UM2388

User manual

Development guidelines for STM32Cube firmware Packs

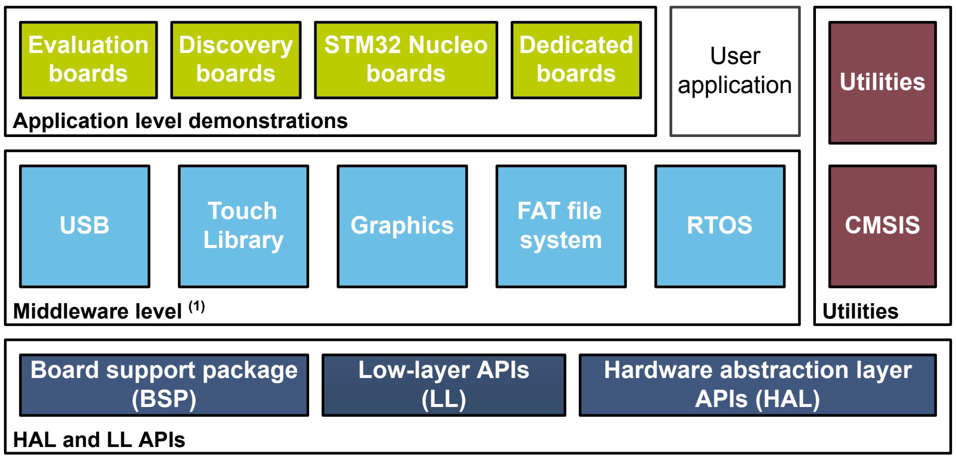
Introduction

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

STM32Cube includes:

- STM32CubeMX, a graphical software configuration tool that allows the generation of C initialization code using graphical wizards.
- A comprehensive embedded software platform, delivered per Series (such as STM32CubeF4 for STM32F4 Series):
  - The STM32Cube HAL, STM32 abstraction layer embedded software ensuring maximized portability across the STM32 portfolio. HAL APIs are available for all peripherals.
  - 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 FAT file system, RTOS, USB, TCP/IP and Graphics.
  - All embedded software utilities, delivered with a full set of examples.

Figure 1. **STM32Cube components**



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

The STM32Cube Expansion Packages contain embedded software components that complement STM32Cube functionalities and enable using STMicroelectronics microcontroller devices in various application domains.

The present user manual describes the concept of firmware components with a focus on the various component interfaces, as well as the interaction model with the different STM32Cube Package layers (HAL, BSP and middleware). It also describes the associated CMSIS Pack located within the PDSC file as specified by Arm.

This document assumes that the reader is familiar with STM32Cube architecture, HAL/LL APIs and programming models. A complete documentation on STM32Cube MCU Packages is available from *www.st.com*.

1 Reference documents

- *STM32Cube BSP drivers development guidelines* user manual (UM2298)
- *Development guidelines for STM32Cube Expansion Packages* user manual (UM2285)

2 Acronyms

Table 1. **List of acronyms**

Acronym	Definition
API	Application Programming Interface
BSP	Board Support Package
CMSIS	Cortex Microcontroller Software Interface Standard
CPU	Central Processing Unit
EXTI	External interrupt/event controller
FAT	File Allocation Table
GPIO	General purpose I/Os
HAL	Hardware abstraction layer
I2C	Inter-integrated circuit
IDE	Integrated Development Environment
LL	Low Layer drivers
MSP	MCU Specific Package
MW	Middleware
PPP	STM32 peripheral or block
SPI	Serial Peripheral interface

3 Definitions

- **Unitary Pack**: a CMSIS Pack that contains a single firmware component as defined by the STM32Cube specification, such as STM32F7 HAL Pack, STM32F7 BSP Pack or LwIP Pack.
- **Standalone Pack**: a set of unitary Packs integrated together and coming with a set of applications that fulfil a single function, such as MEMS Pack or LoRaWAN® Pack.
- **Application Pack**: an application-level Pack that requires additional standalone Packs and that must contain a set of specific applications on top of the elementary Packs. The application Packs are only used with STM32CubeMX and cannot be open by another third-party Pack manager.

4 STM32Cube MCU Package architecture

4.1 Overview of STM32Cube MCU Packages

STM32Cube MCU Packages gather in one single package per Series all the generic embedded software components required to develop an application on Arm®Cortex® STM32 microcontrollers. Following STM32Cube initiative, this set of components is highly portable within all STM32 Series.

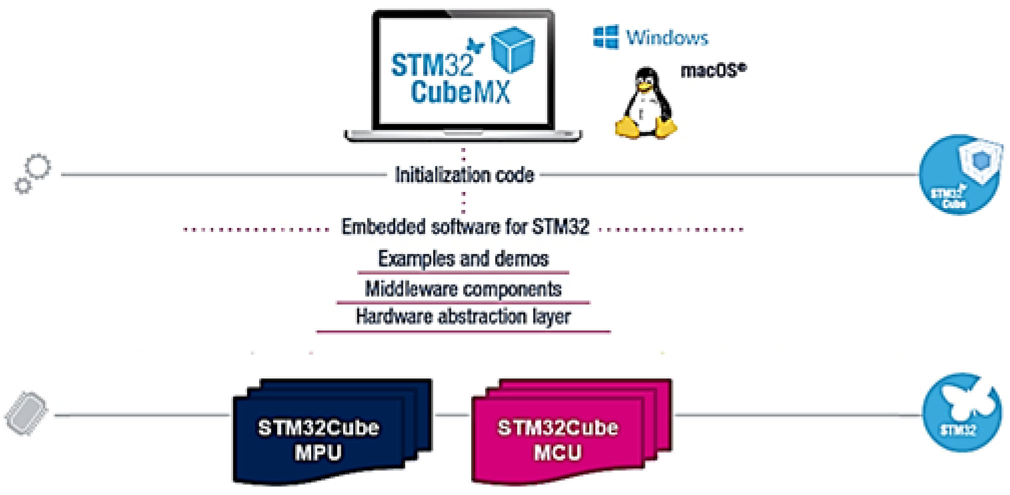
STM32Cube Packages are fully compatible with STM32CubeMX code generator that allows the generation of initialization code. The packages include a low-layer (LL) and a hardware abstraction layer (HAL) that covers the microcontroller hardware, together with an extensive set of examples running on all STMicroelectronics boards.

STM32Cube MCU Packages also contain a set of middleware components with the corresponding examples:

- Full USB Host and Device stack supporting many classes.
  - Host Classes: HID, MSC, CDC, Audio, MTP
  - Device Classes: HID, MSC, CDC, Audio, DFU
- STemWin, a professional graphical stack solution available in binary format and based on STMicroelectronics partner solution SEGGER emWin
- CMSIS-RTOS implementation with FreeRTOS open source solution
- FAT File system based on open source FatFS solution
- TCP/IP stack based on open source LwIP solution
- SSL/TLS secure layer based on open source mbedTLS
- LibJPEG Free JPEG decoder/encoder library

In addition, STM32Cube MCU Packages come with global demonstrations, one per board, based on the different components (drivers and middleware).

