Getting started with STM32CubeWL for STM32WL Series

Introduction

STM32Cube is an STMicroelectronics original initiative to significantly improve developer productivity by reducing development effort, time and cost. STM32Cube covers the whole STM32 portfolio.

STM32Cube includes:

- A set of user-friendly software development tools to cover project development from the design to the production, among which:
  - STM32CubeMX, a graphical software configuration tool that allows the automatic generation of C initialization code using graphical wizards.
  - STM32CubeProgrammer (STM32CubeProg), a programming tool available with both graphical and command-line interfaces.
  - STM32CubeMonitor-Power (STM32CubeMonPwr), a monitoring tool to measure and to help optimizing the MCU power consumption.
  - STM32CubeMonitor, a versatile monitoring tool with a dedicated add-on to perform RF tests with STM32WL (dynamic packet transmission/reception, PER measurements) with a graphical representation of RF performance over time.
- A comprehensive embedded software platform, delivered per series (such as STM32CubeWL for STM32WL Series):
  - The STM32Cube HAL, STM32 abstraction layer embedded software ensuring maximized portability across STM32 portfolio.
  - Low-layer APIs (LL) offering a fast light-weight expert-oriented layer which is closer to the hardware than the HAL. LL APIs are available only for a set of peripherals.
  - A consistent set of middleware components such as FatFS, FreeRTOS™, LoRaWAN®, SubGHz_Phy, Sigfox™, KMS, SE and mbed-crypto.
  - All embedded software utilities coming with a full set of examples.

1 STM32CubeWL main features

The STM32CubeWL MCU Package runs on STM32 32-bit microcontrollers based on the Arm® Cortex®-M processor.

Note: Arm is a registered trademark of Arm Limited (or its subsidiaries) in the US and/or elsewhere.

STM32CubeWL gathers, in a single package, all the generic embedded software components required to develop an application on STM32WL microcontrollers. In line with the STM32Cube initiative, this set of components is highly portable, not only within STM32WL Series but also to other STM32 Series.

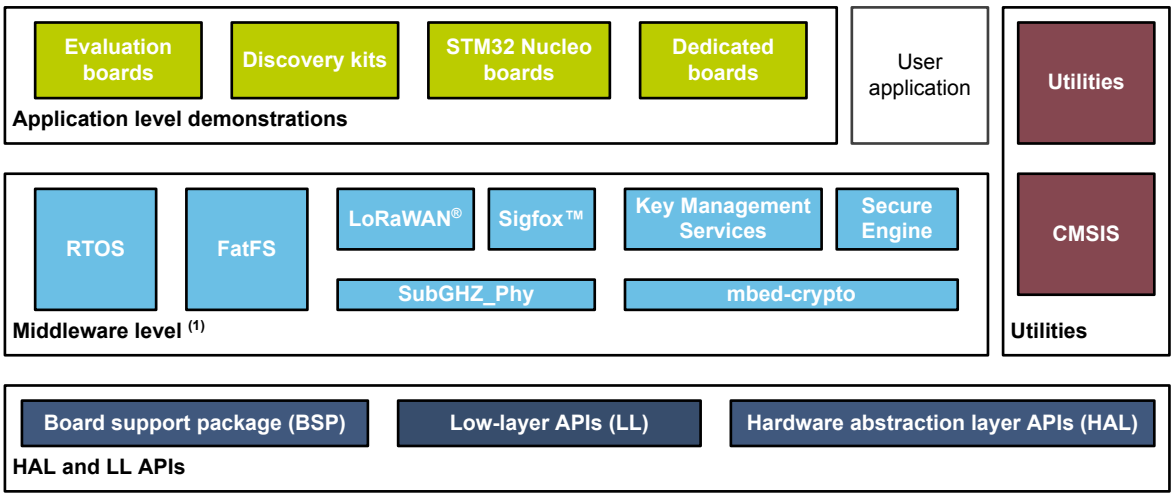
STM32CubeWL is fully compatible with STM32CubeMX code generator that allows generating initialization code. The package includes low-layer (LL) and hardware abstraction layer (HAL) APIs that cover the microcontroller hardware, together with an extensive set of examples running on STMicroelectronics boards. The HAL and LL APIs are available in open-source BSD license for user convenience.

STM32CubeWL MCU Package also contains a set of middleware components with the corresponding examples. They come in free user-friendly license terms:

- CMSIS-RTOS implementation with FreeRTOS™ open source solution
- FAT file system based on open source FatFS solution
- LoRaWAN®, Lora Wide Area Network
- SubGHz_Phy, a common Phy layer according to the OSI model for all above MAC layers
- Sigfox, the Sigfox protocol library
- KMS, Key Management Service
- SE, Secure Engine
- mbed-crypto, mbed Cryptography library

Several applications implementing all these middleware components are also provided in the STM32CubeWL MCU Package.

