_START,
    MY_APP_ERROR_SIG,

    /* Add other signals for the application task here. */
    MY_APP_TEMP_INDICATOR_ISR,
    MY_APP_TIMER_TICK,
    MY_APP_ADC_TIMER_TICK,
    MY_APP_ADC_CONVERSION_DONE,

    /* Error signals */
    MY_APP_TIMER_FAILURE

} MyApp_Signals_t;
```

To avoid any collision with QP/C reserved signal values, the first user signal is defined to be equal to `Q_USER_SIG`. All signal numbers above this value provided by QP/C will be considered as unique user defined signals.

This example is provided with detailed explanation to showcase how a structured event driven QP/C application example can be written. Some other details of less importance could have been exposed here as well but it is not really worth doing it. It is however recommended to open the complete full source code of this application to get a good understanding of how this application works.

16.5 USEFUL LINKS TO GO FURTHER

To dive deeper into how to use QP/C, a good starting point would be to analyse the provided examples like the Blinky example which source code can be accessed here:

https://github.com/QuantumLeaps/qpc/tree/master/examples/workstation/blinky

A more complete and general QP/C documentation is available online and can be found at the following address:

https://www.state-machine.com/qpc

At the time this implementer's guide is written, the QP/C version is the 7.1.3.

A complete API reference can be found here:

https://www.state-machine.com/qpc/api.html

Moreover, a full feature free e-book can be downloaded here:

Practical UML statecharts in C/C++

Finally, EM provides the `nvm_qpc_example` application that can be used as a baseline to design other applications relying on QP/C.

17. EM|BLEU BLUETOOTH LE STACK

17.1 OVERVIEW

The SDK comes along with the em|bleu Bluetooth LE stack that can be used to add Bluetooth LE connectivity to any application running on the EM9305 device.

This stack provides:

- link layer
- host
- profiles
- sample application examples
- technical documentation

The SDK provides a set of libraries that can be used to achieve specific behavior from a Bluetooth LE standpoint. A sample application can be configured with a “central” role simply by linking it with the proper library.

The available ready to use libraries are listed in Table 17-1.

Library	Functional behavior
Controller	Configures the device as a controller only receiving its HCI commands from a transport layer. The host must be implemented on another device.
Peripheral legacy	Configures the device as a 4.2 compliant peripheral device. It advertises and accepts incoming connections. This library does not support any BLE features above 4.2, as a result, applications consume fewer resources.
Central	Configures the device as a central device. It can scan its Bluetooth neighborhood and can initiate connection.
Peripheral	Configures the device as a peripheral device. It advertises and can accept incoming connections.
Pawr_advertiser	Configures the device as a peripheral device and enables additional functionality for the Periodic Advertising with Response feature. In this mode, the device advertises, accepts incoming connections, and handles all responsibilities for a PawR device in the peripheral role.
Pawr_synchronizer	Configures the device as a central device and enables additional functionality for the Periodic Advertising with Response feature. It can scan its Bluetooth neighborhood, initiate connections, and handle all responsibilities for a PawR device in the central role.
Peripheral extended advertiser	Configures the device as a peripheral device that supports the Advertising Extensions feature introduced in BLE 5.0.
Central extended advertiser	Central with Advertising Extensions: Configures the device as a central device that supports the Advertising Extensions feature introduced in BLE 5.0.
Controller with ISO	A 5.4 compliant BLE controller with the Isochronous channels feature enabled. Supports up to 4 BIS streams and 2 CIS streams.

Table 17-1: List of provided ready to use Bluetooth LE libraries

It is possible to build different type of applications from a controller embedding the link layer only to a complete standalone host application exposing different profiles.

In case the device is used as a controller, it accepts receiving standardized and custom HCI commands through the UART, the SPI or the USB transport layer.

A set of sample application examples are also provided in the SDK. For each example, the source code is provided along with the different files required to build them for the EM9305 target.

Thus, the §17.2 dives into an extended advertising sample application step by step description to showcase how this type of application can be written, and also to introduce common usage of the Bluetooth LE stack. For a better understanding, it is recommended to read this chapter with the source code of the `nvm_emb_dts_ext_adv` sample application example.

This sample application does not address all the capabilities of the em|bleu Bluetooth LE stack so it also recommended to dive into its dedicated documentation provided in the SDK.

For a deep dive into the em|bleu Bluetooth LE stack, please refer to the technical documentation provided in the SDK.

17.2 EXTENDED ADVERTISING BLUETOOTH LE APPLICATION EXAMPLE

17.2.1 Overview

This application example extends the previous example by introducing the em|bleu Bluetooth LE stack. By doing so, it gives the previous example over the air communication capabilities.

It is a step-by-step tutorial that smoothly introduces the user on how to write an application that will do the following actions:

- Advertise using extended advertising
- Allow incoming connection and act as a central
- Periodically send data to the peripheral

Since this tutorial explains how to use functions of the em|bleu API to achieve a specific behavior regarding the Bluetooth LE protocol, it is assumed that the reader already has a knowledge of this protocol and knows where to find information in the Bluetooth LE stack core specification when relevant.

Before going further in getting detailed information on the em|bleu stack implementation, the reader is encouraged to read [8].

The SDK provides a Bluetooth LE stack technical documentation that can be helpful to write Bluetooth LE applications. The reader is encouraged to read this documentation before following the example application description given in this implementer’s guide. The SDK Bluetooth LE documentation can be accessed as shown in Figure 17-1.

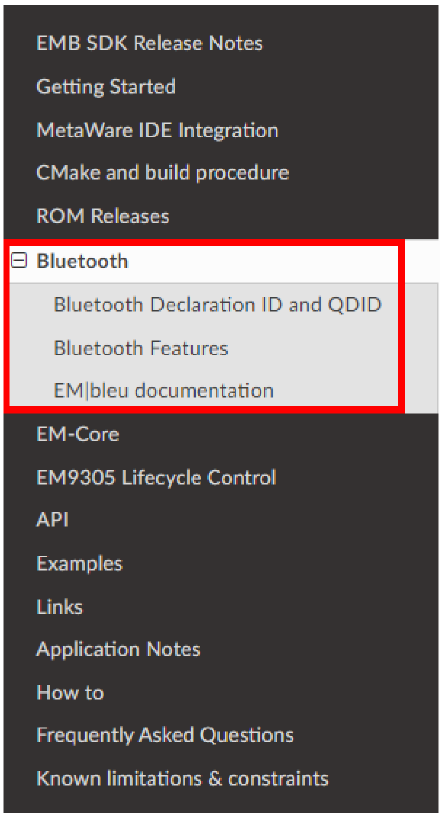


Figure 17-1: Access to Bluetooth LE SDK documentation

A special feature of the em|bleu stack is that its API is at a very low level. This means that almost everything can be configured and fine-tuned from an end user perspective. The flip side is that it requires a better knowledge of its API and to achieve a certain functionality, the end user application has to do all the requested atomic operations. There is no high level API that would abstract a set of lower level functions.

The main pros for the em|bleu stack that should be kept in mind are the following:

☞ The em|bleu stack is very configurable and versatile to meet the exact needs of the end user.

☞ The em|bleu stack may overwrite any settings made in the radio driver. However, this feature can be disabled.

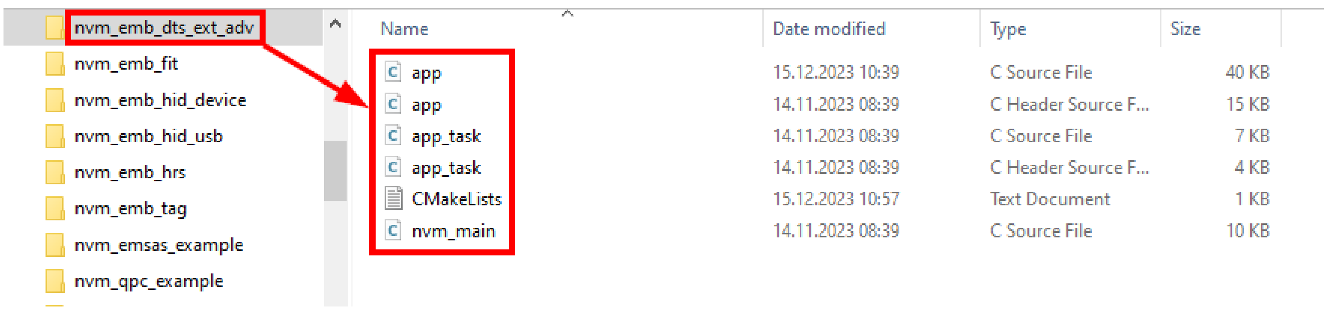
This example shows how to implement a device that implements extending advertising.

Starting from the example application described in §16.4, the following code adds the integration of the Bluetooth LE stack.

The explanations to cover this step-by-step tutorial directly come from the content of [8]. It is recommended that the reader walks through this tutorial and then looks at all relevant details provided in such a document.

17.2.2 Project file structure

This sample application folder structure is very simple. It is depicted in Figure 17-2 and shows a flat structure containing all required files.



Name	Date modified	Type	Size
app	15.12.2023 10:39	C Source File	40 KB
app	14.11.2023 08:39	C Header Source F...	15 KB
app_task	14.11.2023 08:39	C Source File	7 KB
app_task	14.11.2023 08:39	C Header Source F...	4 KB
CMakeLists	15.12.2023 10:57	Text Document	1 KB
nvm_main	14.11.2023 08:39	C Source File	10 KB

Figure 17-2: Folder structure for extended advertising sample application

The Table 17-2 lists the source files that compose this sample application along with a short description of each file.

File	Description
CMakeLists.txt	Contains directives to create the build environment
nvm_main.c	The main file containing the user defined `NVM_ConfigModules()` and `NVM_ApplicationEntry()` functions.
app_task.c	The file containing everything related to the tasks creation
app_task.h	The header file related to the app_task.c file
app.c	The main file that contains all the mechanic to setup and manage the extended advertising
app.h	The header file related to the app.c file

Table 17-2: Files structure for the extended advertising sample application

17.2.3 The CMakeLists.txt file

The first step to do is to change the project name at the beginning of the file:

```
project(nvm_emb_dts_ext_adv C)
```

The name of this application is "NVM EMB Data Transfer Server Extended Advertising" which is shortened with "nvm_emb_dts_ext_adv".

Then, the `CMakeLists.txt` file shall be extended to include the Bluetooth LE stack. Depending on the goal to be achieved, one or more dedicated libraries shall be part of the list of libraries.

The advertising feature corresponds to a specific profile that relies on top of the Bluetooth LE host, which in turn relies on top of the Bluetooth LE link layer. Consequently, all three dedicated libraries shall be added like exposed in the following code snippet:

```
# Set the libraries to link the final app with.
SET(${PROJECT_NAME}_LIBS
    ${NVM_COMMON_LIBS}
    uart_dma
    gpio
    printf
    QPC
    unitimer
    adc
    # Include the PAwR library
    emb_pawr_advertiser
)
```

Having these libraries listed here will make linking your application with the Bluetooth LE stack successful.

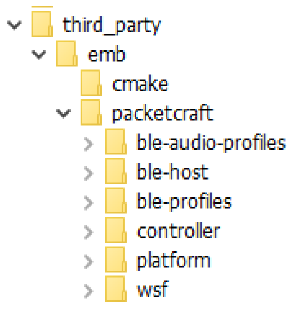
Thus, to be able to compile the application, then all paths to the Bluetooth LE stack header files are needed. This is achieved by completing the statement below by a reserved `CMake` keyword:

```
# Set the include path to header files provided by the SDK.
SET(${PROJECT_NAME}_INCLUDES
    ${COMMON_INCLUDES}
    ${EMB_COMMON_INCLUDES}
)
```

All header files that will be included in the application example will be found in the provided paths.

For information, all the header files related to the Bluetooth LE em|bleu stack are in different folders within the following one:

```
<SDK>/libs/third_party/emb/packetcraft
```



```
third_party
  emb
    cmake
    packetcraft
      ble-audio-profiles
      ble-host
      ble-profiles
      controller
      platform
      wsf
```

Figure 17-3: Bluetooth LE em|bleu SDK's folder structure

The following folders provide the different Bluetooth LE layer headers to be included depending on the functionality to be achieved. The Table 17-3 gives a short description of these layers.

Folder	Description
ble-audio-profiles	Audio profiles header files (AICPC, AICPS, BAPBA, …)
ble-host	Host layer header files
ble-profiles	Profiles layer header files
controller	Link layer related files
platform	Specific header files depending on the platform. Contains files for the EM9305 platform.
wsf	Contains all WSF related header files

Table 17-3: em|bleu Bluetooth LE header files structure

17.2.4 Analysis of the extended advertising sample application

Since this application relies on QP/C, it is event driven.

From a general standpoint, if there is no signal to process, the QP/C port will try to enter into sleep mode for energy saving. If there are still one or more signals to be processed, then the system will not enter into sleep and will process the remaining signals.

The initialization process's main role is to prepare the device for being up and running to advertise. To do so, a set of steps must be completed before the device is actually ready to work.

This process is a two steps process.

17.2.4.1. First step

The first step is shown in Figure 17-4.

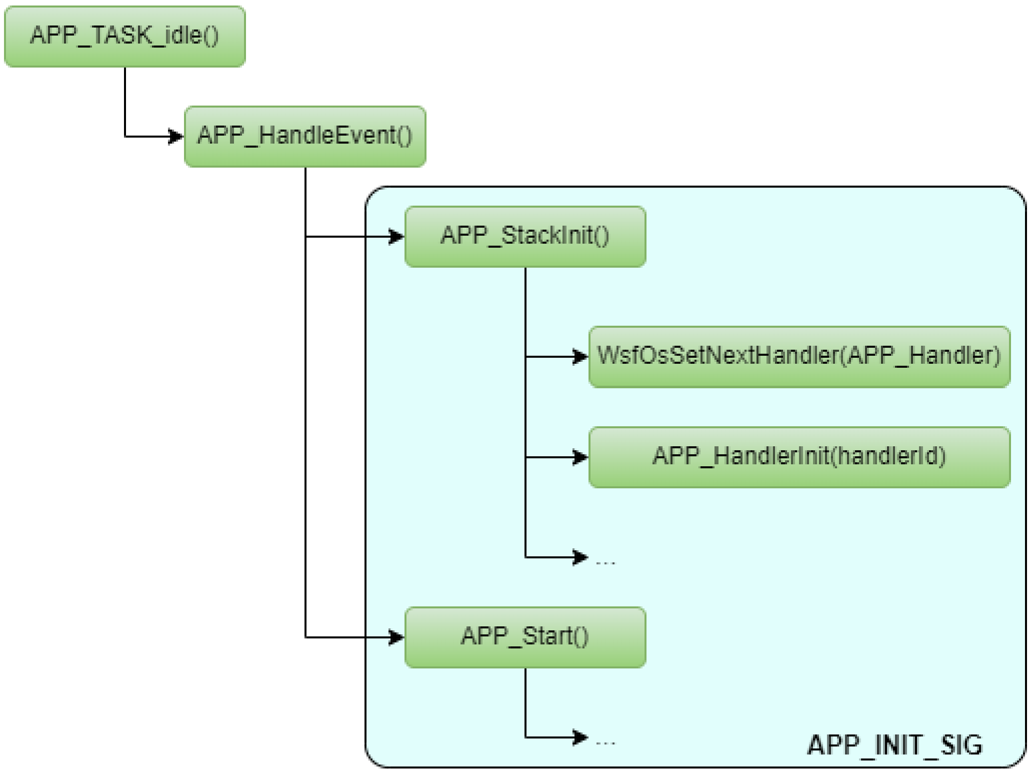


Figure 17-4: Extended advertising sample application initialization step 1

A quite common `APP_TASK_idle()` function is where events are processed. On reception of the `APP_INIT_SIG` signal, the Bluetooth LE stack is initialized and started.

The initialization process registers the special user level function or handler that will be called by the Bluetooth LE stack when the Bluetooth LE stack will need to send notifications to the user level application. This registration is done with the following code snippet:

```
// The user level Bluetooth LE stack event handler, to process Bluetooth events
static void APP_Handler(wsfEventMask_t event, wsfMsgHdr_t *pMsg)
{
      …
}
```

```
// Bluetooth LE stack initialization function called on reception of APP_SIG_INIT
void APP_StackInit(void)
{
      …
      handlerId = WsfOsSetNextHandler(APP_Handler);
      …
}
```

The user level function `APP_Handler()` is registered to the WSF layer so Bluetooth LE stack events can be processed.

Once done, the `APP_HandlerInit()` function is called. This function is in charge of configuring the required functionalities like the characteristics of extended advertising.

To give access of the Bluetooth LE stack to such parameters, a set of dedicated pointers are defined by the Bluetooth LE stack. They are exported so the end user application can manipulate them to have them pointing to specific structures containing the user defined parameters. The code snippet below shows an example on how to proceed.

First, a structure of type `appExtAdvCfg_t` which shall be used to define extended advertising configuration shall be declared as follow:

```
static const appExtAdvCfg_t gcAppExtAdvCfg =
{
      // Advertising durations in ms (0 = infinite).
      { 0},
      // Advertising interval is in 0.625 ms units .Thus, the effective interval
      // value is 160 * 0.625 = 100 ms. More than one interval duration can be specified.
      { 160},
      // maxEaEvents : These are events that Controller will send prior to
      // connection termination
      { 0},
      // useLegacyPdu : If used with extended Advertisement then Advertising length
      // cannot be grater than 31 bytes.
      {FALSE},
      // perAdvInterval : Advertising intervals for periodic advertising in 1.25 ms
      // units . So the effective periodic interval value is 160 * 1.25 = 200 ms.
      {160}
};
```

Then, the address of this structure is given to the Bluetooth LE stack through the use of the reserved pointer like this:

```
static void APP_HandlerInit(wsfHandlerId_t handlerId)
{
      …
      // Set the Bluetooth LE stack extended advertising defined pointer to the address
      // of the structure that contains the extended advertising parameters.
      pAppExtAdvCfg = (appExtAdvCfg_t *) &gcAppExtAdvCfg;
      …
}
```

The same principle is used for the following stack configuration:

- Application configured with a slave role
- Connection parameters
- Security

For example, one parameter that can be setup for the device acting as a slave device is the number of incoming connections it can accept. In the current example, we want only one connection, so we create the following structure:

```
static const appSlaveCfg_t gcAppSlaveCfg =
{
      // limit number of Connections to 1
      1
};
```

Then, still in the `APP_HandlerInit()` function, the dedicated reserved pointer is setup to point to this structure:

```
pAppSlaveCfg = (appSlaveCfg_t *) &gcAppSlaveCfg;
```

Then, the Bluetooth LE stack application framework can be configured in such a way with the right initialization function call:

```
AppSlaveInit();
```

The complete code can be checked in the function `APP_HandlerInit()` on line #608 in the file `<sdk>/projects/nvm_emb_dts_ext_adv/app.c`.

After the Bluetooth LE stack initialization is completed, the Bluetooth LE stack can be prepared for running by setting various callback functions and by issuing a device manager reset. After this reset, the Bluetooth LE stack will be up and running.

This process is done within the function `APP_Start()` on line #481 of the file `app.c`.

The first step is to register a callback function that the device manager will call when scan and advertisement events need to be processed.

Next, a connection event callback function is registered.

Both above-mentioned operations are done like this:

```
static void APP_Start(void)
{
      // Register a callback function for scan and advertising events processing
      DmRegister(APP_DmCback);
```

```
    // Register a callback function for connection events processing.
    DmConnRegister(DM_CLIENT_ID_APP, APP_DmCback);

    …
}
```

It can be highlighted here that the same callback function is use for:

- Scan events
- Advertising events
- Connection events

The callback looks like this:

```
static void APP_DmCback(dmEvt_t *pEvt)
{
    dmEvt_t *pMsg;
    uint16_t len;

    len = DmSizeOfEvt(pEvt);

    // Allocate some memory to store the incoming message
    if ((pMsg = WsfMsgAlloc(len)) != NULL)
    {
        // Copy the incoming message in the structure that will be used to transfer
        // it to the WSF layer
        memcpy(pMsg, pEvt, len);
        // Send this DM event message to wsf event handler
        WsfMsgSend(gAppHandlerId, pMsg);
    }
}
```

The device manager event is forwarded to the WSF for further processing. This will lead to other signal emissions that will be processed within the `APP_Handler()` function that has been registered earlier.

17.2.4.2. Second step

The second step of this application initialization is to process the first event that is sent by the Bluetooth LE stack device manager after it has completed its reset operation. Remember, after having configured all parameters that have to be used for configuring the device, a reset of the device manager is triggered.

Since a Bluetooth LE stack user level handler has been put in place earlier in this process, it is called whenever the device manager issues the `DM_RESET_CMPL_IND` event just after resuming from the reset.

The overall picture that depicts this behavior is given in Figure 17-5.

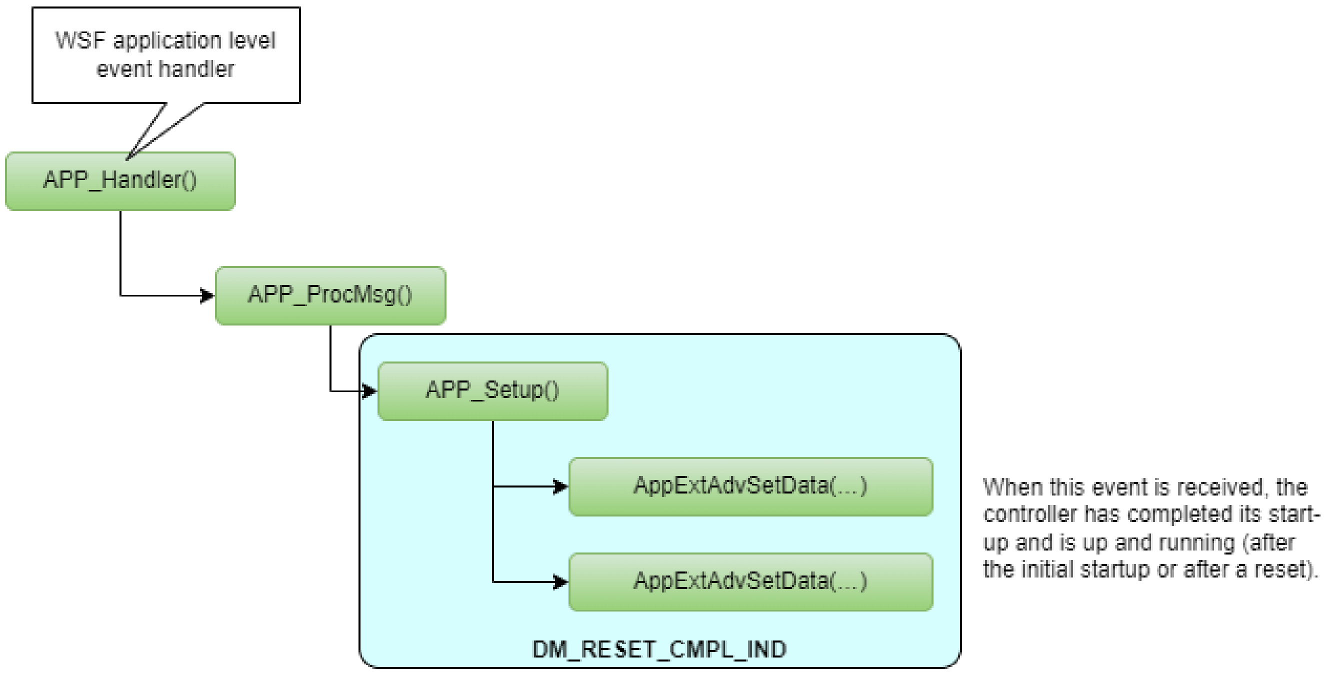

WSF application level event handler
APP_Handler()
APP_ProcMsg()
APP_Setup()
AppExtAdvSetData(...)
AppExtAdvSetData(...)
DM_RESET_CMPL_IND
When this event is received, the controller has completed its start-up and is up and running (after the initial startup or after a reset).

Figure 17-5: Extended advertising initialization process second stage

The user level `APP_Handler()` function is exposed in the file `app.c` on line #639.

This function processes incoming events. It is able to filter them out by checking the numerical range of the event itself, leading to different function calls provided by the Bluetooth LE stack depending on the event that has been triggered.

For instance, if the event is triggered by the device manager, its value should range from `DM_CBACK_START` and `DM_CBACK_END`. Among this range, we find the event sent when the advertising starts or stops, an event sent when a scan is initiated by a device, or the event that is sent when connection is opened, and so on.

So, what happens when the event `DM_RESET_CMPL_IND` is received? In this case, we enter into the `APP_ProcMsg()` function in which this event is processed individually, as it is shown on the following code snippet:

```
static void APP_ProcMsg(appMsg_t *pMsg)
{
      switch (pMsg->hdr.event)
      {
            /* Device Manager Reset Complete Indication event */
            case DM_RESET_CMPL_IND:
                  /* Generate hash for all the GATT database values */
                  AttsCalculateDbHash();
                  /* Generate the ECC Key for LE Secure connection use*/
                  DmSecGenerateEccKeyReq();
                  /* Read all the device database records from NVM*/
                  AppDbNvmReadAll();
                  /* Start the process of setting adv data and starting extended Advertisement*/
                  APP_Setup(pMsg);
            break;

      }
}
```

Some operations are triggered depending on the functionalities that need to be achieved. For example, if it is planned to configure the EM9305 with connection capabilities, then a key must be generated for further secured connections.

The last part of processing this event is to set some characteristics of the advertising. This is done in the `APP_Setup()` function where the some parameters related to advertising are set. Some of these parameters are shown in Table 17-4.

Parameter	Value
DM_ADV_TYPE_FLAGS	discoverable flag + no basic rate/enhanced data rate
DM_ADV_TYPE_LOCAL_NAME	“EMB DTS”
DM_ADV_TYPE_TX_POWER	0 dBm
DM_ADV_TYPE_MANUFACTURER	a set of bytes representing the manufacturer data

Table 17-4: Setting some parameters related to advertising

For example, configuring the parameter `DM_ADV_TYPE_LOCAL_NAME` will allow the device to show up “EMB DTS” on a device that scan the area for getting advertising data, in order to identify the EM9305 device running this sample application.

Still in the `APP_Setup()` function, it is possible to do specific processing when the device manager informs the user level application that it has started or stopped to advertise. In this case, the events `DM_ADV_SET_START_IND` and `DM_ADV_SET_STOP_IND` are received. They can then be processed by adding the following statements:

```
static void APP_Setup(appMsg_t *pMsg)
{
      // Process device manager reset indication
      case DM_RESET_CMIL_IND:
      …
      break;

      // Process start advertising indication
      case DM_ADV_SET_START_IND:
      …
      break;

      // Process stop advertising indication
      case DM_ADV_SET_STOP_IND:
      …
      break;
}
```

In this example, nothing is required to be done but it could be used to synchronize the advertising activity with debugging or measurement for example.

When it comes to send data as soon as a connection is established, then the event `DM_CONN_OPEN_IND` can be used to get synchronized. When processing this event, then a communication channel can be opened to send data for example. And the opposite event DM_CONN_CLOSE_IND can be used to know when the connection is lost.

Consequently, the following statements can be added if they are required:

```
static void APP_Setup(appMsg_t *pMsg)
{
      …
      // Open a communication channel with the device that requested the connection.
      case DM_CONN_OPEN_IND:
            …
      Break;

      // Close the communication channel
      case DM_CONN_CLOSE_IND:
            …
      break;
}
```

As can be seen through this example, an application that uses the em|bleu Bluetooth LE stack is event driven. After configuration and setting various callback, the Bluetooth LE stack starts working and sends notifications, or events/indications, to notify the end user application for specific processing.

And that’s it right now for this example introducing usage of the Bluetooth LE stack. The reader is encouraged to dive into the provided example and to read the Bluetooth LE stack documentation as well to get a better understanding on how it works.

17.3 PERSISTENT BONDING DATABASE (EMB DATABASE)

The EMB Database provides a way to persistently store BLE key material exchanged during pairing. The keys are stored in NVM Info Page 1 by default and persist across power cycles. It is designed to securely store keys when the devices resets for any reason including thermal shutdown.

The EMB Database is not designed to be used with the Packetcraft Application Framework profile (AF). The AF has a RAM database that stores similar device records which do not persist across power cycles.

17.3.1 Example usage storing in NVM Info Page 1

This example is not to a complete application. It is only meant to point the user in the right direction to use this library.

The user application is responsible for handling receiving keys from the stack in the app and passing them to the EMB Database API. The user must initialize the record first, before handling key info from the DM event, storing into the record, before passing to the EMB Database API.

From a high-level perspective, this code snippet does the following:

- Creates EMBDb_Record_t (pairing info) in RAM and populates various key fields during pairing.
- Stores pairing info in NVM after pairing completes successfully.
- Loads pairing info from NVM when the BLE stack initializes.

- Accepts encryption requests from the peer device if that peer has previously paired.
- Pairing info persists after power on resets.

```
#include "emb_database_api.h"

// Database record in RAM to be stored in NVM after pairing completes.
static embDb_record_t pairingInfo;
// Connection ID to be stored when a connection is established.
static dmConnId_t     connId;
// Storage backend for EMB Database to use. In this example, NVM Info P1 is used.
static embDb_storageBackend_t storageBackend;

static void getReceivedKeys(dmEvt_t *pDmEvt)
{
      // The emb_database has fields for each type of BLE key exchanged during
      // pairing (LTK, IRK, and CSRK).
      switch (pDmEvt->keyInd.type)
      {
            case DM_KEY_LOCAL_LTK:
                  pairingInfo.localLtk = pDmEvt->keyInd.keyData.ltk;
                  pairingInfo.localLtkSecLevel = pDmEvt->keyInd.secLevel;
                  pairingInfo.localLtkValid = 1;
                  break;

            case DM_KEY_PEER_LTK:
                  pairingInfo.peerLtk = pDmEvt->keyInd.keyData.ltk;
                  pairingInfo.peerLtkSecLevel = pDmEvt->keyInd.secLevel;
                  pairingInfo.peerLtkValid = 1;
                  break;

            case DM_KEY_IRK:
                  pairingInfo.peerIrk = pDmEvt->keyInd.keyData.irk;
                  BdaCpy(pairingInfo.peerIrk.bdAddr, pDmEvt->keyInd.keyData.irk.bdAddr);
                  pairingInfo.peerIrkValid = 1;
                  break;

            case DM_KEY_CSRK:
                  pairingInfo.peerCsrk = pDmEvt->keyInd.keyData.csrk;
                  pairingInfo.peerCsrkSecLevel = pDmEvt->keyInd.secLevel;
                  pairingInfo.peerCsrkValid = 1;
                  break;

            default:
                  break;
      }
}

// This function is called by the user application's message handler when a
// Device Manager (DM) event has been received. The DM communicates back to
// the user application by sending various events. In the emb_fit example,
// the application's message handler is APP_FitHandler.
```

```
void userDMEventHandler(dmEvt_t *pDmEvt)
{
      switch (pDmEvt->hdr.event)
      {
            case DM_RESET_CMPL_IND:
                  // Initialize the EMB Database to use NVM Info P1 backend (default).
                  // By default embDb_LoadNVMInfoP1Backend sets the maxRecords to 8,
                  // but this can be changed to a maximum of 69 records if desired.
                  // storageBackend.maxRecords = 69;
                  EMBDB_LoadNVMInfoP1Backend(&storageBackend);
                  EMBDB_RegisterBackend(&storageBackend);
                  // Load the stored pairing info if it exists, otherwise create empty record.
                  if (EMB_DB_GET_RECORD_SUCCESS == EMBDB_GetLastRecordStored(&pairingInfo))
                  {
                        // Could setup device filtering and IRK resolution here.
                  }
                  else
                  {
                        EMBDB_InitializeRecord(&pairingInfo);
                  }
                  break;

            case DM_CONN_OPEN_IND:
                  connId = (dmConnId_t)pDmEvt->hdr.param;
                  break;

            case DM_SEC_KEY_IND:
                  getReceivedKeys(pDmEvt);
                  break;

            case DM_SEC_PAIR_CMPL_IND:
                  EMBDB_StoreRecord(&pairingInfo);
                  break;

            // This example showcases using the stored LTK for encrypting the connection.
            // A similar process is applied for the other BLE keys.
            case DM_SEC_LTK_REQ_IND:
                  if (pairingInfo.localLtkValid == 1)
                  {
                        // Key has already been exchanged and stored.
                        DmSecLtkRsp(
                        connId, TRUE,
                        pairingInfo.localLtkSecLevel, pairingInfo.localLtk.key
                        );
                  }
                  else
                  {
                        // Key not found.
                        DmSecLtkRsp(connId, FALSE, 0, NULL);
                  }
                  break;
            default:
                  break;

      };
}
```

17.3.2 Example usage storing in DRAM

The location in which the database records are stored is configurable by changing the read/write backend of the database. This can be done using the `EMBDB_RegisterBackend` API. in the example above, records are stored in NVM Info page 1, but can be changed to store in different locations.

Below is an example of how to tell the EMB Database to store records in DRAM.

```
#include "emb_database_api.h"
#include "errno.h"

// We are 8 using storage records here, but this can be configured with respect
// to available space in the storage backend. If using NVM, the maximum is
// 69 records using database v1.0 as this is the number of records that will
// fit in an 8kB page.
#define EMB_DB_MAX_RECORDS 8U

// DRAM storage implemented using an array.
static EMBDB_Record_t databaseDRAM[EMB_DB_MAX_RECORDS];
// Storage backend for EMB Database to use. In this example, DRAM is used.
static EMBDB_StorageBackend_t backendDRAM;

// Function to erase the data. Data is only erased when requested by the user.
static int32_t eraseDbDRAM(void)
{
      memset(databaseDRAM, 0x00U, sizeof(databaseDRAM));

      return 0;
}

// Function to invalidate records.
static int32_t invalidateRecordAtAddressDRAM(EMBDB_Record_t *address)
{
      address->metadata.recordValid = EMB_DB_RECORD_INVALID;

      return 0;
}

// Function to write records into the DRAM array.
static int32_t writeRecordAtAddressDRAM(const EMBDB_Record_t *record, const void
*storeAddress)
{
      EMBDB_Record_t *pRecord = databaseDRAM;
      int32_t          status   = -ENOSPC;

      for (uint8_t i = 0U; i < EMB_DB_MAX_RECORDS; i++)
```

```
    {
        if (storeAddress == pRecord)
        {
            databaseDRAM[i] = *record;
            status          = 0;
            break;
        }
        pRecord++;
    }

    return status;
}

// In user function where database is to be initialized...
static void userInitializeBondingDatabase(void)
{
    memset(databaseDRAM, 0x00U, sizeof(databaseDRAM));

    backendDRAM.storageStartAddress        = databaseDRAM;
    backendDRAM.databaseSize               = EMB_DB_MAX_RECORDS * sizeof(EMBDB_Record_t);
    backendDRAM.writeRecordAtAddress       = writeRecordAtAddressDRAM;
    backendDRAM.invalidateRecordAtAddress = invalidateRecordAtAddressDRAM;
    backendDRAM.eraseDatabase              = eraseDbDRAM;
    backendDRAM.erasedValue                = 0x00000000;
    backendDRAM.maxRecords                 = EMB_DB_MAX_RECORDS;

    // Check the registerStatus is 0 to ensure backendDRAM structure is setup
    // correctly and the read/write location is suitable for the EMB Database.
    // This function may return -ENXIO if the storage location is unknown to
    // the SoC, but the backend may still work.
    int32_t validationStatus = EMBDB_ValidateBackend(&backendDRAM);

    // If this status is 0, one of the function pointers are NULL.
    int32_t registerStatus = EMBDB_RegisterBackend(&backendDRAM);
}
```

18. USING EM SYSTEM

The EM System library had been quickly introduced in §4.3.

This paragraph deals with details on how to add this library in an application and how to use it.

Adding the features provided by the EM System library is straightforward. This library shall be mentioned in the list of libraries to link within the application `CMakeLists.txt` file, like in the following code snippet:

```
PROJECT(my_application C)
     …
SET(${PROJECT_NAME}_COMMON_LIBS
   Metaware
   em_core
   em_system
   nvm_entry
   …
)
…
```

Then, in the C source file, the related header files have to be included like this:

```
#include "em_system.h"
#include "em_transport_manager.h"
…
```

Since the EM System uses the transport layer which can be over SPI, UART or USB, the transport manager header file shall be included as well.

The EM System library defines a set of HCI commands through an array that has to be registered when the application starts-up after the EM System is initialized or resumed depending on if the system starts up from POR or if it resumes from a sleep period.

When this is done, the transport manager can be initialized and started. This operation has to be done when no Bluetooth LE stack is used. This will add a communication interface to the application either on the SPI or the UART communication bus.

When the Bluetooth LE link layer is in use wihout having the host part integrated, there is no need to initialize the transport layer. This is automatically done by the Bluetooth LE link layer in order to be able to receive HCI commands from this communication layer. and to send HCI events on it. Consequently, the Bluetooth LE stack registers its own HCI transport manager callback.

The code snippet below shows an example on how to fully integrate the EM System in an application.

```
extern const EMSystem_CommandParser_t gEMSCmdRomCommandParsers[];
extern const uint16_t gEMSCmdRomNumberOfCommandParsers;

NO_RETURN void NVM_ApplicationEntry(void)
{
   if (!PML_DidBootFromSleep())
   {
      // Initialize the EM System after a POR or SW reset
      (void)EMSystem_Init();

      // Register the set of commands to be supported.
      (void)EMSystem_RegisterCommandsHandler(gEMSCmdRomCommandParsers,
gEMSCmdRomNumberOfCommandParsers);

      // Initialize and start the EM transport manager with an internal callback.
      // This is only mandatory when the Bluetooth LE link layer is not used.
      (void)EMTransportManager_InitWithSleepCB(NULL,  EMTRANSPORTMANAGER_RX_BUFFER_MAX_SIZE,
4u, EMTRANSPORTMANAGER_TX_BUFFER_MAX_SIZE, 4u);
}
   else
   {
      // Resume the EM system by initializing used variable in non persistent memory.
      EMSystem_Resume();

      // Resume the EM transport manager only if no Bluetooth LE stack link layer is in use.
      EMTransportManager_Resume();
   }
}
```

The function `EMTransportManager_InitWithSleepCB()` registers an internal callback function that will be called when the system is about to go to sleep mode. This function is called by the sleep manager and checks if there is an on-going data transfer. If this is the case, the function will instruct the sleep manager not to switch to sleep mode, otherwise the transferred data could be lost. If there is no on-going data transfer, then the function instructs the sleep manager that it is safe to switch to sleep mode.

Since the example above does not use the Bluetooth LE stack link layer, then the EM transport manager has to be explicitly initialized.

When resuming from sleep, the only thing that needs to be done is to “resume” the EM system. This operation will initialize variables it uses from non-persistent memory since their content has not been saved during the sleep period.

As for resuming the EM transport manager, and as already mentioned earlier, this step is only needed when there is no Bluetooth LE stack link layer in use as it is shown in the example above. Otherwise, the Bluetooth LE stack takes care of this operation and nothing needs to be done at application level.

Thus, resuming the transport manager is a matter of initializing all variables and states it actually uses and stored in non-persistent memory.

When all layers are in place, it is then time to enter an infinite loop in which the transport manager events are processed. In case there is nothing to process, the system is instructed to enter into the sleep mode.

The code snippet below shows how to proceed.