Figure 2. STM32Cube MCU Packages

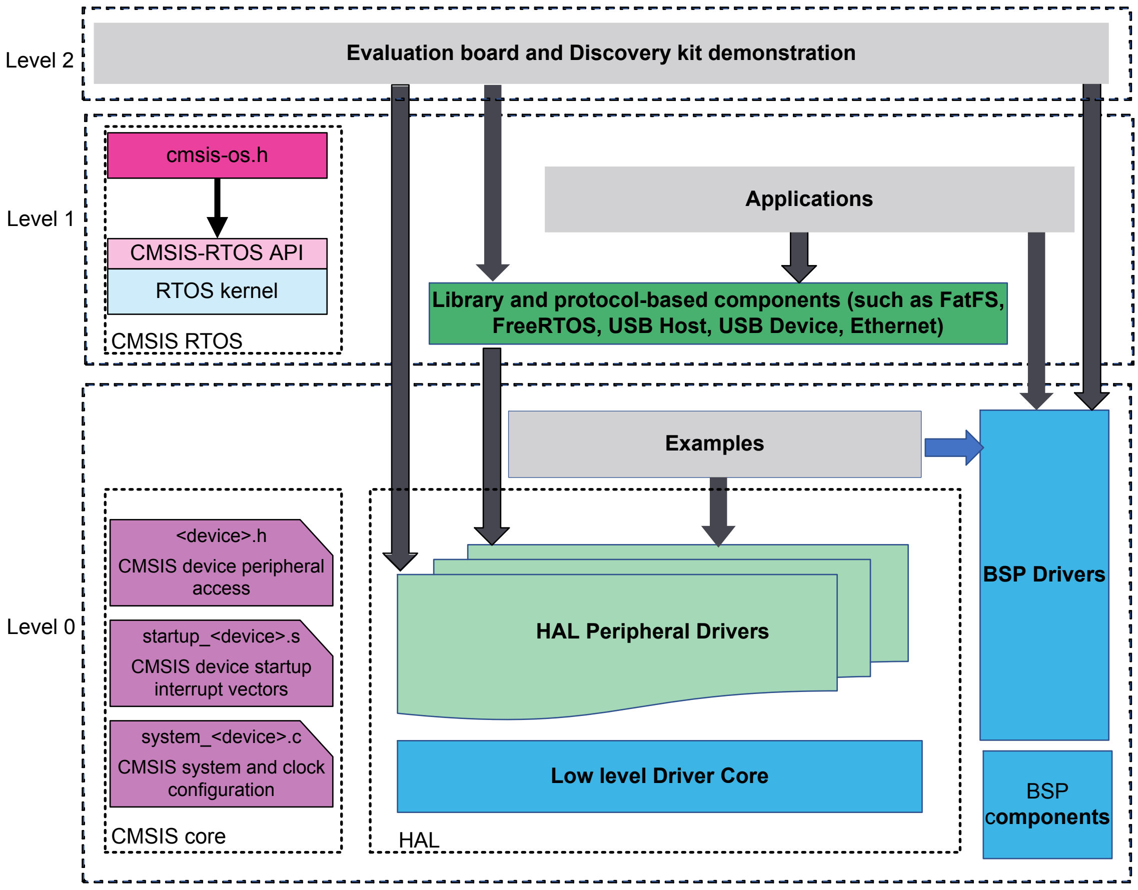


Note: Arm is a registered trademark of Arm Limited (or its subsidiaries) in the US and/or elsewhere.

4.2 Overview of STM32Cube MCU Package architecture

STM32Cube MCU Packages are built around a miscellaneous software utilities and three independent levels that can easily interact with each other through the high-level APIs of each modules and dedicated interface layer (middleware) (seeFigure 3. STM32Cube MCU Package architecture).

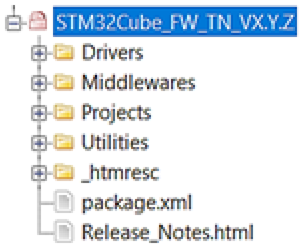
Figure 3. **STM32Cube MCU Package architecture**


Level 2
Evaluation board and Discovery kit demonstration
Level 1
cmsis-os.h
CMSIS-RTOS API
RTOS kernel
CMSIS RTOS
Applications
Library and protocol-based components (such as FatFS, FreeRTOS, USB Host, USB Device, Ethernet)
Level 0
Examples
BSP Drivers
<device>.h
CMSIS device peripheral access
startup_<device>.s
CMSIS device startup interrupt vectors
system_<device>.c
CMSIS system and clock configuration
CMSIS core
HAL Peripheral Drivers
Low level Driver Core
HAL
BSP components

4.3 Packaging model

The STM32Cube MCU Package names must contain the STM32 Series identifier *TN* as well as the package version *VX.Y.Z*. The packages must be organized according to the following directory scheme:

Figure 4. **STM32Cube MCU Package directory scheme**



```
STM32Cube_FW_TN_VX.Y.Z
├ Drivers
├ Middlewares
├ Projects
├ Utilities
├ _htmresc
├ package.xml
└ Release_Notes.html
```

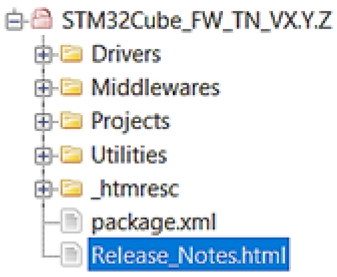
The firmware version must be in the following format, *VX.Y.Z*, where:

- *X* represents the major version index. It must be incremented if there is a compatibility breach in the software API.
- *Y* represents the minor version index. It must be incremented if a functionality is added in a backward-compatible manner, that is without compatibility breach in the software API.
- *Z* represents the patch version index. It must be incremented on bug fixes or when a cosmetic change, without impact on functionalities, is performed.

The root folder is composed of:

- *Release_Notes.html*: HTML document that must contain the whole history of official releases.

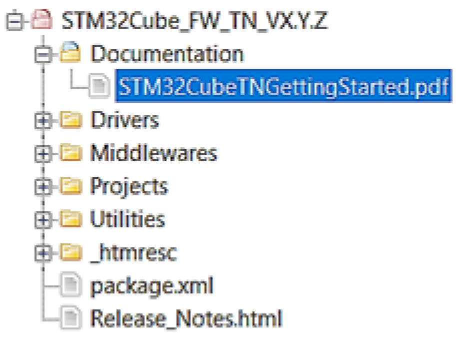
Figure 5. STM32Cube MCU Package directory scheme: Release_Notes folder



```
STM32Cube_FW_TN_VX.Y.Z
├─ Drivers
├─ Middlewares
├─ Projects
├─ Utilities
├─ _htmresc
├─ package.xml
└─ Release_Notes.html
```

- *Documentation*: folder containing the *STM32CubeYNGettingStarted.pdf* document explaining how to get started with the STM32Cube Package.

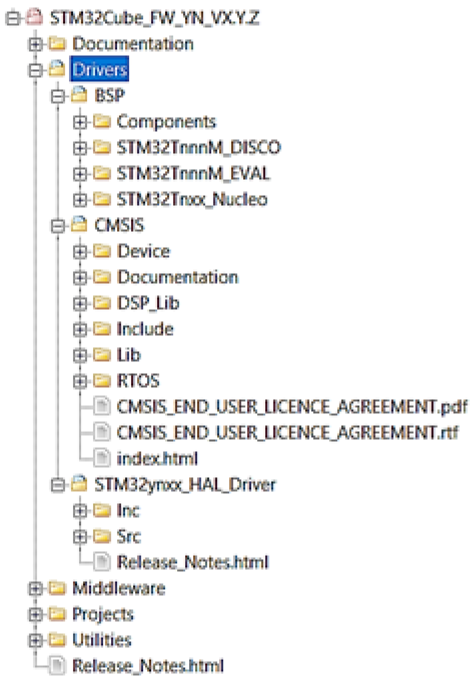
Figure 6. STM32Cube MCU Package directory scheme: Documentation folder



```
STM32Cube_FW_TN_VX.Y.Z
├─ Documentation
│  └─ STM32CubeTNGettingStarted.pdf
├─ Drivers
├─ Middlewares
├─ Projects
├─ Utilities
├─ _htmresc
├─ package.xml
└─ Release_Notes.html
```

- *Drivers*: folder containing the different level-0 components including the CMSIS core files.