Figure 1. STM32CubeWL firmware components

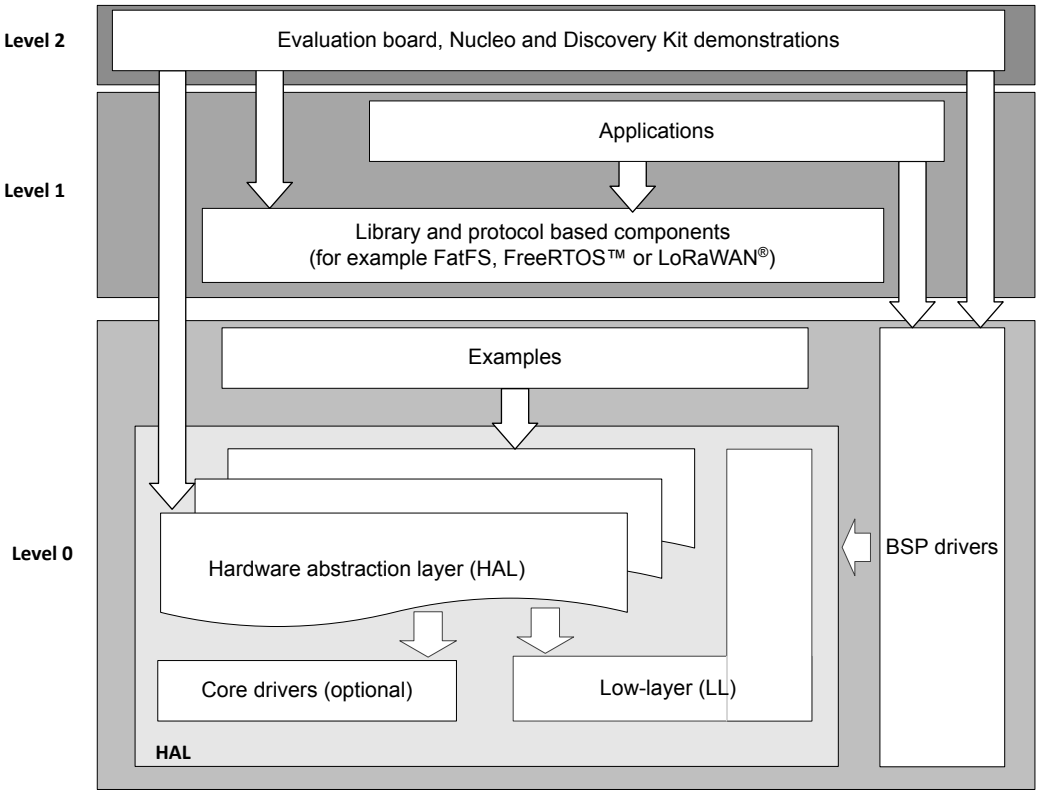


(1) The set of middleware components depends on the product series.

2 STM32CubeWL architecture overview

The STM32CubeWL firmware solution is built around three independent levels that easily interact as described in Figure 2.

Figure 2. STM32CubeWL firmware architecture



2.1 Level 0

This level is divided into three sub-layers:

- Board support package (BSP)
- Hardware abstraction layer (HAL)
  - HAL peripheral drivers
  - Low-layer drivers
- Basic peripheral usage examples

2.1.1 Board support package (BSP)

This layer offers a set of APIs relative to the hardware components in the hardware boards (such as LCD, Audio, microSD™ and MEMS drivers). It is composed of two parts:

- Component

  This is the driver relative to the external device on the board and not to the STM32. The component driver provide specific APIs to the BSP driver external components and could be portable on any other board.

- BSP driver

  It allows linking the component driver to a specific board and provides a set of user-friendly APIs. The API naming rule is `BSP_FUNCT_Action()`.

  Example: `BSP_LED_Init()`, `BSP_LED_On()`

The BSP is based on a modular architecture allowing an easy porting on any hardware by just implementing the low-level routines.

2.1.2 Hardware abstraction layer (HAL) and low-layer (LL)

The STM32CubeWL HAL and LL are complementary and cover a wide range of applications requirements:

- The HAL drivers offer high-level function-oriented highly-portable APIs. They hide the MCU and peripheral complexity to end user.

  The HAL drivers provide generic multi-instance feature-oriented APIs which simplify user application implementation by providing ready to use process. As example, for the communication peripherals (I2S, UART, and others), it provides APIs allowing initializing and configuring the peripheral, managing data transfer based on polling, interrupt or DMA process, and handling communication errors that may raise during communication. The HAL driver APIs are split in two categories:
  - Generic APIs which provides common and generic functions to all the STM32 Series
  - Extension APIs which provides specific and customized functions for a specific family or a specific part number.
- The low-layer APIs provide low-level APIs at register level, with better optimization but less portability. They require a deep knowledge of MCU and peripheral specifications.

  The LL drivers are designed to offer a fast light-weight expert-oriented layer which is closer to the hardware than the HAL. Contrary to the HAL, LL APIs are not provided for peripherals where optimized access is not a key feature, or for those requiring heavy software configuration and/or complex upper-level stack.

  The LL drivers feature:
  - A set of functions to initialize peripheral main features according to the parameters specified in data structures
  - A set of functions used to fill initialization data structures with the reset values corresponding to each field
  - Function for peripheral de-initialization (peripheral registers restored to their default values)
  - A set of inline functions for direct and atomic register access
  - Full independence from HAL and capability to be used in standalone mode (without HAL drivers)
  - Full coverage of the supported peripheral features
- Dual Core implementation:
  - The same HAL/LL drivers support both the single and dual core STM32WL lines
    - In the STM32WL dual core devices all peripherals can be accessed in the same way by the two cores (Cortex®-M4 and Cortex®-M0+). It means there is no peripherals split or default allocation between Cortex®-M4 and Cortex®-M0+. For this reason the same peripheral HAL and LL drivers are shared between the two cores.
    - Furthermore, some peripherals (mainly: RCC, GPIO, PWR, HSEM, IPCC, GTZC, ...) have additional dual core specific features:
      - “DUAL_CORE” define is used to delimit code (defines, functions, macros, ...) available only on dual core lines.
      - “CORE_CM0PLUS” define is used to delimit code where we a specific configuration/code portion for Cortex®-M0+ core on dual core lines. With inverted or "else" statement, this define is used to delimit code where we a specific configuration/code portion for Cortex®-M4 core on dual core lines.

2.1.3 Basic peripheral usage examples

This layer encloses the examples built over the STM32 peripherals using only the HAL and BSP resources.

Single core and dual core examples are available:

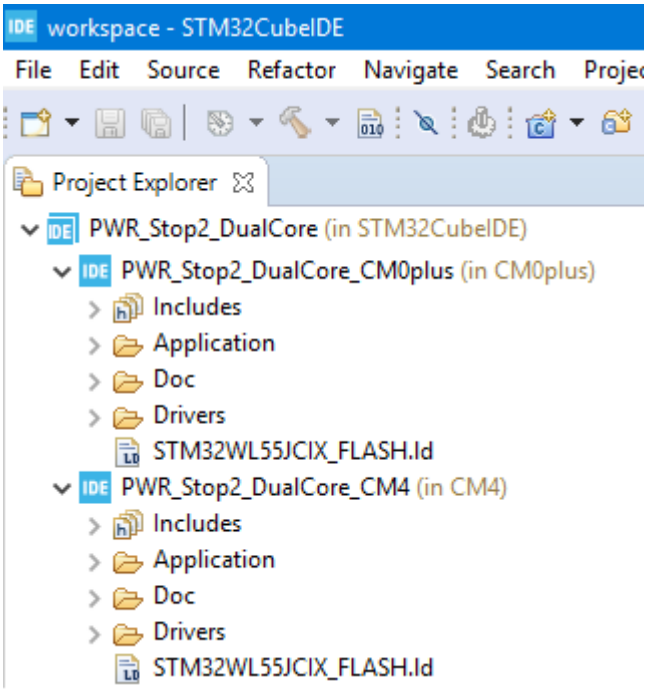
- Single core examples for peripherals or features able to be demonstrated on one core

- Dual core examples for peripherals or features that require programs running simultaneously on both cores (IPCC, HSEM, PWR specific dual core features, GTZC, ...)