```
NO_RETURN void NVM_ApplicationEntry(void)
{
   if(!PML_DidBootFromSleep())
   {
      …
   }
   else
   {
      …
   }

   while(true)
   {
      // Interruptions configuration is saved and interrupts are disabled.
      uint32_t irqThreshold = IRQ_DisableAndSaveInterrupts();

      // All transport manager requests are processed.
      irqThreshold = EMTransportManager_Process(irqThreshold);

      // Everything has been processed, then request to enter into sleep mode
      irqThreshold = SLEEP_MANAGER_GoToSleep(irqThreshold);

      // If we reach this point, then it means that the request to go to sleep mode
      // has been cancelled for some reasons.
      IRQ_RestoreInterrupts(irqThreshold);
   }
}
```

Before processing any transport event, the interruptions need to be disabled so no interrupt can occur during this processing. Note that the interruptions configuration is saved for being restored later on.

The next step is to actually process any transport manager event which are received HCI commands or HCI events to be sent over the transport layer. When nothing else needs to be done, a request is done to enter into sleep mode. It is just a request, so, depending on several conditions, it might be cancelled by the sleep manager.

In case the device actually goes to sleep mode then the last step of the loop (function `IRQ_RestoreInterrupts()`) will never be executed since the CPU will be powered OFF for the sleep duration. At next wake-up, the software will be executed from the start.

If the sleep mode request is cancelled, then the last step is actually executed. It is just here to restore the interruptions configurations.

Then this process is repeated forever.

Note that this example does neither rely on QP/C for real time tasks management nor use the Bluetooth LE stack. It is a bare metal application.

When EM System is included into a QP/C based application, the infinite loop is not needed anymore. It is instead handled by a task that is activated when QP/C is resumed and through the processing of a signal that wakes-up the system. Thus, there is no need to request entering sleep mode since it is automatically handled by QP/C from the idle task when nothing else needs to be done.

The following code snippet shows how the event “entering active mode” is sent over the transport layer when the application starts in an application that uses QP/C.

```
static QState task_idle(QL_Task_t* const this, QEvt* pEvt)
{
   EventParams* pEvent = (QEventParams*)pEvt;

   switch(pEvent->super.sig)
   {
      case Q_ENTRY_SIG:
         uint8_t eventParams[] = { EVENT_VENDOR_CODE_ENTER_ACTIVE_MODE };
         EMTransportManager_SendEvent(EVENT_VENDOR_SPECIFIC,             sizeof(eventParams),
eventParams, NULL);
      break;

      default:
         Q_SUPER(&QHsm_top);
      break;
   }
}
```

When entering the idle task, the `Q_ENTRY_SIG` signal is processed. This signal is sent the first time the idle task is entered so this signal is only processed once. Then, the HCI "enter active mode" event is sent over the transport manager by using the function `EMTransportManager_SendEvent()`.

Since a QP/C application is event driven, there is no need to create an infinite loop. Instead, events are sent when signals are processed, with can be QP/C signals (like the `Q_ENTRY_SIG` signal) or user defined signals.

And finally, other signals in this example are processed by the QP/C scheduler itself as it is written in the `default` statement where their processing is delegated to the QP/C hierarchical state machine.

19. DEBUGGING AN APPLICATION

19.1 OVERVIEW

Debugging is a critical process in software development that helps identify and resolve issues or bugs within an application. Regardless of your programming expertise, it is almost inevitable that you will encounter unexpected behavior or errors in your code. These issues can range from simple syntax errors to complex logical flaws that affect the functionality of your application. Debugging is the key to tracking down these problems and ensuring the smooth operation of your software.

19.2 DEBUGGING TOOLS AND FEATURES

To facilitate the debugging process, our SDK provides a set of powerful tools and features. These tools aid developers in pinpointing problems and gaining valuable insights into the application's behavior. Some of the essential debugging tools include:

- Debugger probe and Metaware debugger: inspect the state of the program by setting breakpoints, step through the program, track the state of a variable, etc.
- GPIO debugging: place GPIO toggles at specific code points to trace the application's flow.
- `printf` debugging: output log messages at different points in the code to trace the application's flow and monitor variable values. Note that to print messages, the UART line shall be enabled on GPIO7 (Tx) and/or GPIO6 (Rx).
- Assertions: combined with the `printf` feature, triggering assertions at different location of the code can also help understanding why an issue occurs.

The following paragraph deals with the different ways of debugging an application.

19.3 DEBUGGER PROBE AND THE METAWARE DEBUGGER

19.3.1 Compilation for debugging

The sleep mode must be disabled to be able to debug. At top of the `nvm_main.c` file, you need to include the global PML Configuration data structure as follows:

```
#include “pml.h”

extern PML_Configuration_t gPML_Config;
```

On the DVK, the JTAG pins are assigned to specific GPIOs according to the following table:

JTAG pin name	GPIO number
TDI	8
TDO	9
TCK	10
TMS	11

Table 19-1: JTAG pin assignment on the DVK

In NVM_ConfigModules function, the sleep mode has to be disabled and the JTAG must be enabled:

```
// Disable sleep mode if JTAG debug is used.
gPML_Config.sleepModeForbiden = true;

// Enable JTAG.
GPIO_EnableJtag4Wires();
```

Alternatively, the `GPIO_EnableJtag2Wires()` function can be called to use only 2 GPIOs for JTAG, the TMS and TCK pins.

If you are not familiar with the firmware build process, you can refer to §6 - CMake build process. Once the firmware is built, it can be uploaded to the target using the BLEngine tool.

19.3.2 Setup the debugging probe

The JTAG-HS2 debugging probe must be connected to the DVK as follows:

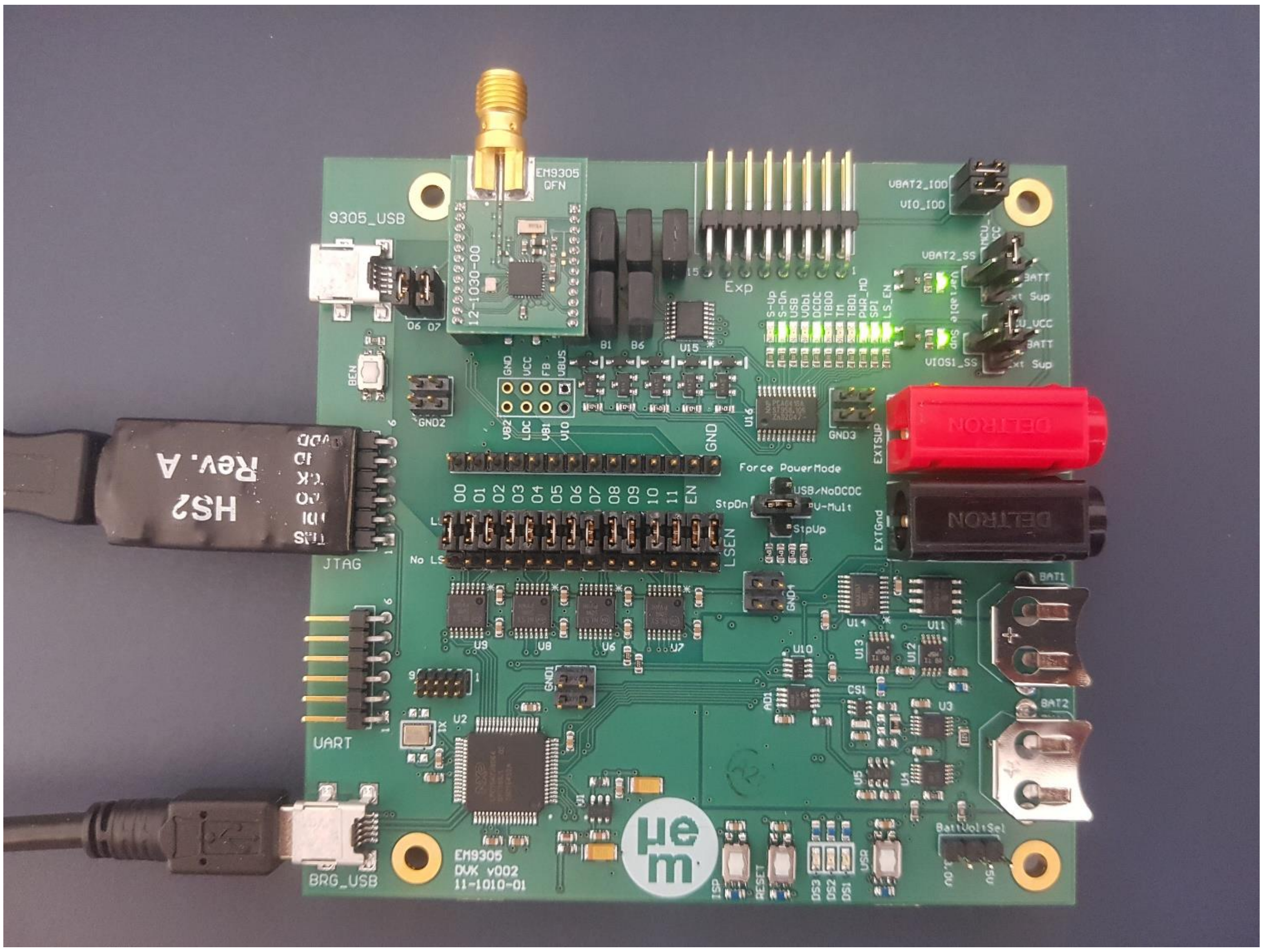


Figure 19-1: Attach the JTAG-HS2 debugging probe to the DVK

19.3.3 Start the Metaware debugger software (MDB)

The debugger software provided by Metaware is called "MDB Metaware Development Toolkit". If the software is not already installed on your system, follow the instructions in the SDK HTML documentation in the *Getting Started→Synopsys MetaWare Installation* section.

In addition, the Digilent Adept 2 software must be installed to interface with the HS2 debug probe. The software can be downloaded from https://digilent.com/reference/software/adept/start.

A batch script is provided in the SDK to facilitate the start of MDB Metaware debugger.

Go to the <SDK_INSTALL_PATH>\tools\debug folder:

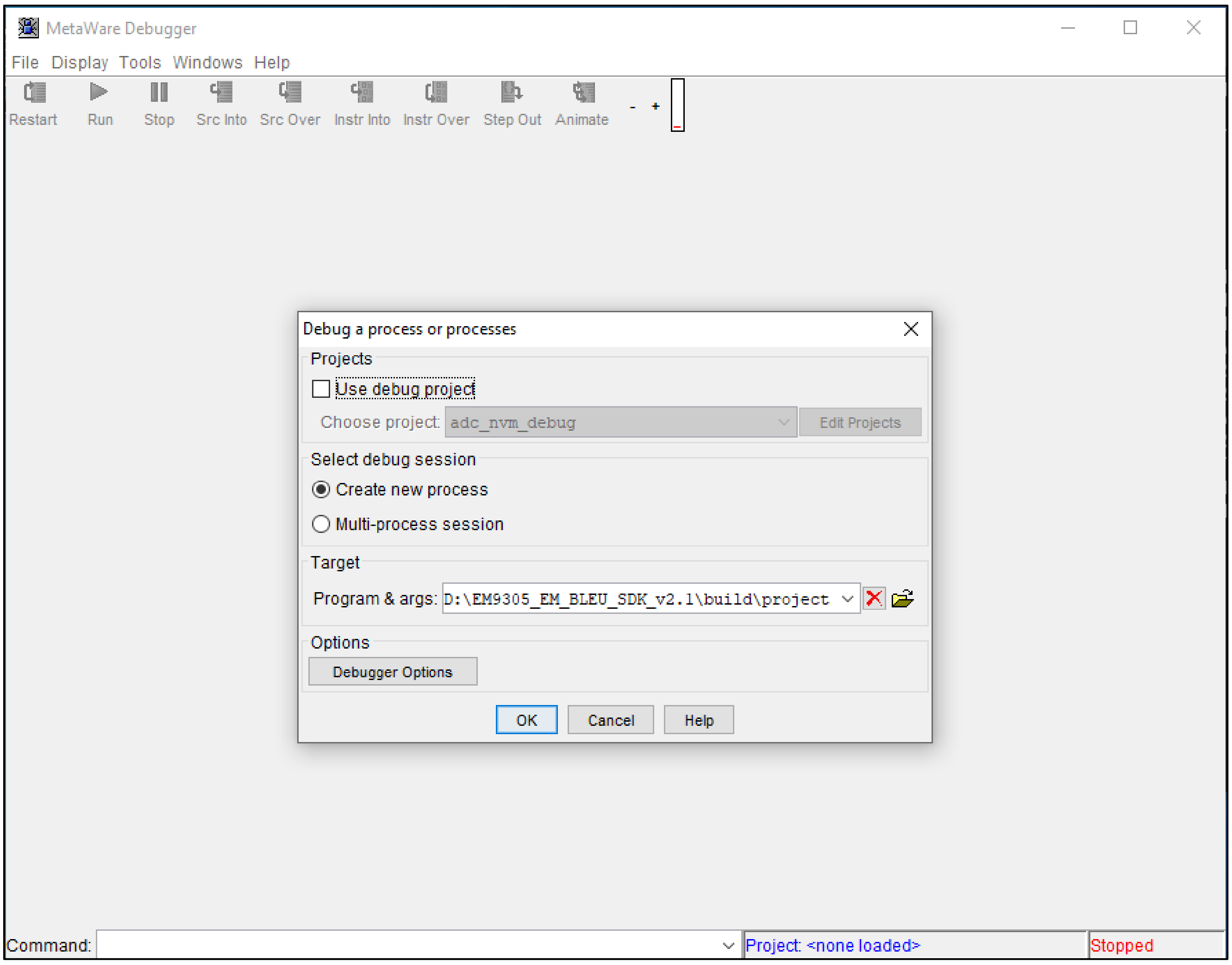
```
# cd <SDK_INSTALL_PATH>\tools\debug
```

Execute the provided batch script to start MDB with the target's ELF file:

```
# .\debug.bat <SDK_INSTALL_PATH>\build\projects\printf_example\printf_example_diXX.elf
```

Note: the path to the ELF file must be an absolute path.

The MetaWare debugger software is started. Press OK to debugging.


MetaWare Debugger
File Display Tools Windows Help
Restart Run Stop Src Into Src Over Instr Into Instr Over Step Out Animate

Debug a process or processes

Projects

Use debug project

Choose project: adc_nvm_debug

Select debug session

(●) Create new process

( ) Multi-process session

Target

Program & args: D:\EM9305_EM_BLEU_SDK_v2.1\build\project

Options

Debugger Options

OK Cancel Help

Command:

Project: <none loaded>

Stopped

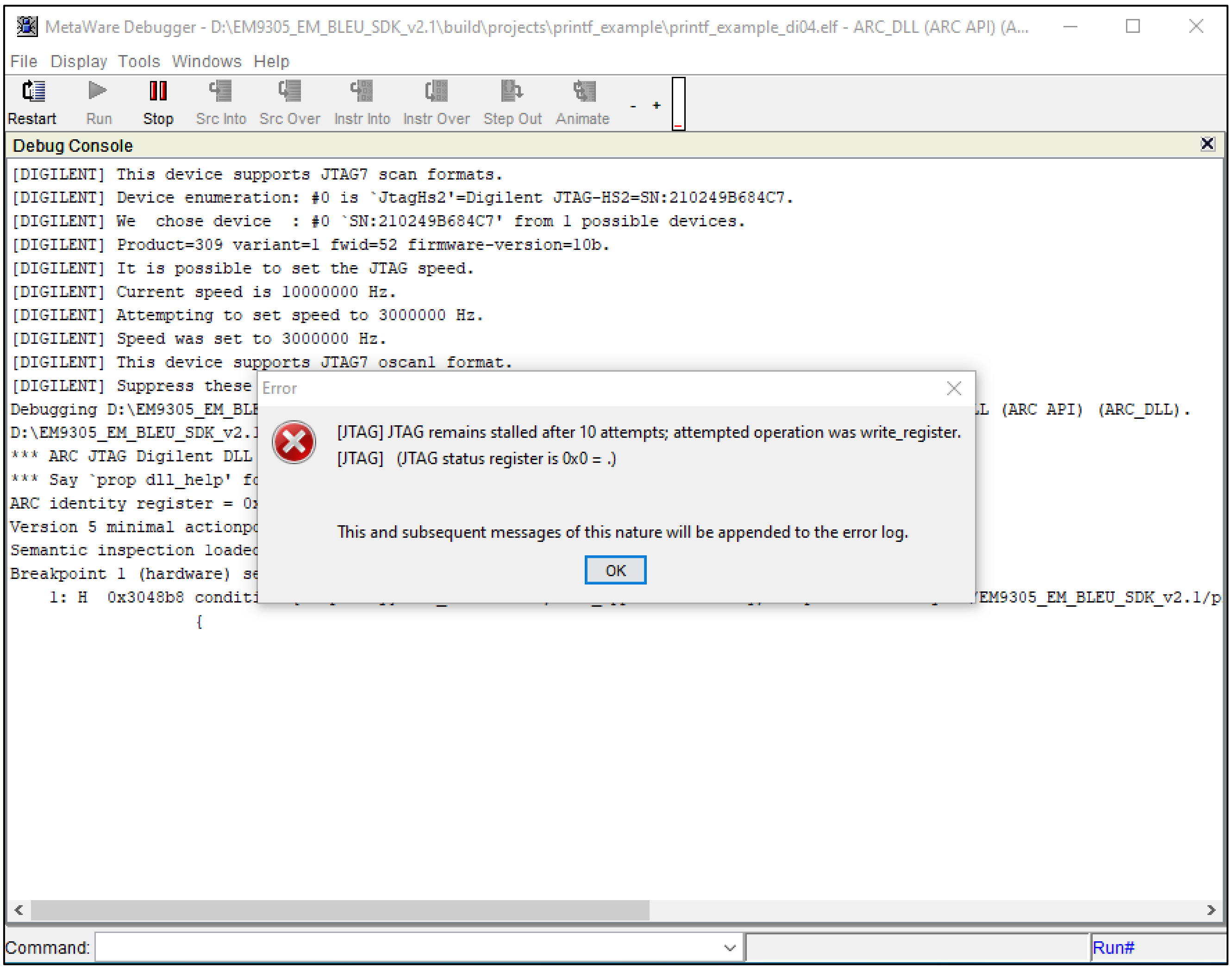
Figure 19-2: MDB MetaWare debugger step 1 – debug a process

Depending on the configuration of the three JTAG related lock bits (see §Table 13-1 and §Table 13-2) an error message may appear.

The different behaviour of MDB with respect to the JTAG-related lock bits is described below:

Lock bit name	PmlJtag2wEn / PmlJtag4wEn				
	Lock bit value	0/0	0/1	1/0	1/1
PmlJtagLock	0	Error message Can debug	No error message Can debug	No error message Can debug	No error message Can debug
	1	Error message Cannot debug	Error message Cannot debug	Error message Cannot debug	Error message Cannot debug

Table 19-2: MDB behaviour with respect to the JTAG-related lock bits


MetaWare Debugger - D:\EM9305_EM_BLEU_SDK_v2.1\build\projects\printf_example\printf_example_di04.elf - ARC_DLL (ARC API) (A...
File Display Tools Windows Help
Restart Run Stop Src Into Src Over Instr Into Instr Over Step Out Animate
Debug Console
Command:
Run#

```
[DIGILENT] This device supports JTAG7 scan formats.
[DIGILENT] Device enumeration: #0 is `JtagHs2'=Digilent JTAG-HS2=SN:210249B684C7.
[DIGILENT] We  chose device  : #0 `SN:210249B684C7' from 1 possible devices.
[DIGILENT] Product=309 variant=1 fwid=52 firmware-version=10b.
[DIGILENT] It is possible to set the JTAG speed.
[DIGILENT] Current speed is 10000000 Hz.
[DIGILENT] Attempting to set speed to 3000000 Hz.
[DIGILENT] Speed was set to 3000000 Hz.
[DIGILENT] This device supports JTAG7 oscan1 format.
[DIGILENT] Suppress these
Debugging D:\EM9305_EM_BLE                                                L (ARC API) (ARC_DLL).
D:\EM9305_EM_BLEU_SDK_v2.1
*** ARC JTAG Digilent DLL
*** Say `prop dll_help' fo
ARC identity register = 0x
Version 5 minimal actionpo
Semantic inspection loaded
Breakpoint 1 (hardware) se
    1: H  0x3048b8 conditi                                                EM9305_EM_BLEU_SDK_v2.1/p
                       {
```

Error

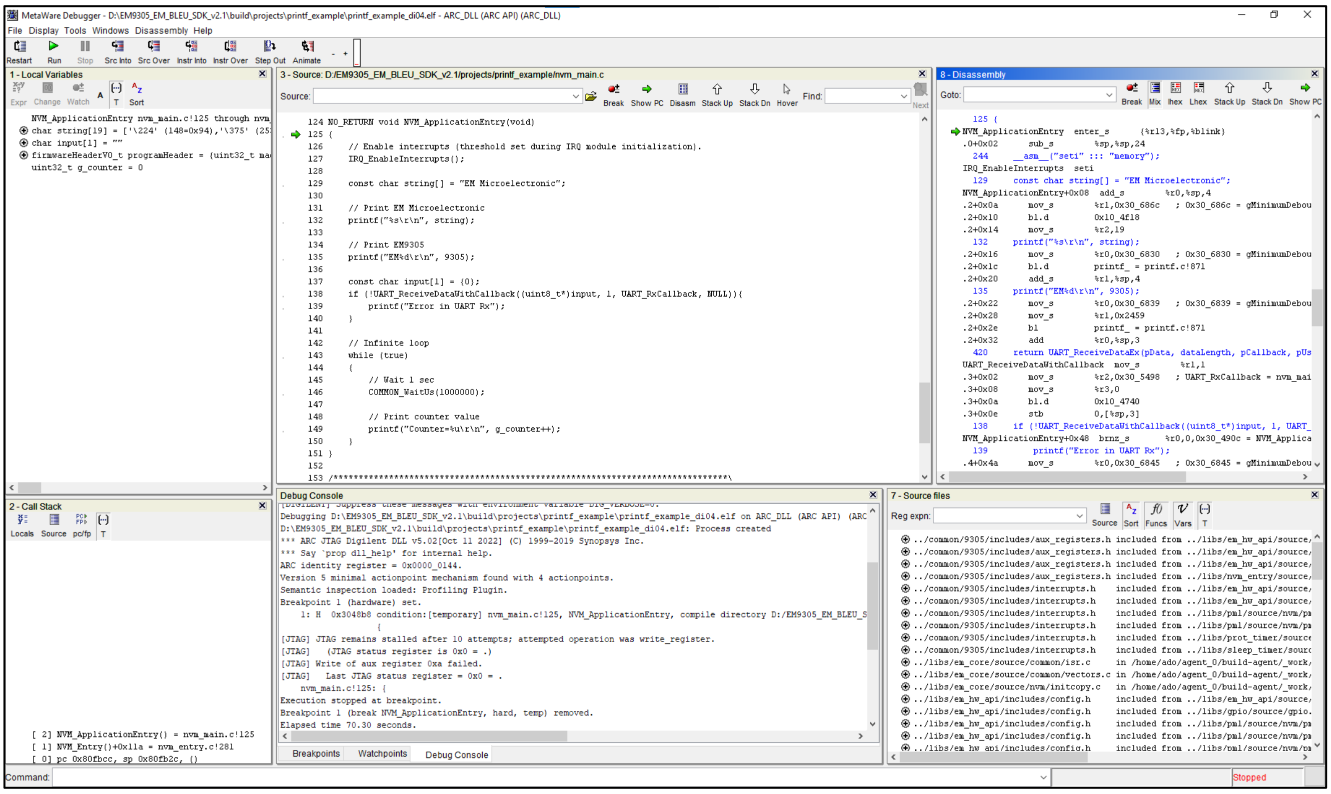
[JTAG] JTAG remains stalled after 10 attempts; attempted operation was write_register.
[JTAG] (JTAG status register is 0x0 = .)

This and subsequent messages of this nature will be appended to the error log.

OK

Figure 19-3: MDB MetaWare debugger step 2 – possible error message

Press OK if an error message appeared and, if MDB was able to start the debug session, the main window should appear with the code execution paused at the beginning of the `NVM_ApplicationEntry()` function.


MetaWare Debugger - D:\EM9305_EM_BLEU_SDK_v2.1\build\projects\printf_example\printf_example_di04.elf - ARC_DLL (ARC API) (ARC_DLL)
File Display Tools Windows Disassembly Help
Restart Run Stop Src Into Src Over Instr Into Instr Over Step Out Animate
1 - Local Variables
Expr Change Watch A T Sort
3 - Source: D:/EM9305_EM_BLEU_SDK_v2.1/projects/printf_example/nvm_main.c
Source: Break Show PC Disasm Stack Up Stack Dn Hover Find: Next
8 - Disassembly
Goto: Break Mix Ihex Lhex Stack Up Stack Dn Show PC
2 - Call Stack
Locals Source pc/fp T
Debug Console
Breakpoints Watchpoints Debug Console
7 - Source files
Reg expn: Source Sort Funcs Vars T
Command:
Stopped

```
NVM_ApplicationEntry nvm_main.c!125 through nvm_
⊕ char string[19] = ['\224' (148=0x94),'\375' (25
⊕ char input[1] = ""
⊕ firmwareHeaderV0_t programHeader = {uint32_t ma
uint32_t g_counter = 0
```

```
124 NO_RETURN void NVM_ApplicationEntry(void)
125 {
126     // Enable interrupts (threshold set during IRQ module initialization).
127     IRQ_EnableInterrupts();
128
129     const char string[] = "EM Microelectronic";
130
131     // Print EM Microelectronic
132     printf("%s\r\n", string);
133
134     // Print EM9305
135     printf("EM%d\r\n", 9305);
136
137     const char input[1] = {0};
138     if (!UART_ReceiveDataWithCallback((uint8_t*)input, 1, UART_RxCallback, NULL)){
139         printf("Error in UART Rx");
140     }
141
142     // Infinite loop
143     while (true)
144     {
145         // Wait 1 sec
146         COMMON_WaitUs(1000000);
147
148         // Print counter value
149         printf("Counter=%u\r\n", g_counter++);
150     }
151 }
152
153 /*****************************************************************************\
```