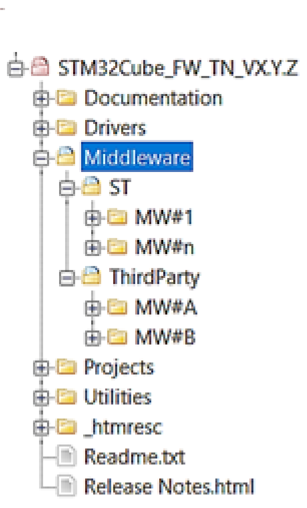
Figure 7. **STM32Cube MCU Package directory scheme: Drivers folder**



```
STM32Cube_FW_YN_VX.Y.Z
├─ Documentation
├─ Drivers
│  ├─ BSP
│  │  ├─ Components
│  │  ├─ STM32TnnnM_DISCO
│  │  ├─ STM32TnnnM_EVAL
│  │  └─ STM32Txxx_Nucleo
│  ├─ CMSIS
│  │  ├─ Device
│  │  ├─ Documentation
│  │  ├─ DSP_Lib
│  │  ├─ Include
│  │  ├─ Lib
│  │  ├─ RTOS
│  │  ├─ CMSIS_END_USER_LICENCE_AGREEMENT.pdf
│  │  ├─ CMSIS_END_USER_LICENCE_AGREEMENT.rtf
│  │  └─ index.html
│  └─ STM32yxxx_HAL_Driver
│     ├─ Inc
│     ├─ Src
│     └─ Release_Notes.html
├─ Middleware
├─ Projects
├─ Utilities
└─ Release_Notes.html
```

- *Middlewares*: folder containing the different native STMicroelectronics and third-party stacks and protocol-based libraries.

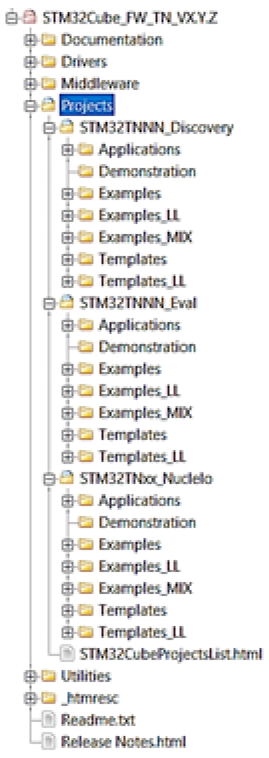
Figure 8. **STM32Cube MCU Package directory scheme: Middleware folder**



```
STM32Cube_FW_TN_VX.Y.Z
├─ Documentation
├─ Drivers
├─ Middleware
│  ├─ ST
│  │  ├─ MW#1
│  │  └─ MW#n
│  └─ ThirdParty
│     ├─ MW#A
│     └─ MW#B
├─ Projects
├─ Utilities
├─ _htmresc
├─ Readme.txt
└─ Release Notes.html
```

- *Projects*: set of applicative projects that explain and provide use cases of the different product features based on the product hardware (boards, STM32 core features, interconnections and peripherals) and built around the different firmware components.

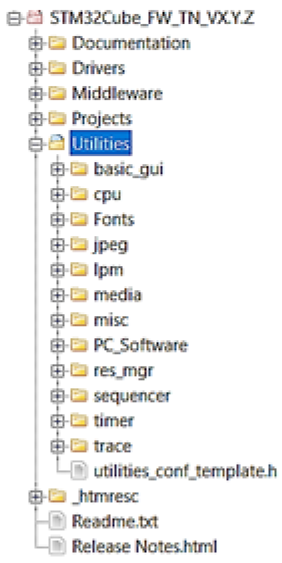
Figure 9. STM32Cube MCU Package directory scheme: Projects folder



```
STM32Cube_FW_TN_VX.Y.Z
  Documentation
  Drivers
  Middleware
  Projects
    STM32TNNN_Discovery
      Applications
      Demonstration
      Examples
      Examples_LL
      Examples_MIX
      Templates
      Templates_LL
    STM32TNNN_Eval
      Applications
      Demonstration
      Examples
      Examples_LL
      Examples_MIX
      Templates
      Templates_LL
    STM32TNxx_Nucleo
      Applications
      Demonstration
      Examples
      Examples_LL
      Examples_MIX
      Templates
      Templates_LL
    STM32CubeProjectsList.html
  Utilities
  _htmresc
  Readme.txt
  Release Notes.html
```

- *Utilities*: miscellaneous software utilities that provide additional system and media resource services such as CPU usage calculation with FreeRTOS, fonts used by the LCD drivers, time server, low-power manager, several trace utilities, and standard library services such as memory, string, time and math services.

Figure 10. STM32Cube MCU Package directory scheme: Utilities folder



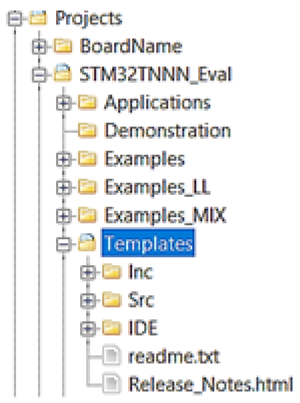
```
STM32Cube_FW_TN_VX.Y.Z
    Documentation
    Drivers
    Middleware
    Projects
    Utilities
        basic_gui
        cpu
        Fonts
        jpeg
        lpm
        media
        misc
        PC_Software
        res_mgr
        sequencer
        timer
        trace
        utilities_conf_template.h
    _htmresc
    Readme.txt
    Release Notes.html
```

4.4 Project organization

Two models of folder structure can be adopted when using a high-level firmware component middleware in STM32Cube MCU Packages:

- Basic structure: the basic structure is often used with HAL/LL examples (level-0 projects). This structure consists in placing the IDE configuration folder at the same level as the sources and organized in source and include subfolders.

Figure 11. Basic folder structure

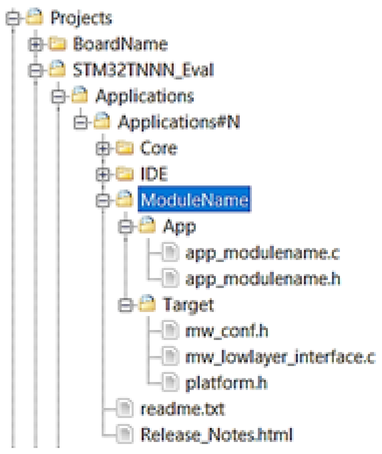


```
Projects
  BoardName
  STM32TNNN_Eval
    Applications
    Demonstration
    Examples
    Examples_LL
    Examples_MIX
    Templates
      Inc
      Src
      IDE
      readme.txt
      Release_Notes.html
```

- Advanced structure: the advanced structure is a modular structure that enables a more efficient and organized folder model by grouping application files per module and simplifying the identification of the specific components and platform-agnostic ones within the project file. This makes the application integration easier when several modules or middleware components are used.

  The advanced structure consists of:
  - A *Core* folder containing main.c/h, stm32tnxx_it.c/h, HAL config and msp files.
  - A *ModuleName/App* folder containing the platform-independent application files relative to a given module.
  - A *ModuleName/target* folder containing the platform-dependent applications files (such as middleware interfaces and configuration files) relative to a given module.

Figure 12. Advanced folder structure



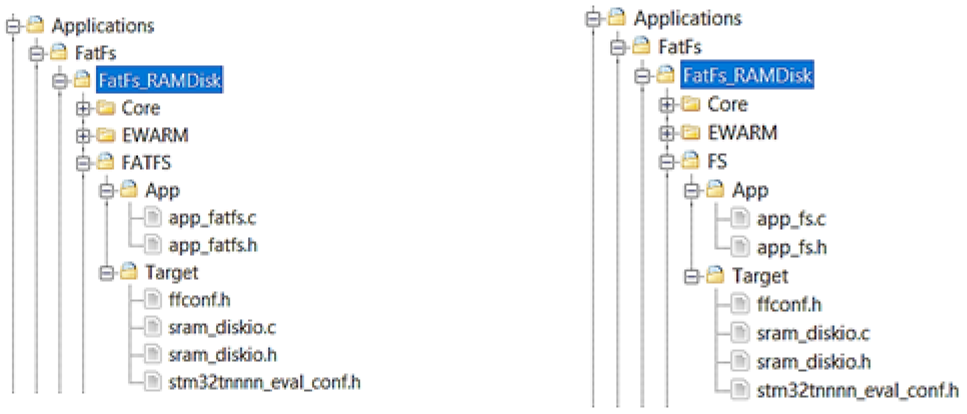
```
Projects
  BoardName
  STM32TNNN_Eval
    Applications
      Applications#N
        Core
        IDE
        ModuleName
          App
            app_modulename.c
            app_modulename.h
          Target
            mw_conf.h
            mw_lowlayer_interface.c
            platform.h
        readme.txt
        Release_Notes.html
```

A module is a set of application files that fulfil the same functionality. These files are identified by the high-level APIs that must be platform independent and by the target files relative to the platform being used. Typical STM32Cube Package modules are the application files on top of the middleware components or the BSP board and components drivers.