Dual core examples and applications follow a dedicated architecture:

- In dual core lines, only one project (one workspace) per example/application is provided. This is to be compatible with legacy (single core lines) architecture.
- Two target projects configuration per workspace (one per core) named with suffix "CM4" and "CM0PLUS".
- Each target configuration has its own option settings: target device, linker options, RO, RW zones, preprocessor symbols (CORE_CM0PLUS) so that user code can be compiled, linked and programmed separately for each core. The compilation results in two binaries: CM4 binary and CM0+ binary.

Figure 3. Dual core project architecture



2.2 Level 1

This level is divided into two sub-layers:

- Middleware components
- Examples based on the middleware components

2.2.1 Middleware components

The middleware is a set of libraries covering FatFS, FreeRTOS™, LoRaWAN®, SubGHz_Phy, Sigfox™, KMS, SE and mbed-crypto. Horizontal interactions between the components of this layer is done directly by calling the feature APIs while the vertical interaction with the low-layer drivers is done through specific callbacks and static macros implemented in the library system call interface. For example, the FatFS implements the disk I/O driver to access microSD™ drive.

The main features of each middleware component are as follows:

- FAT file system
  - FatFS FAT open source library
  - Long file name support
  - Dynamic multi-drive support
  - RTOS and standalone operation
  - Examples with microSD™

- FreeRTOS™
  - Open source standard
  - CMSIS compatibility layer
  - Tickless operation during low-power mode
  - Integration with all STM32Cube middleware modules
- LoRaWAN®
  - It offers a very compelling mix of long range, low power consumption and secure data transmission. Public and private networks using this technology can provide coverage that is greater in range compared to that of existing cellular networks. It is easy to plug into the existing infrastructure and offers a solution to serve battery-operated IoT applications.
- SubGHz_Phy
  - It implements the PHY layer for sub-gigahertz protocol. Although specialized for LoRaWAN® MAC it can interface most sub-gigahertz protocol. It provides an abstraction layer managing transmission, radio reception handlers and timeouts. With its unique radio API, it also implements several radio drivers for other I-CUBE-LRAWAN
- Sigfox™
  - It implements the Sigfox™ protocol library compliant with the Sigfox™ protocol Network. It is also including the RF test protocol library to test against RF Sigfox™ tools.
- SE
  - The secure engine middleware (SE) provides a protected environment to manage all critical data and operations (such as cryptography operations accessing firmware encryption key and others). Protected code and data are accessible through a single entry point (called gate mechanism) and it is therefore not possible to run or access any SE code or data without passing through it, otherwise, a system reset is generated.
- KMS
  - The key management services middleware (KMS) provides cryptographic services to the user application through the PKCS #11 APIs (KEY ID-based APIs). Security is insured by locating KMS in the secure enclave. User application keys are stored in the secure enclave and can be updated in a secure way (authenticity check, decryption, and integrity check before the update).
- mbed-crypto
  - mbed-crypto middleware is delivered as open source code. This middleware provides a PSA cryptography API implementation.

2.2.2 Examples based on the middleware components

Each middleware component comes with one or more examples (called also Applications) showing how to use it. Integration examples that use several middleware components are provided as well.

3 STM32CubeWL firmware package overview

3.1 Supported STM32WL devices and hardware

STM32Cube offers a highly portable hardware abstraction layer (HAL) built around a generic architecture. It allows the build-upon layers, such as the middleware layer, to implement their functions without knowing, in-depth, the MCU used. This improves the library code re-usability and guarantees an easy portability on other devices.

In addition, thanks to its layered architecture, the STM32CubeWL offers full support of all STM32WL Series. The user has only to define the right macro in *stm32wlxx.h*.

Table 1 shows the macro to define depending on the STM32WL device used. This macro must also be defined in the compiler preprocessor.

Table 1. Macros for STM32WL Series

Macro defined in stm32wlxx.h	STM32WL devices
STM32WL54xx	STM32WL54CC, STM32WL54JC
STM32WL55xx	STM32WL55CC, STM32WL55JC
STM32WLE4xx	STM32WLE4C8, STM32WLE4CB, STM32WLE4CC, STM32WLE4J8, STM32WLE4JB, STM32WLE4JC
STM32WLE5xx	STM32WLE5C8, STM32WLE5CB, STM32WLE5CC, STM32WLE5J8, STM32WLE5JB, STM32WLE5JC

STM32CubeWL features a rich set of examples and applications at all levels making it easy to understand and use any HAL driver and/or middleware components. These examples run on the STMicroelectronics boards listed in Table 1.