```
  125 {
NVM_ApplicationEntry  enter_s      {%r13,%fp,%blink}
.0+0x02      sub_s        %sp,%sp,24
  244      __asm__("seti" ::: "memory");
IRQ_EnableInterrupts  seti
  129      const char string[] = "EM Microelectronic";
NVM_ApplicationEntry+0x08  add_s       %r0,%sp,4
.2+0x0a      mov_s        %r1,0x30_686c   ; 0x30_686c = gMinimumDebou
.2+0x10      bl.d         0x10_4f18
.2+0x14      mov_s        %r2,19
  132      printf("%s\r\n", string);
.2+0x16      mov_s        %r0,0x30_6830   ; 0x30_6830 = gMinimumDebou
.2+0x1c      bl.d         printf_ = printf.c!871
.2+0x20      add_s        %r1,%sp,4
  135      printf("EM%d\r\n", 9305);
.2+0x22      mov_s        %r0,0x30_6839   ; 0x30_6839 = gMinimumDebou
.2+0x28      mov_s        %r1,0x2459
.2+0x2e      bl           printf_ = printf.c!871
.2+0x32      add          %r0,%sp,3
  420      return UART_ReceiveDataEx(pData, dataLength, pCallback, pUs
UART_ReceiveDataWithCallback  mov_s       %r1,1
.3+0x02      mov_s        %r2,0x30_5498   ; UART_RxCallback = nvm_mai
.3+0x08      mov_s        %r3,0
.3+0x0a      bl.d         0x10_4740
.3+0x0e      stb          0,[%sp,3]
  138      if (!UART_ReceiveDataWithCallback((uint8_t*)input, 1, UART_
NVM_ApplicationEntry+0x48  brnz_s      %r0,0,0x30_490c = NVM_Applica
  139          printf("Error in UART Rx");
.4+0x4a      mov_s        %r0,0x30_6845   ; 0x30_6845 = gMinimumDebou
```

```
[ 2] NVM_ApplicationEntry() = nvm_main.c!125
[ 1] NVM_Entry()+0x11a = nvm_entry.c!281
[ 0] pc 0x80fbcc, sp 0x80fb2c, ()
```

```
Debugging D:\EM9305_EM_BLEU_SDK_v2.1\build\projects\printf_example\printf_example_di04.elf on ARC_DLL (ARC API) (ARC
D:\EM9305_EM_BLEU_SDK_v2.1\build\projects\printf_example\printf_example_di04.elf: Process created
*** ARC JTAG Digilent DLL v5.02[Oct 11 2022] (C) 1999-2019 Synopsys Inc.
*** Say `prop dll_help' for internal help.
ARC identity register = 0x0000_0144.
Version 5 minimal actionpoint mechanism found with 4 actionpoints.
Semantic inspection loaded: Profiling Plugin.
Breakpoint 1 (hardware) set.
    1: H  0x3048b8 condition:[temporary] nvm_main.c!125, NVM_ApplicationEntry, compile directory D:/EM9305_EM_BLEU_S
                  {
[JTAG] JTAG remains stalled after 10 attempts; attempted operation was write_register.
[JTAG]   (JTAG status register is 0x0 = .)
[JTAG] Write of aux register 0xa failed.
[JTAG]   Last JTAG status register = 0x0 = .
    nvm_main.c!125: {
Execution stopped at breakpoint.
Breakpoint 1 (break NVM_ApplicationEntry, hard, temp) removed.
Elapsed time 70.30 seconds.
```

```
⊕ ../common/9305/includes/aux_registers.h included from ../libs/em_hw_api/source/
⊕ ../common/9305/includes/aux_registers.h included from ../libs/em_hw_api/source/
⊕ ../common/9305/includes/aux_registers.h included from ../libs/em_hw_api/source/
⊕ ../common/9305/includes/aux_registers.h included from ../libs/nvm_entry/source/
⊕ ../common/9305/includes/interrupts.h    included from ../libs/em_hw_api/source/
⊕ ../common/9305/includes/interrupts.h    included from ../libs/em_hw_api/source/
⊕ ../common/9305/includes/interrupts.h    included from ../libs/pml/source/nvm/pm
⊕ ../common/9305/includes/interrupts.h    included from ../libs/pml/source/nvm/pm
⊕ ../common/9305/includes/interrupts.h    included from ../libs/prot_timer/source
⊕ ../common/9305/includes/interrupts.h    included from ../libs/sleep_timer/sourc
⊕ ../libs/em_core/source/common/isr.c     in /home/ado/agent_0/build-agent/_work/
⊕ ../libs/em_core/source/common/vectors.c in /home/ado/agent_0/build-agent/_work/
⊕ ../libs/em_core/source/nvm/initcopy.c   in /home/ado/agent_0/build-agent/_work/
⊕ ../libs/em_hw_api/includes/config.h     included from ../libs/em_hw_api/source/
⊕ ../libs/em_hw_api/includes/config.h     included from ../libs/gpio/source/gpio.
⊕ ../libs/em_hw_api/includes/config.h     included from ../libs/pml/source/nvm/pm
⊕ ../libs/em_hw_api/includes/config.h     included from ../libs/pml/source/nvm/pm
⊕ ../libs/em_hw_api/includes/config.h     included from ../libs/pml/source/nvm/pm
```

Figure 19-4: MDB MetaWare debugger step 3 – main window

MDB has all the standard features of a debugger, including breakpoints, watchpoints, disassembly code viewer, call stack viewer, etc. For further explanation on these specific topics, the MDB user manual can be found under the *Help→Online Documentation* tab.

The user should be mindful that triggering breakpoints in the code will halt the chip's internal timer, potentially impacting the operation of Bluetooth and/or the application. Consequently, resuming the program after a pause may not guarantee correct behaviour. However, while the program is stopped, it allows reading static variables and registers, rendering it an excellent tool for debugging applications.

19.4 GPIO DEBUGGING

In this chapter, we will explore the use of GPIO (General Purpose Input/Output) pins as a powerful method to debug embedded software.

Advantages of GPIO Debugging:

- Real-time Insights: GPIO debugging provides real-time insights into your application's behaviour on the target hardware, allowing you to monitor state changes, function execution, and small variable values during runtime.
- Hardware Integration: Leveraging GPIO pins for debugging requires minimal additional hardware, making it a cost-effective and readily available solution for embedded developers.
- Custom Indicators: You can create custom debugging indicators using GPIO pins to signal specific events, making it easier to understand and analyse the execution flow of your code.

19.4.1 Setting up GPIO for debugging

A GPIO is configured as an output in the following way:

```
// Configure GPIO 8 as output
GPIO_SetOutputPinFunction(8, GPIO_PIN_FUNC_OUT_GPIO);
GPIO_DisablePullDown(8);
```

```
GPIO_DisablePullUp(8);
GPIO_DisableInput(8);
GPIO_SetLow(8);
GPIO_EnableOutput(8);
```

For example, a GPIO can be configured as an output in the `NVM_ApplicationEntry()` function only on power-up, meaning that the GPIO configuration is done only once and not on every wake-up from sleep. This way the GPIO is available for debugging after the GPIO module has been restored, i.e. after the function `NVM_ConfigModules()` has been called by the boot/wake-up code.

19.4.2 Use of GPIO for debugging

Once set up, the GPIOs can be used in a variety of ways to debug application code.

A pulse can then be generated on a GPIO in the following way:

```
// Short positive pulse on GPIO 8
GPIO_SetHigh(8);
GPIO_SetLow(8);
```

A function execution time can be measured in the following way:

```
// Measure the execution time of a function
GPIO_SetHigh(8);
funcToMeasureExecTime();
GPIO_SetLow(8);
```

Note: Be aware that an interrupt may be triggered during the execution of the function, which will increase the measured execution time.

A small variable can be displayed in the following way:

```
// Display a variable with a small value, i.e. less than 10
for (int i = 0; i < varToDisplay; i++)
{
    GPIO_SetHigh(8);
    GPIO_SetLow(8);
}
```

A logic analyser can be used to monitor the state of the GPIOs to check for the patterns shown above.

There are many other ways to use GPIOs for debugging and the examples above are just some of them, the application developer is free to adapt them to their own needs.

19.5 PRINTF DEBUGGING

Printf debugging is a powerful technique used by embedded software engineers to gain insights into the behavior of their code during runtime. By printing relevant information to the console or log, developers can analyze the flow of execution, monitor variable values, and detect potential errors. In this chapter, we will explore the principles and best practices of printf debugging and how it can be effectively utilized in our SDK.

19.5.1 Setting up printf debugging

Before you can start using printf debugging, you need to set up the necessary infrastructure to capture and display the debug output. This section will guide you through the steps required to enable printf debugging effectively.

- Add the necessary libraries to the `CMakeLists.txt` project file, i.e. *uart_dma* and *printf*

```
SET(${PROJECT_NAME}_LIBS
     ${NVM_COMMON_LIBS}
# PROJECT SPECIFIC LIBRARIES
```

```
     uart_dma
     printf
)
```

- Include the UART header file and declare the UART configuration variable in the main application file

```
#include "uart.h"

// UART Configuration structure
extern volatile UART_Configuration_t gUART_Config;
```

- Register and enable the UART module in the `NVM_ConfigModules()` function

```
// Register the UART module.
UART_RegisterModule();

// Enable UART.
gUART_Config.enabled = true;
```

- Configure the GPIO for UART Tx function in the `NVM_ApplicationEntry()` function

```
// Configure GPIO 7 as UART Tx
GPIO_DisableInput(7);
GPIO_DisablePullDown(7);
GPIO_EnablePullUp(7);
GPIO_SetOutputPinFunction(7, GPIO_PIN_FUNC_OUT_UART_TXD);
GPIO_EnableOutput(7);

// Ensure that UART Rx is not mapped to a GPIO
GPIO_SetInputFunctionPin(GPIO_PIN_FUNC_IN_UART_RXD, GPIO_FUNCTION_NOT_MAPPED);
```

- Finally, to use `printf()`, include `printf.h` in the file and call it as needed

```
printf("Hello World\r\n");
```

19.5.2 Printf debugging considerations

Here is a list of good practices that can help debugging your application.

- Selective Debugging: Instead of flooding your code with `printf` statements, use them selectively in specific areas that are relevant to your debugging goals. Target critical sections, loops, or problematic functions.
- Avoid Side Effects: Be cautious when using `printf` statements within loops or time-sensitive code. The I/O operations can cause delays and affect the timing of your program, potentially masking or introducing new bugs.
- Timestamps: When dealing with real-time systems or time-sensitive operations, consider adding timestamps to your debug messages. This will assist in understanding the timing of events and potential synchronization issues.
- The `printf` implementation is not reentrant, meaning that it is not designed to be safely called simultaneously from multiple tasks.

The `printf` debugging is a relatively heavy operation involving I/O. When used extensively or in critical sections of high-priority tasks, `printf` can disrupt the scheduling and timing of lower-priority tasks, affecting the overall system behaviour.

19.6 ASSERTIONS DEBUGGING

When using QP/C, there is a possibility to add assertion statements at different locations within the application code. It is then possible to write a user defined assertion callback function that is called whenever an assertion is triggered.

The QP/C framework comes along with two assertion macro definitions that can be used by the application:

- `Q_ASSERT(<test>)`
- `Q_ASSERT(id, <test>)`

The usage of these two macros depends on the goal to be achieved. However, whatever is the used macro, the callback function has the same prototype definition which is the following:

```
void Q_onAssertExt(const char* file, int_t data)
{
      // your code here
}
```

The `file` parameter is the name of the source file containing the assertion, while the `data` parameter meaning depends on whether the assertion has been triggered by the `Q_ASSERT(<test>)` macro or by the `Q_ASSERT(id, <test>)` macro.

The Table 9-3 gives the definition of the `data` parameter according to the used assertion macro.

Assertion macro used	’data’ parameter definition
Q_ASSERT(<test>)	line number of the macro in the file ’file’
Q_ASSERT(id, <test>)	’id’ value specified in the macro

Table 19-3: assertion ’data’ parameter definition

As it has been mentioned above, the assertion handling can be combined with the `printf` debugging to send debugging traces over the UART to a host computer.

Here is an example of such a combination that offers some flexibility in the debugging process.