In the fist case, one module must be provided for each middleware component (calling the middleware core high-level APIs) and must in turn be called by the core application files.

For independent middleware applications, there is a single module per middleware component that is built on top of the middleware high-level APIs. It can have the same name as the middleware component or of the class to which the middleware belongs (see Figure 13. Project folder structure for independent middleware application modules):

Figure 13. Project folder structure for independent middleware application modules

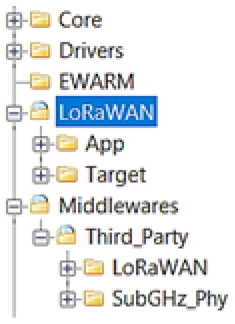


```
Applications
  FatFs
    FatFs_RAMDisk
      Core
      EWARM
      FATFS
        App
          app_fatfs.c
          app_fatfs.h
        Target
          ffconf.h
          sram_diskio.c
          sram_diskio.h
          stm32tnnnn_eval_conf.h
```

```
Applications
  FatFs
    FatFs_RAMDisk
      Core
      EWARM
      FS
        App
          app_fs.c
          app_fs.h
        Target
          ffconf.h
          sram_diskio.c
          sram_diskio.h
          stm32tnnnn_eval_conf.h
```

For interdependent middleware based applications, some middleware are come on top of others. In this case the associated module always references the middleware that exports its services to the application or the class to which the middleware belongs. For example, the LoRaWAN® middleware uses the sub-GHz middleware by its low-level interface (see Figure 14. Project folder structure for LoRaWAN® application module).

Figure 14. Project folder structure for LoRaWAN® application module

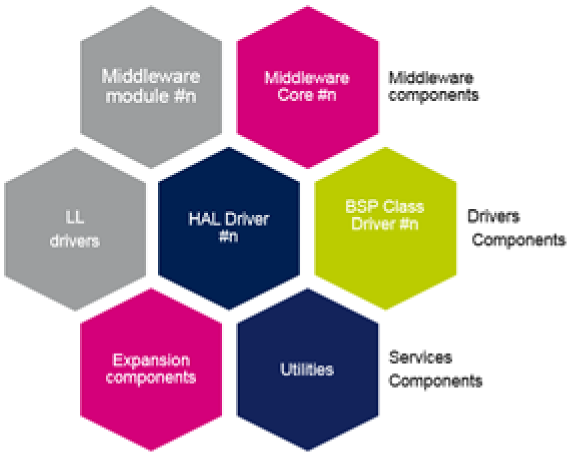


```
Core
Drivers
EWARM
LoRaWAN
  App
  Target
Middlewares
  Third_Party
    LoRaWAN
    SubGHz_Phy
```

5 STM32Cube firmware components

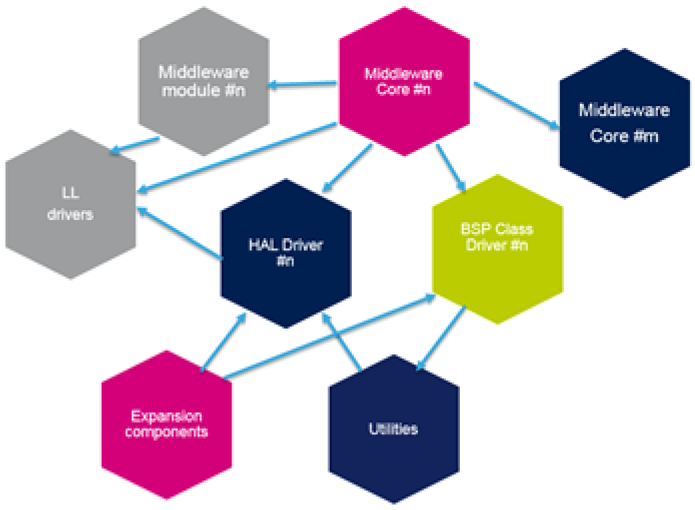
In the STM32Cube framework, the layers and firmware categories (middleware, projects and drivers) correspond to a macroscopic view of the firmware entity hierarchy within the global architecture. However the STM32Cube MCU Package entities are considered as standalone bricks that are referred to as '*STM32Cube firmware component*' or simply '*firmware component*' in the next sections of the present document.

Figure 15. **STM32Cube firmware components**



The firmware components are standalone entities. This means that they must not depend on each other from architecture point of view, whereas it can be the case from functional point of view.

Figure 16. **STM32Cube firmware component interactions**



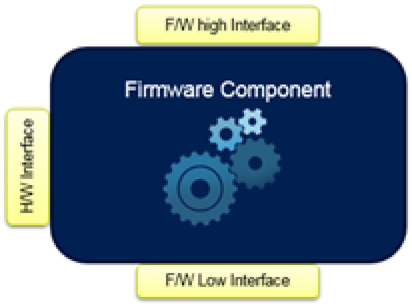
For example, the middleware components are linked to the HAL/BSP drivers to get working, but they are standalone components since they can interface with the hardware through other components (such as LL drivers, user library or direct register access).

5.1 Component concept

A firmware component is a set of functions relative to a hardware device or a system. The firmware functions of the component are activated by external controls or external actions issues by hardware or firmware. They often communicate with other devices to perform functional operations, to configure, calibrate and diagnose the device, or to output log messages.

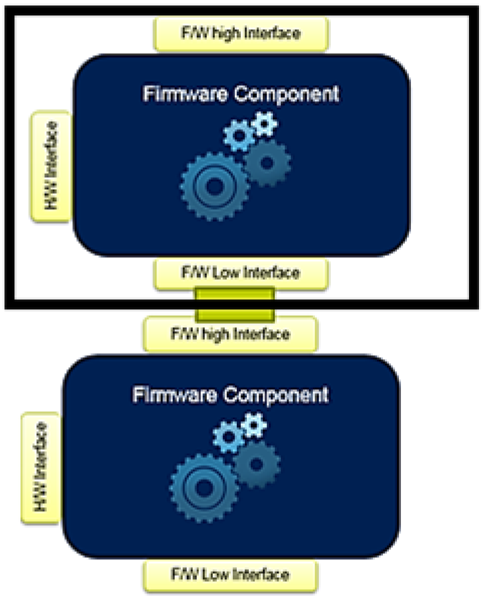
The firmware component complexity depends on the complexity of the devices it is used to control. The component can provide both one-shot services and a set of protocol-based services built on an internal state machine to manage the internal functional states handled by its own processes and the interactions between its submodules.

Figure 17. **Firmware component internal interfaces**



The firmware component must provide a standalone set of services and interact with the external world (application, other firmware components or services) thanks to dedicated communication services (interfaces) based on various techniques (glue methods).

Figure 18. **Firmware component external interfaces**



5.2 Component interfaces

The interfaces are part of the firmware component call and entry points. They must consequently be standardized for a given component. This results in the following interface requirements:

- Limited number of exported header files for each interface
- Limited set of APIs
- Fully customizable low-layer interface (open to the user)
- APIs referenced by a API specific version that can differ from the core firmware version

Refer to Figure 17. Firmware component internal interfaces for an overview of the firmware component interfaces.