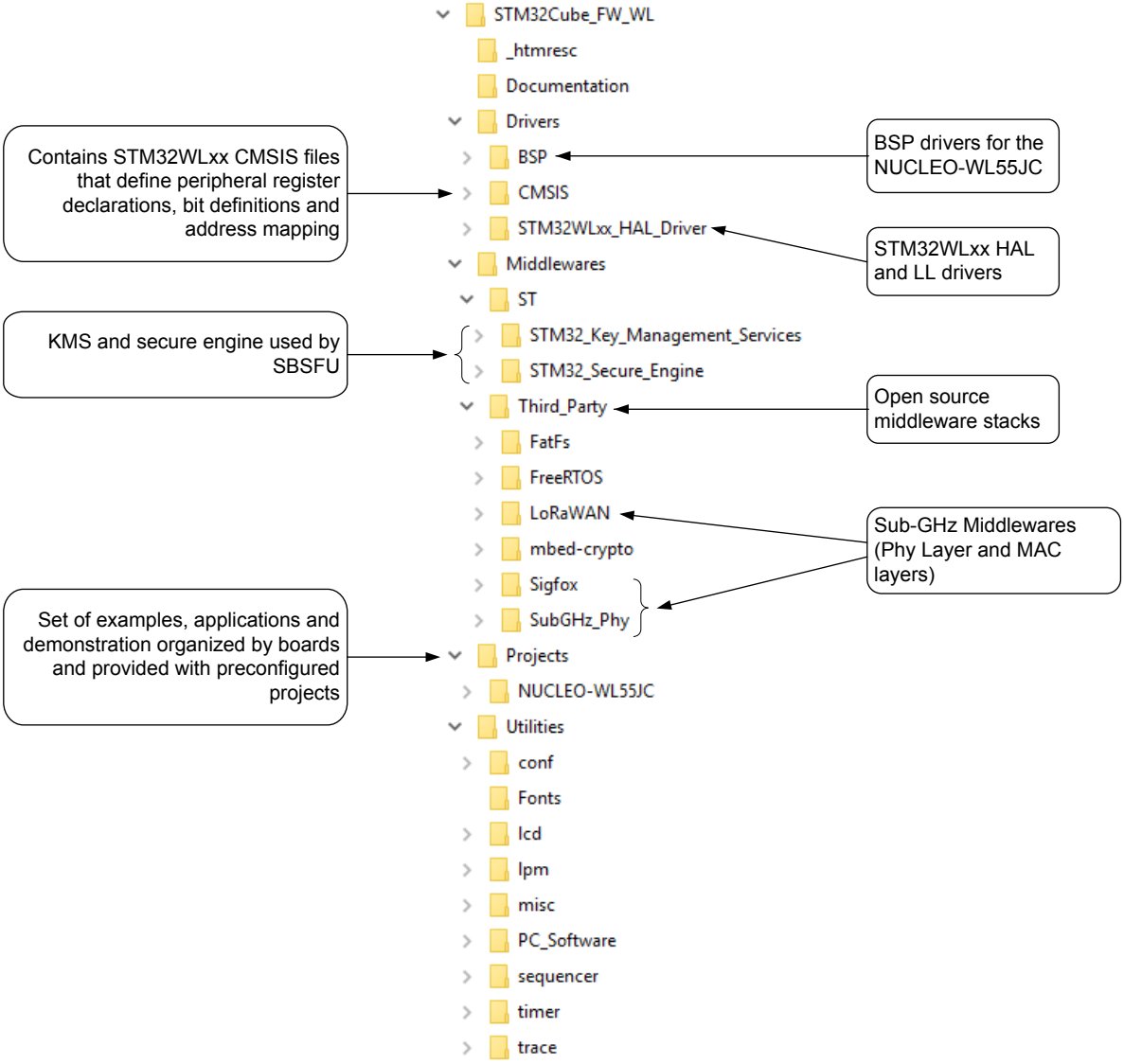
Table 2. Boards for STM32WL Series

Supported boards
NUCLEO-WL55JC

3.2 Firmware package overview

The STM32CubeWL firmware solution is provided in one single zip package having the structure shown in Figure 4.

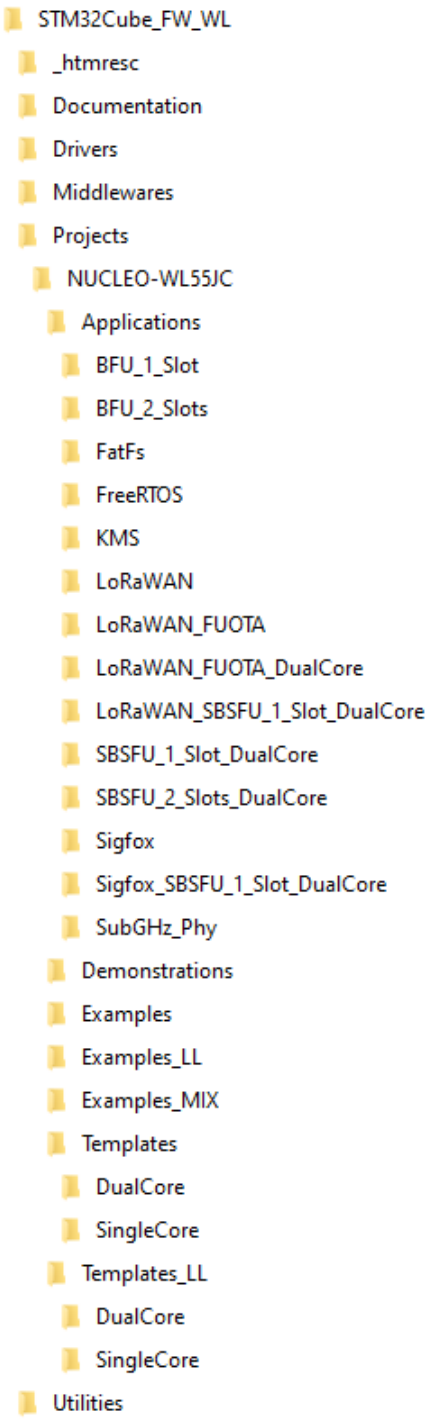
Figure 4. STM32CubeWL firmware package structure



Attention: *The components files must not be modified by the user. Only the \Projects sources are eligible to changes by the user.*

For NUCLEO-WL55JC board, a set of examples are provided with pre-configured projects for EWARM toolchain. Figure 5 shows the project structure for the NUCLEO-WL55JC board.

Figure 5. Overview of STM32CubeWL examples



The examples are classified depending on the STM32Cube level they apply to, and are named as explained below:

- Level 0 examples are called Examples, Examples_LL and Examples_MIX. They use respectively HAL drivers, LL drivers and a mix of HAL and LL drivers without any middleware component.
- Level 1 examples are called Applications. They provide typical use cases of each middleware component. The *LoRaWAN* folder contains typical Lora use cases.

Templates projects available in the Templates and Templates_LL directories permit to quickly build any firmware application on a given board. Two projects are available in each template directory: single core for all STM32WL lines and dual core projects for STM32WL5x (dual core) line.

Single core examples have the same structure:

- *Inc* folder that contains all header files.
- *Src* folder for the sources code.
- *EWARM*, *MDK-ARM*, *STM32CubeIDE* folders contain the pre-configured project for each toolchain.
- *readme.txt* describing the example behavior and needed environment to make it working
- **.ioc* file that allows users to open most of firmware examples within STM32CubeMX (starting from STM32CubeMX 6.1.0)

All dual core examples have the same structure:

- Two separate folders *CM4* and *CM0PLUS* respectively for Cortex®-M4 and Cortex®-M0+
- Each folder (*CM4* and *CM0PLUS*) provides:
  - \Inc folder that contains all header files for Cortex®-M4/M0+
  - \Src folder for the sources code files for Cortex®-M4/M0+
- A *common* folder with *Inc* and *Src* containing the common header and source files for both cores.
- *EWARM*, *MDK-ARM*, *STM32CubeIDE* folders contain the pre-configured project for each toolchain (both Cortex®-M4 and Cortex®-M0+ target configuration)
- *readme.txt* describing the example behavior and needed environment to make it working
- **.ioc* file that allows users to open most of firmware examples within STM32CubeMX (starting from STM32CubeMX 6.1.0)