```
uint8_t table_lookup(uint8_t table, uint8_t size)
{
      for(uint8_t i = 0; i < size; i++)
      {
            // Access the table with index i
      }
}

uint8_t TheTable[MAX_SIZE];
uint8_t value = table_lookup(TheTable, X);
```

This example shows a function which looks up for values into a table. This table is of maximum `MAX_SIZE` size and any access to this table shall not exceed this size. However, using this table can be done with a length lower than the maximum size. This is what the size parameter contains and it is expected that the following condition is met under nominal conditions:

```
size < MAX_SIZE
```

However, the `size` parameter can be specified with a value that exceeds the `MAX_SIZE` value.

In case a value greater than `MAX_SIZE` is passed to the function, then an out of bound access will be done and the software might crash.

One way to debug use case is to add an assertion at the beginning of the function, like in this code snippet:

```
uint8_t table_lookup(uint8_t table, uint8_t size)
{
      Q_ASSERT(size < MAX_SIZE);
      for(uint8_t i = 0; i < size; i++)
      {
            // Access the table with index i
      }
      …
}
```

Then a user defined callback function is added:

```
void Q_onAssertExt(const char* file, int_t line)
{
      printf("Assertion failure on line %d in file '%s'.\r\n", line, file);
}
```

Using the `printf` statement implies configuring the UART and integrating the `printf` library. See §19.5 for more details on how to proceed.

Then, if the test `size < MAX_SIZE` fails, then the terminal windows on the host computer will display a message similar to this one:

```
Assertion failure on line 234 in file 'my_app.c'.
```

And then, it becomes obvious that out-of-bound access has been attempted.

Another way of debugging is to use the `Q_ASSERT_ID()` macro and to enforce it with a `false` statement, like in this code:

```
Q_ASSERT_ID(1234, false);
```

This will force the assertion and the user defined callback will be executed. In this case, the `line` parameter will contain the value 1234 (which is in fact the ID passed to the macro). It can then be used to trigger specific actions.

An example of assertion callback is given in the following code snippet:

```
void Q_onAssertExt(const char* file, int_t parameter)
```

```
{
    switch(parameter) :
        case 1234 :
            // Do a specific processing
            break ;
        case XXXX :
            // Do another processing
            break ;
        default :
            // Do the default processing
            break ;
}
```

In this prototype, the parameter named `parameter` (instead of `line` in the previous example) contains the `ID` that has been specified in the assertion statement. It can be used to execute different code blocks like activating some GPIOs, or sending a message over UART or SPI, and so on.

The QP/C framework actively uses assertion statements in its code. Thus, setting a user defined assertions callback will also capture assertion failures in QP/C execution.

APPENDIX 1 GLOSSARY

The Table 19-4 is a glossary for the acronyms and specific terms used along this document.

Acronym	Description
ADC	Analog to Digital Converter
AICPC	Audio Input Control Profile Client
AICPS	Audio Input Control Profile Server
AICS	Audio Input Control Service
ASIC	Analog Specialized Integrated Circuit
API	Application Programming Interface
BAPBA	Basic Audio Profile Broadcast Assistant
Bluetooth LE	Bluetooth Low Energy
BSP	Board Support Package
CSP	Chip-Scale Package
DCCM	Data Closely Coupled Memory
DI	Design Iteration
DMA	Direct Memory Access
DSP	Digital Signal Processor
ELF	Executable and Linkable Format
GAP	Generic Access Profile
GATT	Generic Attribute Profile
GPIO	General Purpose Input Output
HF	High Frequency
HSM	QP/C Hierarchical State Machine
I^{2}C	Inter Integrated Circuits
ICCM	Instruction Closely Coupled Memory
IRAM	Instruction RAM (RAM from where the ARC CPU fetches its instructions)
LDO	Low Drop Out (voltage regulator)
MCU	Micro-Controller Unit
NVM	Non Volatile Memory
PML	Power Management Logic
POR	Power On Reset (sometimes called cold reset)
QFN	Quad Flat No-lead
ROM	Read Only Memory
SDK	Software Development Kit
SoC	System on Chip
SPI	Serial Peripheral Interface
TBI	To Be Issued

UART	Universal Asynchronous Receiver and Transmitter
UUID	Universally Unique Identifier
WSF	Wireless Software Foundations

Table 19-4: Glossary

APPENDIX 2 COMMON LIBRARIES

The most commonly used libraries are gathered under the `NVM_COMMON_LIBS` variable in the `CMakeLists.txt` file. This encompasses the list given in Table 19-5.

Library	Description
adc	Analog to digital support driver
em_core	The EM core library itself.
em_hw_api	This module contains all hardware definitions (registers mapping, bits definition, bits manipulation helper functions, …).
em_system	This is the EM commands system by which a set of HCI commands can be included in the application. For these commands, the application will process the related actions.
gpio	General purpose I/O library management.
header_access	This modules contains an API the end user can include to access to all fields of the header that is built along with an application.
Metaware	The Metaware libraries provided along with the toolchain.
nvm_bootloader	Contains functions to be used to manipulate the firmware images header by adding an abstraction layer.
nvm_entry	The actual entry point of an application mapped into NVM. This is the address at which the NVM Bootloader will jump. Not to be confused by the end user application main entry point which is called later on by the NVM_Entry.
pml	Power management
prot_timer	Protocol timer for Bluetooth LE low level protocol timings
radio	Radio management.
rc_calib	Calibration of the reference clock.
sleep_manager	Sleep manager in case the sleep feature has to be fully controlled by the end user application.
sleep_timer	Sleep timers management.
transport	The transport layer for communicating with the device through SPI or UART.

Table 19-5: Common libraries list and description

APPENDIX 3 EM INFORMATION PAGE

The Table 19-6 lists all the parameters stored in the EM information page (otherwise called page 3) that are filled in at the EM factory before being shipped to any customer. This page is locked so it cannot be modified or erased.

It is given here to provide an overview of its structure but without giving any value which at the end depends on the device itself (e.g., trimming values) and the end customer requirements (e.g., lock bits configuration).

Address Hex	Address Dec	Size [B]	Description
0x8000	32768	0	
0x7FC0	32704	64	**Reserved**
0x7F88	32648	56	Test History
0x7F80	32640	8	"w2Pass\0\0"
0x7F10	32528	112	**Reserved**
0x7F0C	32524	4	EM authentication required flag (1's complement)
0x7F08	32520	4	EM authentication required flag
0x7F04	32516	4	NVM programmed flag (page3 programmed) 1's complement
0x7F00	32512	4	NVM programmed flag (page3 programmed)
0x7E08	32264	248	**Reserved**
0x7DC8	32200	64	EM Authentication Public Key (Diversified Key)
0x7D88	32136	64	EM Patch Signature Public Key (fixed Key)
0x7D84	32132	4	CRC
0x7D80	32128	4	Length
0x7D38	32056	72	**Reserved**
0x7D34	32052	4	Master lock register 6 - data
0x7D30	32048	4	Master lock register 6 - address
0x7D2C	32044	4	Lock register 5 - data
0x7D28	32040	4	Lock register 5 - address
0x7D24	32036	4	Lock register 4 - data
0x7D20	32032	4	Lock register 4 - address
0x7D1C	32028	4	Lock register 3 - data
0x7D18	32024	4	Lock register 3 - address
0x7D14	32020	4	Lock register 2 - data
0x7D10	32016	4	Lock register 2 - address
0x7D0C	32012	4	Lock register 1 - data
0x7D08	32008	4	Lock register 1 - address
0x7D04	32004	4	CRC
0x7D00	32000	4	Length
0x7CB0	31920	80	**Reserved - migh be used for additional 10 trimming registers**
0x7CAC	31916	4	Trimming register 5 - data
0x7CA8	31912	4	Trimming register 5 - address (e.g. `REG_RF_XO_DEBUG_DIG`)

0x7CA4	31908	4	Trimming register 4 - data
0x7CA0	31904	4	Trimming register 4 - address (e.g. `REG_RF_XO_SEQ_DIG`)
0x7C9C	31900	4	Trimming register 3 - data
0x7C98	31896	4	Trimming register 3 - address (e.g. `RegNvmTime`)
0x7C94	31892	4	Trimming register 2 - data
0x7C90	31888	4	Trimming register 2 - address (e.g. `RegNvmRedunCfg`)
0x7C8C	31884	4	Trimming register 1 - data
0x7C88	31880	4	Trimming register 1 - address (e.g. `RegPmlTrim`)
0x7C84	31876	4	CRC
0x7C80	31872	4	Length
0x7C4C	31820	52	Product Information (incl. Device ID and Wafer Information (depends on the version)
0x7C48	31816	4	Product Information Structure Version
0x7C44	31812	4	CRC
0x7C40	31808	4	Length
0x7C10	31760	48	**Reserved**
0x7C08	31752	8	MAC address
0x7C04	31748	4	CRC
0x7C00	31744	4	Length
0x7BC4	31684	60	**Reserved**
0x7BBC	31676	8	ADC Calibration , ADC Source = 0x3, ADC SelVhi = 0x1
0x7BB8	31672	4	CRC
0x7BB4	31668	4	Length
0x7BAC	31660	8	ADC Calibration , ADC Source = 0x3, ADC SelVhi = 0x0
0x7BA8	31656	4	CRC
0x7BA4	31652	4	Length
0x7B9C	31644	8	ADC Calibration , ADC Source = 0x2
0x7B98	31640	4	CRC
0x7B94	31636	4	Length
0x7B8C	31628	8	ADC Calibration , ADC Source = 0x1
0x7B88	31624	4	CRC
0x7B84	31620	4	Length
0x7784	30596	1024	ADC Calibration , ADC Source = 0x0, ADC Range = 0x2
0x7780	30592	4	CRC
0x777C	30588	4	Length
0x737C	29564	1024	ADC Calibration , ADC Source = 0x0, ADC Range = 0x1
0x7378	29560	4	CRC
0x7374	29556	4	Length
0x6F74	28532	1024	ADC Calibration , ADC Source = 0x0, ADC Range = 0x0
0x6F70	28528	4	CRC
0x6F6C	28524	4	Length
0x6F64	28516	8	Temperature Indicator (TI) Calibration data, Coefficient + Offset
0x6F60	28512	4	CRC

0x6F5C	28508	4	Length
0x6000	24576	3932	**Reserved**

Table 19-6: EM reserved information pages

APPENDIX 4 USER INFORMATION PAGE

The Table 19-7 lists the details of the content of the user information page. According to its name, this page is not locked and is up to the end user to configure it at will.

As already explained for the EM information page in Appendix 3 , the list of paired values (register address; register data) in the block ranging from address `0x5D00` to `0x5D7F` are an example. It is up to the end user to fill these blocks with the list of registers for which lock is needed.

Address Hex	Address Dec	Size [B]	Description
0x6000	24576	0	
0x5F10	24336	240	**Reserved**
0x5F0C	24332	4	USER authentication required flag 1's complement value
0x5F08	24328	4	USER authentication required flag
0x5F04	24324	4	NVM read lock 1's complement value
0x5F00	24320	4	NVM read lock
0x5DC8	24008	312	**Reserved**
0x5D88	23944	64	USER Patch Signature Public Key (fixed key)
0x5D84	23940	4	CRC
0x5D80	23936	4	Length
0x5D38	23864	72	**Reserved - migh be used for additional 9 lock registers**
0x5D34	23860	4	Master lock register 6 - data
0x5D30	23856	4	Master lock register 6 - address
0x5D2C	23852	4	Lock register 5 - data
0x5D28	23848	4	Lock register 5 - address
0x5D24	23844	4	Lock register 4 - data
0x5D20	23840	4	Lock register 4 - address
0x5D1C	23836	4	Lock register 3 - data
0x5D18	23832	4	Lock register 3 - address
0x5D14	23828	4	Lock register 2 - data
0x5D10	23824	4	Lock register 2 - address
0x5D0C	23820	4	Lock register 1 - data
0x5D08	23816	4	Lock register 1 - address
0x5D04	23812	4	CRC
0x5D00	23808	4	Length
0x5C98	23704	104	**Reserved - migh be used for additional 13 trimming registers**
0x5C94	23700	4	Trimming register 2 - data
0x5C90	23696	4	Trimming register 2 - address (e.g. `REG_RF_XO_DEBUG_DIG`)
0x5C8C	23692	4	Trimming register 1 - data
0x5C88	23688	4	Trimming register 1 - address (e.g. `REG_RF_XO_SEQ_DIG`)
0x5C84	23684	4	CRC
0x5C80	23680	4	Length
0x5C10	23568	112	**Reserved**
0x5C08	23560	8	MAC address

0x5C04	23556	4	CRC
0x5C00	23552	4	Length
0x5BFC	23548	4	PML Configuration
0x5BF8	23544	4	CRC
0x5BF4	23540	4	Length
0x5BB8	23480	60	**Reserved**
0x5BB0	23472	8	ADC Calibration, ADC Source = 0x3, ADC SelVhi = 0x1
0x5BAC	23468	4	CRC
0x5BA8	23464	4	Length
0x5BA0	23456	8	ADC Calibration, ADC Source = 0x3, ADC SelVhi = 0x0
0x5B9C	23452	4	CRC
0x5B98	23448	4	Length
0x5B90	23440	8	ADC Calibration, ADC Source = 0x2
0x5B8C	23436	4	CRC
0x5B88	23432	4	Length
0x5B80	23424	8	ADC Calibration, ADC Source = 0x1
0x5B7C	23420	4	CRC
0x5B78	23416	4	Length
0x5778	22392	1024	ADC Calibration, ADC Source = 0x0, ADC Range = 0x2
0x5774	22388	4	CRC
0x5770	22384	4	Length
0x5370	21360	1024	ADC Calibration, ADC Source = 0x0, ADC Range = 0x1
0x536C	21356	4	CRC
0x5368	21352	4	Length
0x4F68	20328	1024	ADC Calibration, ADC Source = 0x0, ADC Range = 0x0
0x4F64	20324	4	CRC
0x4F60	20320	4	Length
0x4F58	20312	8	Temperature Indicator (TI) Calibration Data : Coefficient + Offset
0x4F54	20308	4	CRC
0x4F50	20304	4	Length
0x4000	16384	3920	**Reserved**

Table 19-7: User information pages

If the goal is to lock all the lock bits (no more modification can be done), then the master lock bit has to be set. In such a case, the last pair of [address; data] of the block shall be the Master lock bits register address with the corresponding configuration data value.

In this example, the master lock pair is the last one of the block with the Master lock bit register address at `0x5D30` and at `0x5D34` for the value to be written into the register. It will then be the last lock bits configuration applied.

APPENDIX 5 CONFIGURATION MODE SUPPORTED COMMANDS

The Table 19-9 list all the commands the configuration mode actually supports.

Each command is given with its operational code (OpCode) and the required authentication level. These levels are described in Table 19-8.

Level	Description
Not required	the command is executed w/o any authentication process requested
User	the command is executed only if a successful user authentication has been done previously
EM	the command is executed only if a successful EM authentication has been done previously

Table 19-8: Commands authentication levels

Note that the EM authentication is the highest level. This means that once an EM authentication is completed, all commands are executed whatever is the required authentication level. This is the highest privileged level.

Command	OpCode	Authentication	Description
Reset	0x0C03	Not required	Reset the notification system (TBC)
Read product information	0xFC01	Not required	Read device and wafer information
CPU reset	0xFC42	Not required	Trigger a CPU reset (= SW reset)
Set clock source	0xFC45	User	Select the clock source between RC or XTAL.
Set HF clock frequency	0xFC46	User	Set the core CPU internal clock frequency between 24MHz and 48MHz.
Get memory usage	0xFC47	User	Return the actual persistent and non persistent memory pool usage.
Set power mode	0xFC48	User	Request a new sleep mode.
Execute JLI function	0xFC4B	EM	Deprecated - Do not use. Still supported for backward compatibility but will be removed in the future.
Execute function	0xFC4C	EM	Execute a function which address in memory is specified, with arguments. The returned value is sent back to the caller.
Jump to function	0xFC4D	EM	Execute a function which address in memory is specified. Unlink the ‘Execute function’ no value is returned.
Calculate CRC32	0xFC4E	User	Calculate the CRC32 value in the specified range.
Leave configuration mode	0xFC50	Not required	Leave the configuration mode to switch to another mode (EM-Core, end user bootloader, end user application).
Read MAC address	0xFC52	Not required	Return the MAC address.
EM get challenge	0xFC81	Not required	Request the device to send a challenge as a prerequisite to the authentication (see [1] for details)

EM authenticate	0xFC82	Not required	Execute an EM authentication once the challenge has been completed. This can only be done within EM key management infrastructure.
User write private key	0xFC83	User	Let the user write its own private key into the key container.
User get challenge	0xFC84	Not required	Request the device to send a challenge as a prerequisite to the authentication (see [1] for details)
User authenticate	0xFC85	Not required	Execute a user authentication once the challenge has been completed (see [1] for details).
EM read public key	0xFC86	Not required	Read the public key stored in EM information page.
Read at address	0xFD01	User	Read data starting from the specified address.
Read continue	0xFD02	User	Keep reading data after the last read block (no need to specify again the start address)
Write at address	0xFD03	User	Write data starting from the specified address.
Write continue	0xFD04	User	Keep writing data after the last written block (no need to specify again the start address).
NVM erase full	0xFD05	EM	Erase the whole NVM (including information pages)
NVM erase main	0xFD06	User	Erase the NVM content except the information pages.
NVM erage page	0xFD07	User	Erase one single specified page in the main area, or one specified page in the information pages area.
NVM get lock page	0xFD08	User	Get the lock flag of the specified page.
NVM power control	0xFD0A	EM	Controls the NVM power line.
Write at address without response	0xFD0C	User	Write data using without waiting for response from the device (this speeds up the process).
Write AUX register	0xFD41	EM	Read the content of an auxiliary register.
Read AUX register	0xFD42	EM	Write the content of an auxiliary register.

Table 19-9: List of commands supported by configuration mode

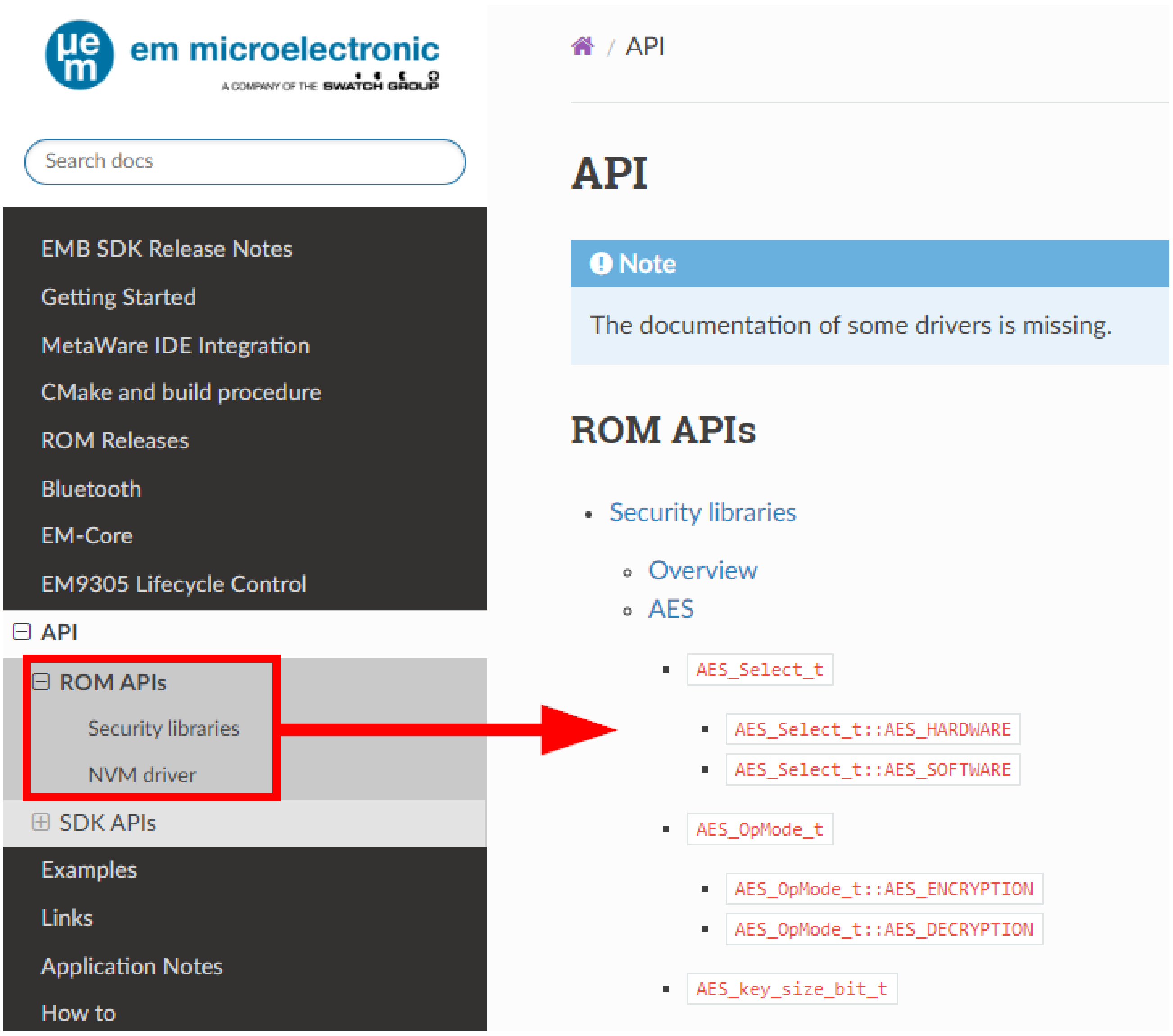
APPENDIX 6 ROM V3.0 CONTENT

The ROM v3.0 contains a set of functions that an end user application can use. These functions are related to the security libraries and the NVM access driver.

The SDK's documentation ROM APIs chapter lists all the functions along with parameters type used that an end user application can call.

Refer to this list for a more detailed description of each function.

The Figure 19-5 shows a snapshot of the part of the SDK documentation. For further detailed information, it is recommended to dive in this documentation.



API

Note

The documentation of some drivers is missing.

ROM APIs

- Security libraries
  - Overview
  - AES
    - `AES_Select_t`
      - `AES_Select_t::AES_HARDWARE`
      - `AES_Select_t::AES_SOFTWARE`
    - `AES_OpMode_t`
      - `AES_OpMode_t::AES_ENCRYPTION`
      - `AES_OpMode_t::AES_DECRYPTION`
    - `AES_key_size_bit_t`

Figure 19-5: ROM functions API listed in the SDK

For the link process, the SDK provides one ROM symbols file for each ROM version.

As an example, the Figure 19-6 shows the location of the `rom.sym` file containing the ROM v3.0 symbols.

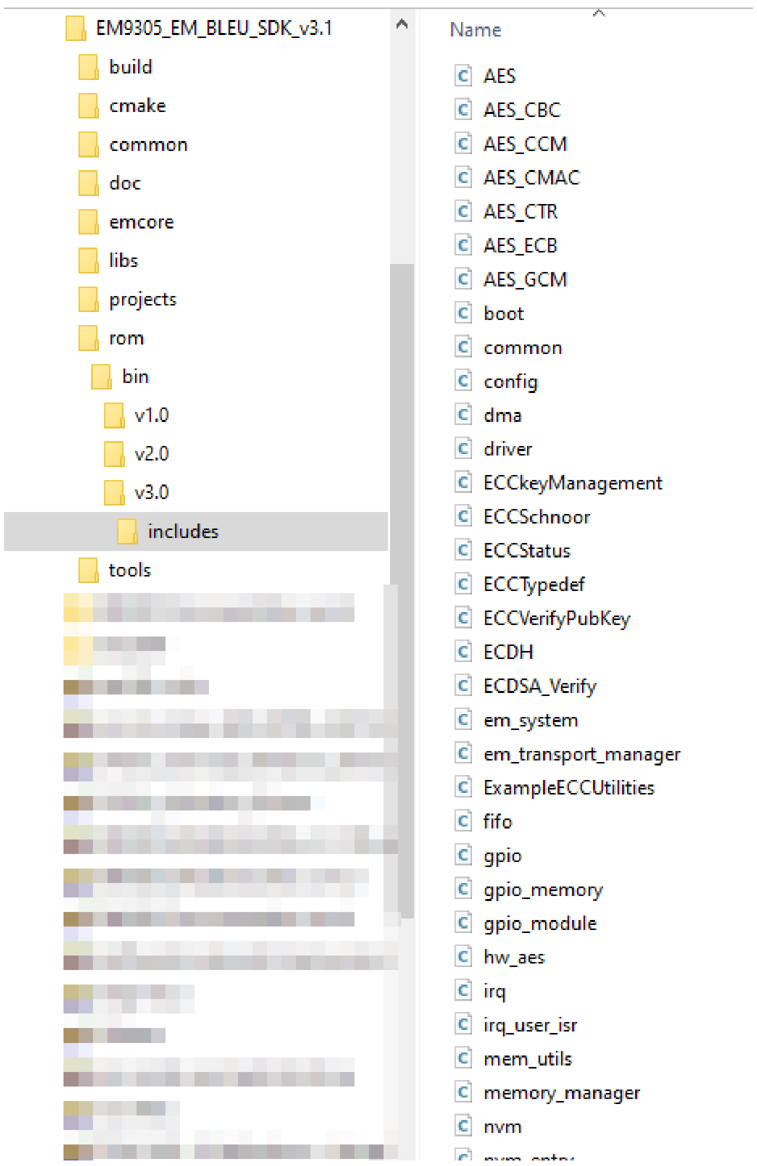


Figure 19-6: ROM symbol files

The `rom.sym` file is used during the link process when all objects and archive files are linked together to build the final executable.

The “includes” folder contains all header files that are needed for the compilation process.

APPENDIX 7 AVAILABLE RESOURCES

This appendix lists the hardware resource accessible by the software that are available to the end user application. This list is given in Table 19-10.

Such resources are not used by any of the SDK's libraries or drivers so they can freely be used by the end user. It can somehow happen that even if some resources are already used by the SDK, they are still available for the end user with some limitations.

When it is said that a resource is used by the SDK, it means that one or more libraries or drivers use this resource.

Resource	Description	Used by SDK?	Limitations?
Timer 0	The CPU ARC timer 0 accessible with the auxiliary registers (AUX). Timer 0 is connected to IRQ #0. A user defined handler can be registered using the following function: `IRQ_SetIRQUserTimerARCTimer0(void* pHandler);`	No	No
Timer 1	The CPU ARC timer 1 accessible with the auxiliary registers (AUX). Timer 1 is connected to IRQ #1. A user defined handler can be registered using the following function: `IRQ_SetIRQUserTimerARCTimer1(void* pHandler);`	No	No

Table 19-10: Free device resources

Last page

(this obviously means that this is the end of the story)