The component can interact with other layers and other firmware components through three types of interfaces:

- **High interface**: application programming interface based on a set of defined API initialization/configuration services and operation services (such as transfer, action or background tasks).
- **Low interface**: low-level services required by a component to communicate with other components. Generally the low interface is composed of a set of MW_Low_Level_Function routines that must be filled from other component high-level services or from a set of function pointers (fops) that need to be linked to this interface.
- **Hardware interface**: it allows the access to the hardware through the BSP, HAL, CMSIS, LL interface or direct register access for user oriented services (such as debug console, RTC, or USB CDC-class communication interface)

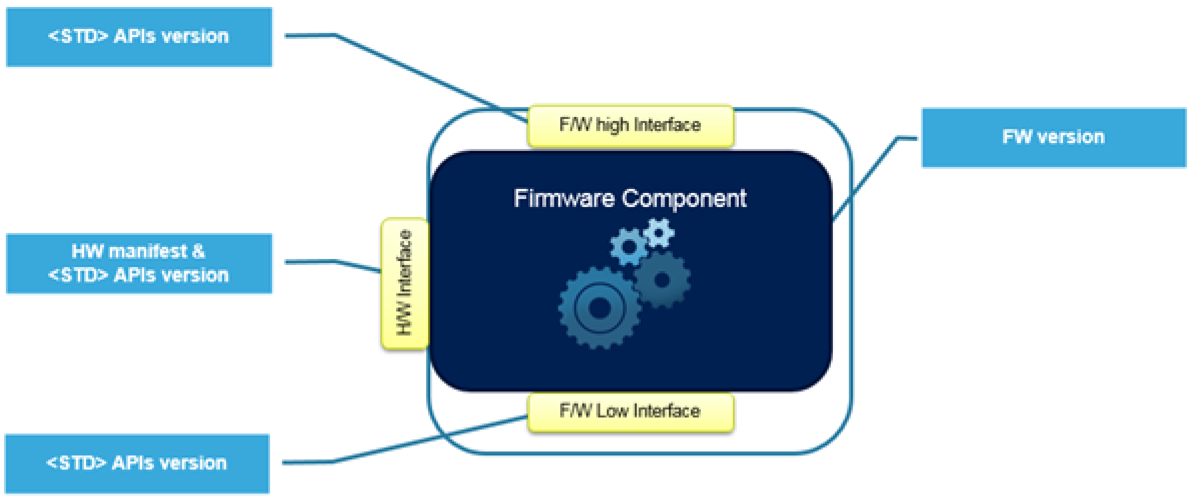
Several components can integrate additional services to interact with the full environment, such as RTOS services or memory allocation that is not considered as a standard interface.

Note: The low-layer interface is provided in a template format with empty or partially empty functions.

5.3 Component description

A component is identified in the firmware framework by its firmware version, the version and hardware manifest of its high-level and low-level APIs, and its standard APIs.

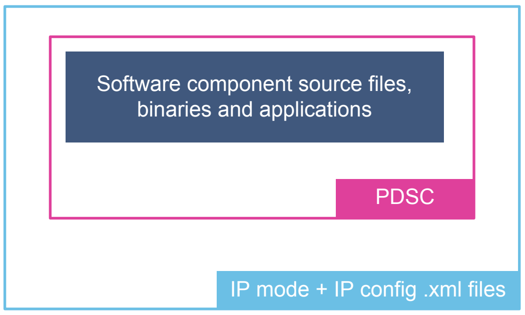
Figure 19. **Firmware component detailed description**



A component Pack is a standardized way to define a deliverable software component. Depending on the information provided in the Pack, two levels of Pack usage through STM32CubeMX can be distinguished:

- If no STM32CubeMx STM32CubeMX files are present in the Pack, then the tool can only copy the software components of the Pack in the generated project.
- If the Pack includes STM32CubeMX configuration files, a user interface is used to configure the software components and the initialization C code is generated accordingly in the project.
  - *.xml files: they describe how the software component configuration parameters can be modified through the STM32CubeMX user interface
  - *.flt files: they provide the initialization code to be generated by the STM32CubeMX

Figure 20. **Component Pack structure**


Software component source files, binaries and applications
PDSC
IP mode + IP config .xml files

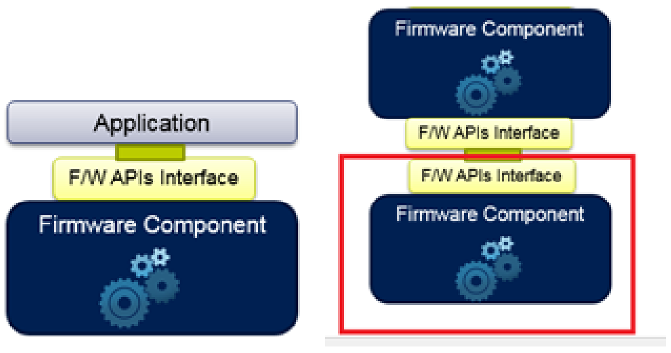
5.4 Component interaction model

The firmware component interfaces can be of three types: high, low and hardware (refer to Section 5.2 Component interfaces for a description of each interface type).

Depending on the component integration model, the low interface and the hardware interface can be present or not. This leads to have four component interface configuration as described below:

- Single interface component: only the high interface is available and a set of direct access APIs are provided. The interface can be used by the application or by another high-level component.

Figure 21. **Single interface firmware component**

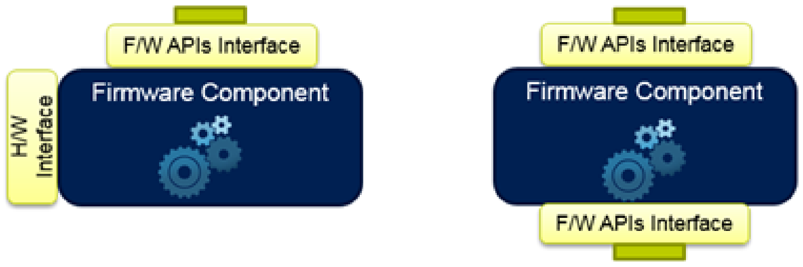


In the STM32Cube framework, the single interface components generally include the HAL drivers, the LL drivers, the utilities and the logical services (with no hardware access such as LibJPEG).

Note: The HAL drivers interact internally with the LL driver services, with exclusive access to unitary functions and direct hardware access for optimization purposes. Since the interaction is internal and not visible for the end-user, the HAL drivers do not have an explicit low interface.

- Dual interface component: in addition to the mandatory high interface, such a component can have an additional interface than can be either a low or a hardware interface.

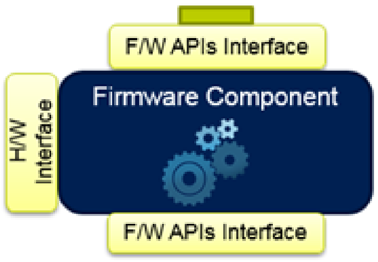
Figure 22. **Dual interface firmware component**



In the STM32Cube framework, the dual interface components generally include the BSP drivers as well as several middleware components.

- Multiple interface component: such a component includes the three types of components.

Figure 23. **Multiple interface firmware component**



In the STM32Cube framework, the multiple interface components generally include some middleware components.

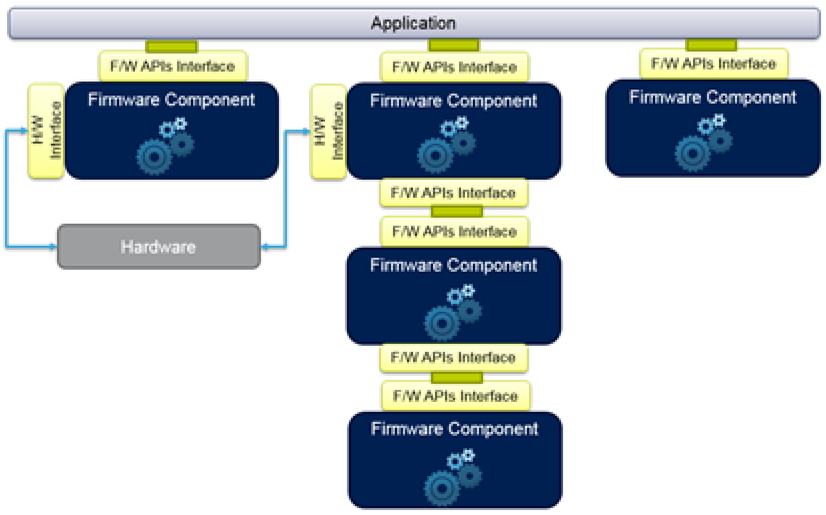
The following table summarizes the interface availability for the firmware components in the native STM32Cube framework.