4 Getting started with STM32CubeWL

4.1 Running your first example

This section explains how simple is to run a first example within STM32CubeWL. It uses as illustration the generation of a simple LED toggle running on NUCLEO-WL55JC board:

1. Download the STM32CubeWL firmware package. Unzip it into a directory of your choice. Make sure not to modify the package structure shown in Figure 5. Note that it is also recommended to copy the package at a location close to your root volume (*C:\Eval* or *G:\Tests*) because some IDEs encounter problems when the path length is too long.
2. Browse to *\Projects\NUCLEO-WL55JC\Examples*.
3. Open *\GPIO*, then *\GPIO_EXTI* folders.
4. Open the project with your preferred toolchain. A quick overview on how to open, build and run an example with the supported toolchains is given below.
5. Rebuild all files and load your image into target memory.
6. Run the example: each time you press the user pushbutton (SW1), LED2 toggles (for more details, refer to the example *readme* file).

To open, build and run an example with the supported toolchains, follow the steps below:

EWARM

1. Under the *example* folder, open *\EWARM* sub-folder
2. Launch the Project.eww workspace(*)
3. Rebuild all files: Project→Rebuild all
4. Load project image: Project→Debug
5. Run program: Debug→Go(F5)

MDK-ARM

1. Under the *example* folder, open the *\MDK-ARM* subfolder.
2. Open the Project.uvproj workspace(*)
3. Rebuild all files: Project→Rebuild all target files.
4. Load project image: Debug→Start/Stop Debug Session.
5. Run program: Debug→Run (F5).

• STM32CubeIDE

1. Open the STM32CubeIDE toolchain.
2. Click File→Open projects from file system
3. Browse to the STM32CubeIDE workspace directory and select the project.
4. Rebuild all project files: select the project in the “Project explorer” window then click on Project→build project menu.

(*): The workspace name may change from one example to another.

4.2 Developing your own application

4.2.1 Using STM32CubeMX to develop or update your application

In the STM32CubeWL MCU Package, nearly all Example projects are generated with the STM32CubeMX tool to initialize the system, peripherals and middleware.

The direct use of an existing Example project from the STM32CubeMX tool requires STM32CubeMX 6.1.0 or higher:

- After the installation of STM32CubeMX, open and if necessary update a proposed project. The simplest way to open an existing project is to double-click on the *.ioc file so that STM32CubeMX automatically opens the project and its source files.

- The initialization source code of such projects is generated by STM32CubeMX; the main application source code is contained by the comments “USER CODE BEGIN” and “USER CODE END”. In case the IP selection and setting are modified, STM32CubeMX updates the initialization part of the code but preserves the main application source code.

For developing a custom project in the STM32CubeMX, follow the step-by-step process:

1. Select the STM32 microcontroller that matches the required set of peripherals.
2. Configure all the required embedded software using a pinout-conflict solver, a clock-tree setting helper, a power consumption calculator, and the utility performing MCU peripheral configuration (such as GPIO or USART) and middleware stacks (such as USB).
3. Generate the initialization C code based on the selected configuration. This code is ready to use within several development environments. The user code is kept at the next code generation.

For more information about STM32CubeMX, refer to the STM32CubeMX for STM32 configuration and initialization C code generation (UM1718).

For a list of the available example projects for the STM32CubeWL, refer to the STM32Cube firmware examples for STM32WL Series application note (AN5409).

4.2.2 Drivers applications

4.2.2.1 HAL application

This section describes the steps required to create a user HAL application using STM32CubeWL:

1. **Create user project**

   To create a new project, start either from the *Template* project provided for each board under *\Projects\<STM32xxx_yyy>\Templates* or from any available project under *\Projects\<STM32xxy_yyy>\Examples* or *\Projects\<STM32xx_yyy>\Applications* (where *<STM32xxx_yyy>* refers to the board name, such as NUCLEO-WL55JC).

   The Template project is providing empty main loop function, however it is a good starting point to get familiar with project settings for STM32CubeWL. The template has the following characteristics:

   – It contains the source code of HAL, CMSIS and BSP drivers which are the minimal components required to develop a code on a given board.

   – It contains the include paths for all the firmware components.

   – It defines the STM32WL device supported, thus allowing to configure the CMSIS and HAL drivers accordingly.

   – It provides read-to-use user files pre-configured as shown below:

   HAL initialized with default time base with ARM Core SysTick.

   SysTick ISR implemented for `HAL_Delay()` purpose.

Note: When copying an existing project to another location, make sure to update the include paths.

2. **Add the necessary middleware to user project (optional)**

   The available middleware stacks are: FatFS, FreeRTOS™ and LoRaWAN®. To know which source files must be added to the project file list, refer to the documentation provided for each middleware. It is possible to look at the applications available under *\Projects\STM32xxx_yyy\Applications\<MW_Stack>* (where *<MW_Stack>* refers to the middleware stack, such as FreeRTOS™) to know which source files and which include paths must be added.

3. **Configure the firmware components**

   The HAL and middleware components offer a set of build time configuration options using macros # define declared in a header file. A template configuration file is provided within each component, it has to be copied to the project folder (usually the configuration file is named *xxx_conf_template.h*, the word ‘_template’ needs to be removed when copying it to the project folder). The configuration file provides enough information to know the impact of each configuration option. More detailed information is available in the documentation provided for each component.

4. **Start the HAL Library**

   After jumping to the main program, the application code must call `HAL_Init()` API to initialize the HAL Library, which do the following tasks:

   a. Configuration of the Flash prefetch and SysTick interrupt priority (through macros defined in *stm32wlxx_hal_conf.h*).

   b. Configuration of the SysTick to generate an interrupt every millisecond at the SysTick interrupt priority `TICK_INT_PRIO` defined in *stm32wlxx_hal_conf.h*.

   c. Setting of NVIC Group Priority to 0.

   d. Call of `HAL_MspInit()` callback function defined in stm32wlxx_hal_msp.c user file to perform global low-level hardware initializations.

5. **Configure the system clock**

   The system clock configuration is done by calling the two APIs described below:

   a. `HAL_RCC_OscConfig()`: this API configures the internal and/or external oscillators, as well as the PLL source and factors. The user chooses to configure one oscillator or all oscillators. The PLL configuration can be skipped if there is no need to run the system at high frequency.

   b. `HAL_RCC_ClockConfig()`: this API configures the system clock source, the Flash memory latency and AHB and APB prescalers.