Table 2. **Firmware component interface availability in native STM32Cube MCU Packages**

Component	High interface	Low interface	Hardware interface
HAL drivers	X	X (implicit to LL)	-
LL drivers	X	-	-
BSP drivers	X	X (components)	-
FreeRTOS + CMSIS OS	X	-	X
File System (FatFS)	X	X	X
LibJPEG	X	-	-
TCP/IP stack (LwIP)	X	X	-
mbedTLS	X	X	-
USB Host Library	X	X	X (classes)
USB Device library	X	X	X (classes)
Graphical libraries	X	X	-
Utilities	X	X	-
Applications	-	X	X

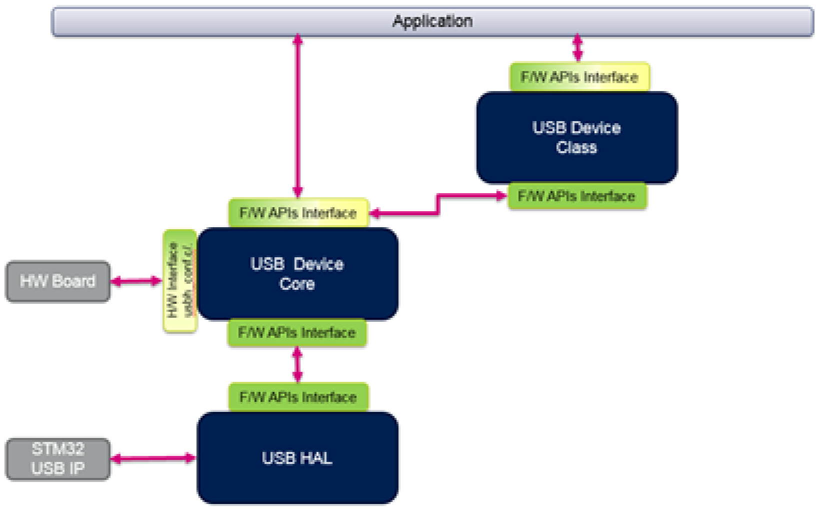
The following figure shows an example of components interaction model that uses the different interface types.

Figure 24. **Firmware component interaction model**



An example of full interaction based on the USB Device middleware is shown below. Both USB Device Class and Core provide high-level APIs to the application. The class is dynamically connected to the core through internal glue (Register Class) and uses common core APIs for dedicated class request and data transfer operations. The core is linked to the USB PCD HAL driver through the low interface and to the hardware services for the class specific operations.

Figure 25. **Example of firmware component full interaction using USB Device middleware**



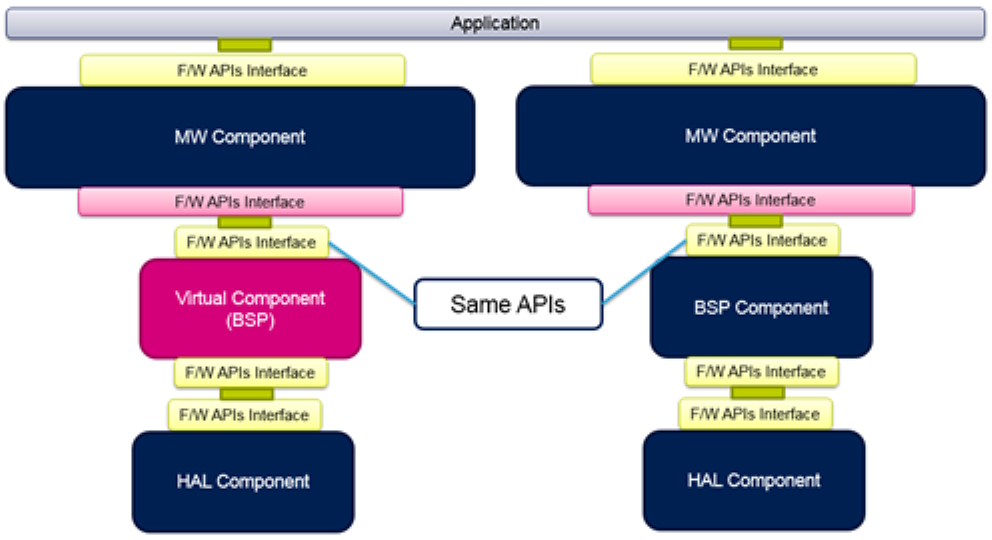
Note: A component can have several hardware interfaces. They must be located in separate files according to the required service family or class. For example, hardware interfaces can support message console, storage services (such as microSD) for USB Device (mass storage) and RTC services for the FatFS.

5.5 Virtual components

The virtual components are an alternative to missing official components (BSP, HAL drivers). The virtual components are based on the APIs of the different missing components and allow to create a temporary solution to substitute missing services. A virtual component can be easily removed and replaced by the official component once available. This is particularly useful when one or several BSP class drivers are missing. In this case temporary virtual components can be created based on the APIs of the corresponding interfaces.

The following figure shows how to use a virtual component to replace missing BSP services for a specific class.

Figure 26. **Replacing missing BSP services by a virtual component**



The virtual component advantage is to avoid directly using lower layers (the HAL in this case) to add missing services, which will affect the generated code. The virtual component is a light implementation of the official component based on HAL service calls. It can have restrictions, such as being available only for a limited number of STM32 Series. However it must implement consistent APIs whose code must be portable across all existing implementations so that the virtual component can be seamlessly replaced by the official implementation, when it is available.

One of the main case of virtual component usage is to provide BSP common services for controlling and driving LEDs and buttons. Standard BSP APIs can be called instead of creating unitary applicative services such as *MX_GPIO_Init ()* and *HAL_GPIO_WritePin ()* in the *main.c* file or in middleware low interfaces. However the internal implementation is completely generated by the STM32CubeMX using the HAL service APIs (such as GPIO and buses).

The virtual components must be stored in the application space under the *Services* folder when the application *Core* and *Middleware* folders model are used, or directly in the project *Sources* and *Include* folders when no middleware is used.

Note: *No Pack is generated for a virtual component since it is an ephemeral firmware entity. This type of component is fully created by the user and simply generated by STM32CubeMX.*

6 STM32Cube Pack architecture

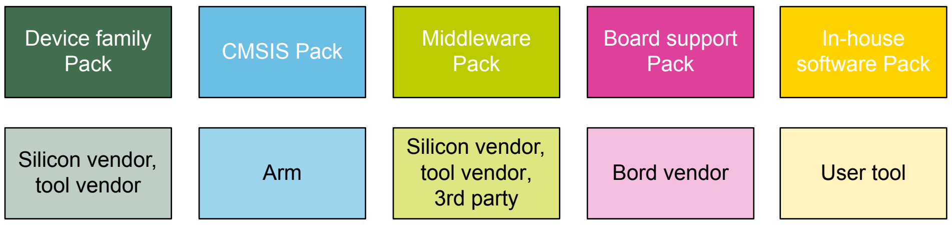
6.1 CMSIS Pack overview

The CMSIS Pack describes a delivery mechanism for software components, device parameters and evaluation board support. The XML-based package description (PDSC) file describes the content of a software Pack (file collection). It includes:

- source code, header files, and software libraries
- documentation and source code templates
- device parameters along with start-up code and programming algorithms
- example projects

The complete file collection along with the PDSC file are shipped as a software Pack, in zip format. The PDSC file is designed for software development environments and describes the user- and device-relevant context for the files supplied within this software Pack. The Pack can cover all the STM32Cube MCU Package firmware components in addition to external services as shown in Figure 27 :

Figure 27. **Content of the CMSIS Pack**



A Pack is set of components (middleware, drivers, and projects). The location and interaction between components are defined through the CMSIS Pack attributes and keywords.

The firmware components can be distributed through software Packs. A software Pack is a zip file containing a single Pack description file that describes dependencies towards devices, processors, toolchains, or other software components.

Each Pack description file (*.PDSC) contains a descriptive text of the software Pack as well as information on devices, components, and examples. Information to ease software Pack downloading, updating and versioning is also provided. In addition, the *.PDSC file includes the complete Software Pack revision history, which contains a brief list of the most significant changes.

The XML schema file, PACK.xsd, defines the sections used in a *.PDSC file. The current PACK.xsd can be found under the ARM.CMSIS.*.Pack located in the *.\CMSIS\Utilities-directory* folder.

The Pack Description (*.PDSC) Format is structured using grouping elements and specified according to Keil® documentation available from https://www.keil.com web site.

6.2 CMSIS Pack component organization

The */package/components* element describes the software components contained in the Pack. A component lists the files that belong to a given component and are relevant for a given project. The component itself or each individual file may refer to a condition that must resolve to true, otherwise the component or file is not applicable in the given context.

In the STM32Cube Package framework, the components are classified into four categories, Drivers, Middleware, Projects and Utilities, that are in turn split into subcategories, BSP drivers per board, HAL drivers per series, ST, third parties, as shown in Figure 28.

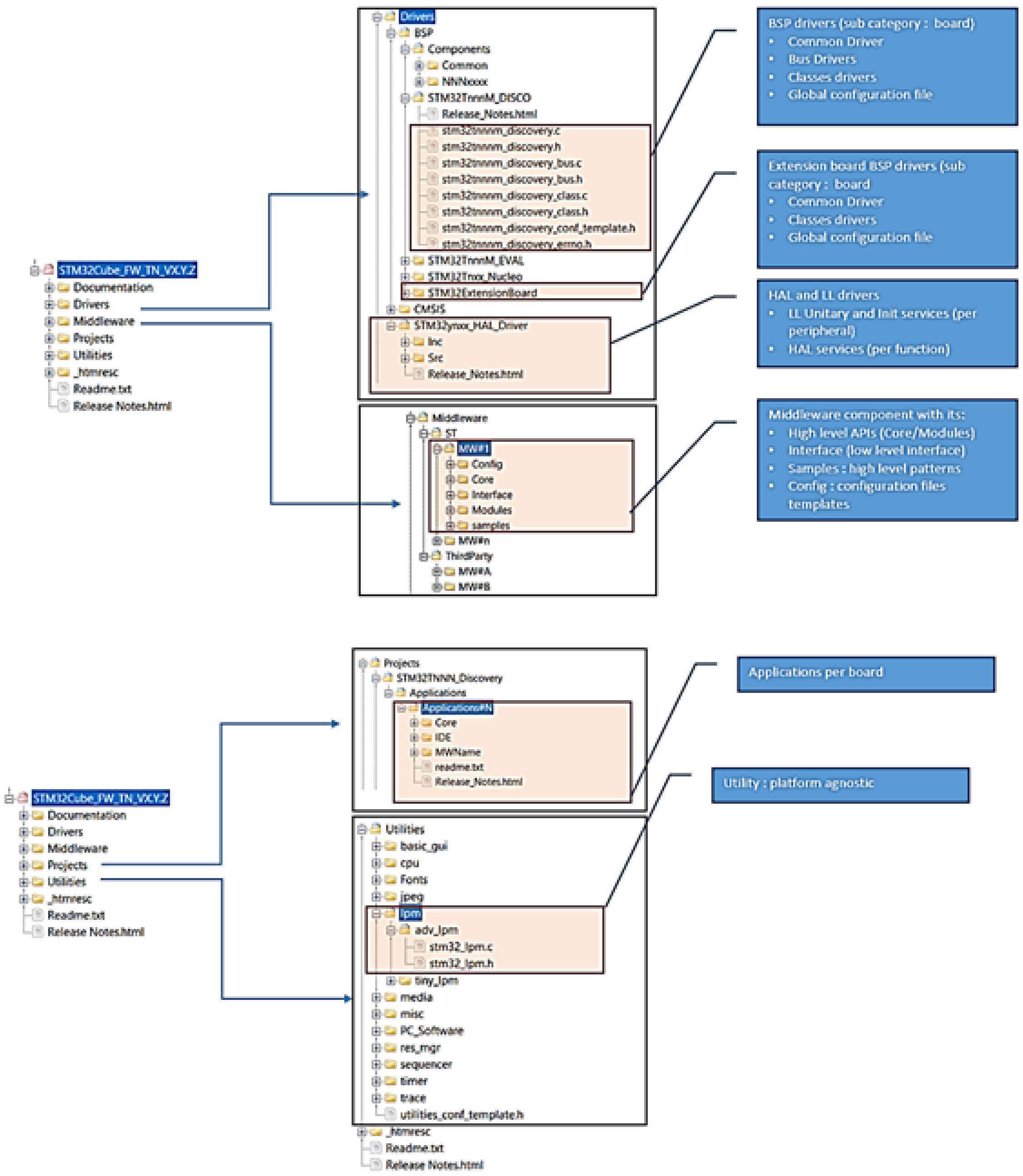


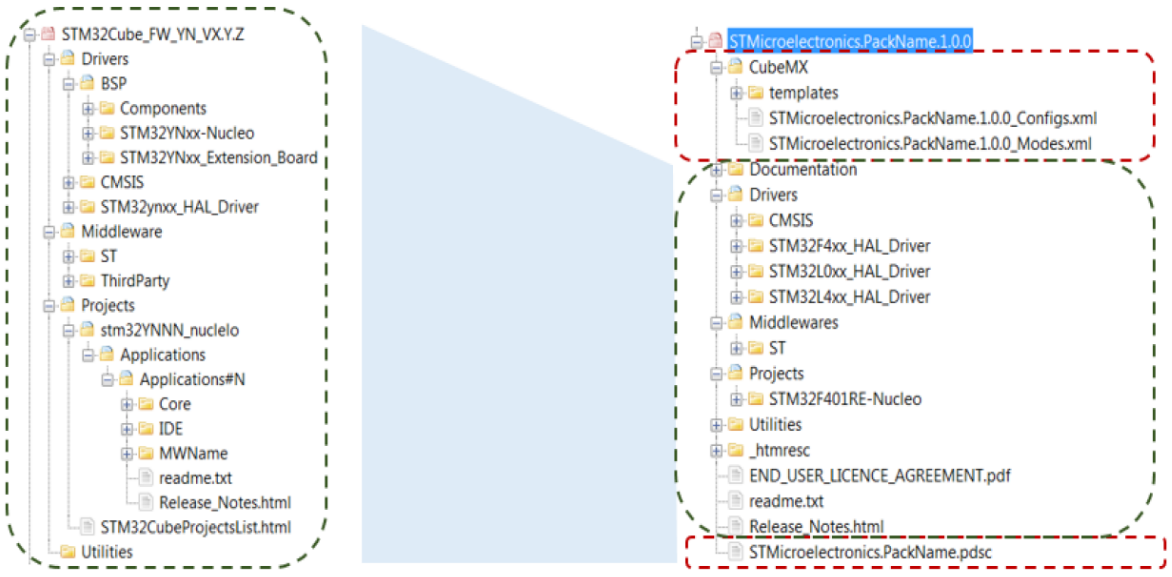
Figure 28. **CMSIS component architecture**

6.3 CMSIS Pack component location

A CMSIS Pack is set of firmware components that is described with an XML based package description (PDSC) file. It includes the user and device relevant parts of a file collection (named software Pack) with sources, header, and library files, documentation, Flash programming algorithms, source code templates, and example projects. Development tools and web infrastructures use the PDSC file to extract device parameters, software components, and evaluation board configurations.

The different Pack firmware components must be located as recommended by the global STM32Cube firmware architecture specification depending on the layer level. For example, when a Pack contains middleware layer components, the later must be located in the middleware folder of the STM32Cube MCU Package, either under the *ST* or *Third Party* subfolder, depending on the license (see Figure 29).

Figure 29. CMSIS Pack component location



Note: In an STM32Cube MCU Package, only one version per Pack component is allowed. As a result, the version must not be mentioned in the file and folder names but rather inside the Pack descriptor itself and the associated documents (readme.txt and release note).