6. **Initialize the peripheral**

   a. First write the peripheral `HAL_PPP_MspInit` function. Proceed as follows:

      - Enable the peripheral clock.
      - Configure the peripheral GPIOs.
      - Configure the DMA channel and enable DMA interrupt (if needed).
      - Enable peripheral interrupt (if needed).

   b. Edit the *stm32xxx_it.c* to call the required interrupt handlers (peripheral and DMA), if needed.

   c. Write process complete callback functions if peripheral interrupt or DMA is needed.

   d. In user *main.c* file, initialize the peripheral handle structure then call the function `HAL_PPP_Init()` to initialize user peripheral.

7. **Develop user application**

   At this stage, the system is ready and user application code development can start.

   a. The HAL provides intuitive and ready-to-use APIs to configure the peripheral. It supports polling, interrupts and DMA programming model, to accommodate any application requirements. For more details on how to use each peripheral, refer to the rich examples set provided in the STM32CubeWL MCU Package.

   b. If user application has some real-time constraints, a large set of examples showing how to use FreeRTOS™ and integrate it with all middleware stacks is provided within STM32CubeWL. This is a good starting point to develop user application.

Caution: In the default HAL implementation, SysTick timer is used as timebase: it generates interrupts at regular time intervals. If `HAL_Delay()` is called from peripheral ISR process, make sure that the SysTick interrupt has higher priority (numerically lower) than the peripheral interrupt. Otherwise, the caller ISR process is blocked. Functions affecting timebase configurations are declared as `__weak` to make override possible in case of other implementations in user file (using a general purpose timer for example or other time source). For more details, refer to `HAL_TimeBase` example.

4.2.2.2 *LL application*

This section describes the steps needed to create your own LL application using STM32CubeWL.

1. **Create your project**

   To create a new project you either start from the Templates_LL project provided for each board under *\Projects\<STM32xxx_yyy>\Templates_LL* or from any available project under *\Projects\<STM32xxy_yyy>\Examples_LL (<STM32xxx_yyy>* refers to the board name, such as NUCLEO-WL55JC).

   The Template project provides an empty main loop function, however it is a good starting point to get familiar with project settings for STM32CubeWL.

   Template main characteristics are the following:

   – It contains the source codes of the LL and CMSIS drivers which are the minimal components needed to develop code on a given board.
   – It contains the include paths for all the required firmware components.
   – It selects the supported STM32WL device and allows to configure the CMSIS and LL drivers accordingly.
   – It provides ready-to-use user files, that are pre-configured as follows:

     main.h: LED & USER_BUTTON definition abstraction layer.

     main.c: System clock configuration for maximum frequency.

2. **Port an existing project to another board**

   To port an existing project to another target board, start from the Templates_LL project provided for each board and available under *\Projects\<STM32xxx_yyy>\Templates_LL*:

   a. Select a LL example

      To find the board on which LL examples are deployed, refer to the list of LL examples *STM32CubeProjectsList.html*.

   b. Port the LL example

      ◦ Copy/paste the *Templates_LL* folder - to keep the initial source - or directly update existing *Templates_LL* project.
      ◦ Then porting consists principally in replacing *Templates_LL* files by the *Examples_LL* targeted project.
      ◦ Keep all board specific parts. For reasons of clarity, board specific parts have been flagged with specific tags:

```
/* ============== BOARD SPECIFIC CONFIGURATION CODE BEGIN ============== */
/* ============== BOARD SPECIFIC CONFIGURATION CODE END ============== */
```

      Thus the main porting steps are the following:

      ◦ Replace the *stm32wlxx_it.h* file
      ◦ Replace the *stm32wlxx_it.c* file
      ◦ Replace the main.h file and update it: keep the LED and user button definition of the LL template under `'BOARD SPECIFIC CONFIGURATION'` tags.
      ◦ Replace the main.c file and update it:

        Keep the clock configuration of the `SystemClock_Config()` LL template function under `'BOARD SPECIFIC CONFIGURATION'` tags.

        Depending on LED definition, replace each LEDx occurrence with another LEDy available in *main.h*.

Thanks to these adaptations, the example is functional on the targeted board.

4.2.2.3 *How to migrate a single core example from CPU1 to CPU2*

STM32WL single core examples are run from CPU1 (CM4) only, CPU2 (CM0+) remaining disable.

It is also possible to run any single core example on CPU2, following these steps:

1. Start from a dual core project. To ease project creation, some minimalist dual core projects are available in STM32WL firmware package:
   – Template dual core using HAL driver:
     ...\Firmware\Projects\NUCLEO-WL55JC\Templates\DualCore
   – Template dual core using LL driver:
     ...\Firmware\Projects\NUCLEO-WL55JC\Templates_LL\DualCore
2. Select any example single core to migrate to CPU2
3. Copy source files from example to template.
   Example with HAL example GPIO_EXTI migration to dual core template:
   – Copy content of *GPIO_EXTI/Inc* to *Templates\DualCore\CM0PLUS\Inc*
   – Copy content of *GPIO_EXTI/Src* to *Templates\DualCore\CM0PLUS\Src*

Note: Program of CPU1 is compatible with CPU2, therefore can be executed by CPU2, with some exceptions:

- *IRQ handlers and interruptions are specific to selected core: In files stm32wlxx_it.h and stm32wlxx_it.c, IRQ handlers must be customized to the selected core. In source files, interruptions configurations with NVIC functions must also be customized accordingly. For example, IRQ handler MemManage_Handler is available only on CPU1 (CM4). Refer to list of IRQ handlers in startup files of the selected CPU.*
- *Few peripherals features are specific to selected core: refer to compilation switch `CORE_CM0PLUS` in HAL and LL drivers.*

4. Remove system clock configuration in CPU1 program: dual core templates perform system clock configuration by CPU1 (generic configuration) before performing CPU2 boot. By copying example program in CPU2 folder, system clock configuration is also performed by CPU2. This system clock configuration is matching the example requirements and must be kept.

Note: An alternate solution is to perform system clock configuration by CPU1 only, with configuration matching the example requirements. However, in case of wake up from Standby or Shutdown modes, CPU2 program must have the routines to restore system clock.

Note: Program of CPU1 is now performing only 1 action: boot CPU2.

5. Optionally, CPU1 can be put in low power mode after CPU2 boot. The deepest low power mode is recommended (Shutdown mode) to allow CPU2 to manage all system low power modes (Stop, Standby, Shutdown) on demand.
   Example of code to make CPU1 enter in Shutdown mode using HAL driver:

```
  /* Request to enter in Shutdown mode */
  HAL_PWREx_EnterSHUTDOWNMode();
```

Example of code to make CPU1 enter in Shutdown mode using LL driver:

```
  /* Request to enter in Shutdown mode */
  LL_PWR_SetPowerMode(LL_PWR_MODE_SHUTDOWN);

  /* Set SLEEPDEEP bit of Cortex System Control Register */
  LL_LPM_EnableDeepSleep();

  /* This option is used to ensure that store operations are completed */
#if defined ( __CC_ARM)
  __force_stores();
#endif
  /* Request Wait For Interrupt */
  __WFI();
```

Caution: Regeneration of dual core template with CubeMX (file *.ioc*) would overwrite the code migrated previously
Optionally, CubeMX configuration of the dual core template (file *.ioc*) can be updated to match the example migrated to CPU2.

4.2.3 Security applications

This package is delivered with security applications.

4.2.3.1 *KMS applications*

KMS applications demonstrates key management services usage:

- data encryption and decryption with AES key (embedded keys)
- data signature and verification with RSA key (embedded keys)
- data encryption and decryption based on key derivation (embedded keys)
- generation of KMS blob binary on PC side (used to import keys on devices)
- data encryption and decryption with AES key (keys imported with KMS blob)

4.2.3.2 *SBSFU applications*

4.2.3.2.1 STM32WL5x (dual core) line

STM32WL5x dual core devices permit to support secure solution thanks to TZ-like dual core isolation and other security features (such as HDP, WRP, RDP).

The SBSFU application demonstrates a secure boot and secure firmware update for 2 images (M4 user application and M0+ LPWAN stack).

SBSFU applications are delivered in 2 firmware update configurations:

- 2 slots (1 download slot used to update the execution slot matching the downloaded image)
- 1 slot (no download slot, the downloaded image is written directly into its matching execution slot).

Note: This example is given using UART for the download part. Nevertheless, another example is given to demonstrate firmware update over the air, using LORA download.

4.2.3.2.2 STM32WLEx (single core) line

STM32WLEx single core devices do not support full security features.

Nevertheless, firmware update is possible and the following BFU application is delivered: boot and firmware update for 1 image, using 1 slot. Download is performed over UART.

STM32WLEx single core devices do not support full security features.

Nevertheless, firmware update is possible and the BFU applications are delivered in 2 configurations:

- 2 slots (1 download slot associated to 1 execution slot)
- 1 slot (no download slot, the image is written directly into the execution slot).

4.2.4 RF applications

Three types of RF application are available in the package. They are listed below:

LoRaWan® examples are LoRaWan® examples implementing a LoRaWan® device exercising the LoRaWan® stack and the SubGHz_Phy RF driver. They are located in *Projects\NUCLEO-WL55JC\Applications*\.

- LoRaWAN\LoRaWAN_AT_Slave implements a LoRaWan® modem that is controlled though AT command interface over UART by an external host. This application is available in single core and dual core format with and without KMS (key management system). More information can be found in AN5406 and AN5481. LoRaWAN_AT_Slave devices can be connected to STM32CubeMonitor to send AT commands to the device, perform packet error rate measurement and more.
- LoRaWAN\LoRaWAN_End_Node implements a LoRaWan® application device sending sensors data to LoRaWan® network server. This application is available in single core, dual core with optionally FreeRTos and KMS. More information can be found in in AN5406.
- LoRaWAN_FUOTA_DualCore implements a dual core LoRaWan® application device sending sensors data to LoRaWan® network server. Moreover, the LoRaWan® application and stack can be updated over the air from a LoRaWan® network server using the SBSFU as a secure boot and secure firmware update framework. As well, the LoRaWAN_FUOTA example is fully secured using isolation features provided by the STM32WL55. More information is provided under AN5554.
- LoRaWAN_FUOTA implements a single core LoRaWan® application device sending sensors data to LoRaWan® network server. Moreover, the LoRaWan® application and stack can be updated over the air from a LoRaWan® network server using the BFU as a boot and firmware update framework.

- LoRaWAN_SBSFU_1_Slot_DualCore implements a dual core Secure LoRaWan® application device sending sensors data to LoRaWan® network server. This application is secured with security environment offered by secure boot (SB) and secure engine. The application can also be updated with local loader Y-modem (SFU).

SubGHz_Phy example: these applications feature the SubGHz_Phy radio middleware. These examples can be found under *Projects\NUCLEO-WL55JC\Applications\SubGHz_Phy*. More information is provided in application note AN5406.

- the PingPong application is available in both single core and dual core format. The PingPong application features a radio link between two PingPong devices.
- the PER application is available in single core only. It features a radio link between a transmitting device and a receiving device and count packet error rate (PER).

Sigfox™ example: these applications implement Sigfox™ up and running Sigfox™ device. They can be found in project path *Projects\NUCLEO-WL55JC\Applications*

- Sigfox™ applications are available in both single core and dual core format. Dual core project can run with or without KMS. More information is provided under AN5480.
- Sigfox_AT_Slave implements a Sigfox™ application modem that is controlled though AT command interface over UART by an external host, like a computer executing a terminal.
- Sigfox_PushButton an example of a Sigfox™ object sending temperature and battery level to a Sigfox™ network when pressing a user button.
- Sigfox_SBSFU_1_Slot_DualCore implements a dual core Secure Sigfox® application device sending sensors data to Sigfox® network. This application is secured with security environment offered by secure boot (SB) and secure engine. The application can also be updated with local loader Y-modem (SFU).

4.2.5 RF demonstration

The local network demonstration features a non LoRaWan® local network with one concentrator and up to 14 sensors that can connect and send sensor data to the concentrator. More information is provided in UM2786. Local network projects are located under *Projects\NUCLEO-WL55JC\Demonstrations\LocalNetwork*.

- One **concentrator** project flashed into STM32WLxx_Nucleo board implements a concentrator sending one beacon frame and one sync frame every 16 seconds to administrate a network of up to 14 sensors and receives each connected sensor data. The concentrator can be connected to STM32CubeMonitor to configure the geographical area and display the list of sensors detected and connected sensor data.
- One **sensor** project that implements one sensor sending data to the concentrator.

4.2.6 Getting STM32CubeWL release updates

The new STM32CubeWL MCU Package releases and patches are available from www.st.com/stm32wl. They may be retrieved from the "CHECK FOR UPDATE" button in STM32CubeMX. For more details, refer to section 3 of STM32CubeMX for STM32 configuration and initialization C code generation (UM1718).

5 FAQ

5.1 What is the license scheme for the STM32CubeWL firmware?

The HAL is distributed under a non-restrictive BSD (berkeley software distribution) license.

The middleware stacks LoRaWAN® and SubGHz_Phy made by Semtech and STMicroelectronics are distributed under a non-restrictive BSD (berkeley software distribution) license.

The middleware stack Sigfox™ is distributed under SLA0044 and Sigfox™ specific terms.

The middleware security secure engine and key management service are distributed under SLA0044.

The middleware based on well-known open-source solutions (FreeRTOS™ and FatFS) have user-friendly license terms.

For more details, refer to the license agreement of each middleware.

5.2 What boards are supported by the STM32CubeWL firmware package?

NUCLEO-WL55JC

5.3 Are any examples provided with the ready-to-use toolset projects?

Yes. STM32CubeWL provides a rich set of examples and applications. They come with the pre-configured projects for IAR™-based toolchain.

5.4 Is there any link with standard peripheral libraries?

The STM32Cube HAL and LL drivers are the replacement of the standard peripheral library:

- The HAL drivers offer a higher abstraction level compared to the standard peripheral APIs. They focus on peripheral common features rather than hardware. Their higher abstraction level allows defining a set of user-friendly APIs that are easily portable from one product to another.
- The LL drivers offer low-layer APIs at registers level. They are organized in a simpler and clearer way than direct register accesses. LL drivers also include peripheral initialization APIs, which are more optimized compared to what is offered by the SPL, while being functionally similar. Compared to HAL drivers, these LL initialization APIs allows an easier migration from the SPL to the STM32Cube LL drivers, since each SPL API has its equivalent LL API(s).

5.5 Does the HAL layer take benefit from interrupts or DMA? How can this be controlled?

Yes. The HAL layer supports three API programming models: polling, interrupt and DMA (with or without interrupt generation).

5.6 How are the product/peripheral specific features managed?

The HAL drivers offer extended APIs, including specific functions as add-ons to the common API to support features available on some products/lines only.

5.7 When should I use HAL versus LL drivers?

HAL drivers offer high-level and function-oriented APIs, with a high level of portability. Product/peripherals complexity is hidden for end users.

LL drivers offer low-layer APIs at registers level, with a better optimization but less portability. They require a deep knowledge of product/peripherals specifications.

5.8 How can I include LL drivers in my environment? Is there any LL configuration file as for HAL?

There is no configuration file. Source code must directly include the necessary *stm32wlxx_ll_ppp.h* file(s).

5.9 Can I use HAL and LL drivers together? If yes, what are the constraints?

It is possible to use both HAL and LL drivers. One handles the peripheral initialization phase with HAL and then manages the I/O operations with LL drivers.

The major difference between HAL and LL is that HAL drivers require to create and use handles for operation management while LL drivers operate directly on peripheral registers. Mixing HAL and LL is illustrated in *Examples_MIX* example.

5.10 Are there any LL APIs which are not available with HAL?

Yes, there are.

A few Cortex® APIs have been added in *stm32wlxx_ll_cortex.h* , for instance for accessing SCB or SysTick registers.

5.11 Why are SysTick interrupts not enabled on LL drivers?

When using LL drivers in standalone mode, you do not need to enable SysTick interrupts because they are not used in LL APIs, while HAL functions requires SysTick interrupts to manage timeouts.

5.12 How are LL initialization APIs enabled?

The definition of LL initialization APIs and associated resources (structure, literals and prototypes) is conditioned by the `USE_FULL_LL_DRIVER` compilation switch.

To be able to use LL APIs, add this switch in the toolchain compiler preprocessor.

5.13 How can STM32CubeMX generate code based on embedded software?

STM32CubeMX has a built-in knowledge of STM32 microcontrollers, including their peripherals and software, that is used to provide a graphical representation to the user and generate **.h/*.c* files based on user configuration.

5.14 How to get regular updates on the latest STM32CubeWL MCU Package releases?

Refer to Section 4.2.6 .

Revision history

Table 3. **Document revision history**

Date	Revision	Changes
11-Dec-2019	1	Initial release.
12-Oct-2020	2	Updated Section Introduction. Updated Section 1 STM32CubeWL main features. Updated Figure 1. STM32CubeWL firmware components. Updated Section 2.1.2 Hardware abstraction layer (HAL) and low-layer (LL). Updated Section 2.1.3 Basic peripheral usage examples. Updated Section 3 STM32CubeWL firmware package overview. Updated Section 4.1 Running your first example. Added Section 4.2.1 Using STM32CubeMX to develop or update your application. Reorganized Section 4.2.2 Drivers applications. Added Section 4.2.2.3 How to migrate a single core example from CPU1 to CPU2. Updated Section 5.1 What is the license scheme for the STM32CubeWL firmware?. Added Section 5.13 How can STM32CubeMX generate code based on embedded software?. Added Section 5.14 How to get regular updates on the latest STM32CubeWL MCU Package releases?.
16-Mar-2021	3	Replaced STM32CubeMonitor-RF by STM32CubeMonitor in Section Introduction.
07-Jun-2021	4	Updated Figure 5. Overview of STM32CubeWL examples. Updated Section 4.2.3.2 SBSFU applications Updated Section 4.2.4 RF applications

Contents

List of tables

List of figures

IMPORTANT NOTICE – PLEASE READ CAREFULLY

STMicroelectronics NV and its subsidiaries (“ST”) reserve the right to make changes, corrections, enhancements, modifications, and improvements to ST products and/or to this document at any time without notice. Purchasers should obtain the latest relevant information on ST products before placing orders. ST products are sold pursuant to ST’s terms and conditions of sale in place at the time of order acknowledgement.

Purchasers are solely responsible for the choice, selection, and use of ST products and ST assumes no liability for application assistance or the design of Purchasers’ products.

No license, express or implied, to any intellectual property right is granted by ST herein.

Resale of ST products with provisions different from the information set forth herein shall void any warranty granted by ST for such product.

ST and the ST logo are trademarks of ST. For additional information about ST trademarks, please refer to www.st.com/trademarks. All other product or service names are the property of their respective owners.

Information in this document supersedes and replaces information previously supplied in any prior versions of this document.

© 2021 STMicroelectronics – All rights reserved