6.4 Description of CMSIS Pack components

CMSIS Packs must describe and identify the different STM32Cube component categories and their respective subcategories through the components keywords. The following table summarizes the correspondence between a unitary Pack description and firmware components.

Unitary Pack	Firmware component	Granularity
HAL Pack	HAL and LL drivers	Single Pack for all the HAL/LL drivers for a given STM32 Series.
CMSIS driver Pack	CMSIS core and device drivers from Arm, repackaged by STMicroelectronics	CMSIS Pack for a given STM32 Series.
BSP component	BSP components drivers	Single Pack for each function set of BSP components. Example: • **Audio** • **MEMS** • **LCDController** • **TouchScreen** • **IOExpander**
BSP motherboard Pack	BSP common, bus and class drivers	Single Pack for each board containing all the class drivers.
BSP extension board Pack	BSP common and class drivers *Note: Bus drivers are generated by STM32CubeMX.*	Single Pack for each board containing all the class drivers.
Middleware Pack	Core, modules and interface files.	All core, modules, interface files for a given middleware.
Utility Pack	Utility components	Single Pack for each utility component.
Application Pack	Applications and examples	Single Pack for all the projects supported by a given board.

Each component must have a Class (Cclass=), a Group (Cgroup=), and a Version (Cversion=), which are used to identify the component. Optionally, a component may have Sub-Group (Csub=) and Variant (Cvariant=) additional categories. The Class, Group, Sub-Group, Variant and Version are used together with the vendor specified by the Pack to identify a given component. A component vendor must ensure that the combination of Class, Group, Sub-Group and Version is unique and not used by multiple components. The bundle (Cbundle) field is not required. However, multiple interdependent components belonging to the same Cclass can be grouped into a bundle.

6.4.1 Description of HAL drivers

Element usage

The following HAL component description is only applicable to HAL unitary Packs. In standalone Packs, the HAL is referenced from STM32CubeMX generated projects.

Table 3. **HAL driver element usage**

Element	Value	Examples
Cbundle	STM32Cube_{SERIE}_HAL_Drivers	STM32Cube_L5_HAL_Drivers
Cvendor	STMicroelectronics	-
Cclass	Device	Device
Cgroup	PPP[1]	For example ADC, PCD or RCC
Csub	HAL or LL	-
Capiversion	N/A	-
Cversion	x.y.z	0.5.0
Condition	HAL_Condition	CMSIS_Driver

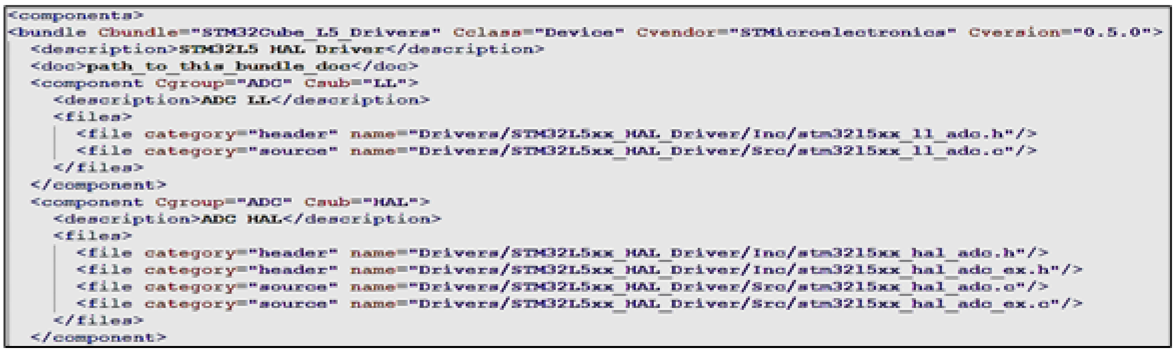
1. *Cgroup refers to the PPP HAL driver name and not to the physical peripheral driver, as defined in the STM32Cube specifications.*

Component path

STM32Cube_FW_TN_VX.Y.Z\Drivers\STM32tnxx_HAL_Driver\{Src/Inc}

PDSC format

Figure 30. **PDSC format for HAL drivers**



```
<components>
<bundle Cbundle="STM32Cube_L5_Drivers" Cclass="Device" Cvendor="STMicroelectronics" Cversion="0.5.0">
  <description>STM32L5 HAL Driver</description>
  <doc>path_to_this_bundle_doc</doc>
  <component Cgroup="ADC" Csub="LL">
    <description>ADC LL</description>
    <files>
      <file category="header" name="Drivers/STM32L5xx_HAL_Driver/Inc/stm32l5xx_ll_adc.h"/>
      <file category="source" name="Drivers/STM32L5xx_HAL_Driver/Src/stm32l5xx_ll_adc.c"/>
    </files>
  </component>
  <component Cgroup="ADC" Csub="HAL">
    <description>ADC HAL</description>
    <files>
      <file category="header" name="Drivers/STM32L5xx_HAL_Driver/Inc/stm32l5xx_hal_adc.h"/>
      <file category="header" name="Drivers/STM32L5xx_HAL_Driver/Inc/stm32l5xx_hal_adc_ex.h"/>
      <file category="source" name="Drivers/STM32L5xx_HAL_Driver/Src/stm32l5xx_hal_adc.c"/>
      <file category="source" name="Drivers/STM32L5xx_HAL_Driver/Src/stm32l5xx_hal_adc_ex.c"/>
    </files>
  </component>
```

6.4.2 Description of CMSIS drivers

Element usage

The CMSIS component elements and descriptions are defined in the same way as Arm native Packs. The STM32 start-up and device components have been introduced and the RTOS/RTOS2 subfolders and files reorganized as described below:

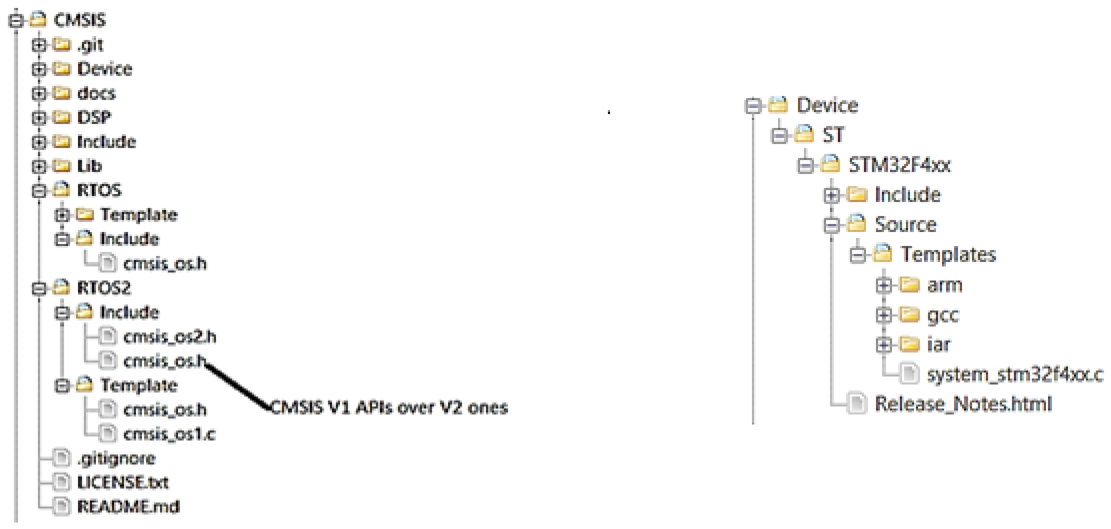
Table 4. **CMSIS driver element usage**

Element	Value	Csub	Cvariant
Cbundle	STM32Cube_CMSIS_Drivers	-	-
Cvendor	Arm	-	-
Cclass	Device	-	-
Cgroup	CORE	-	-
	Start-up	STM32{SERIE}: STM32YN	C Startup None
	Device	STM32{SERIE}: STM32YN	-
	DSP	-	Source Library
	NNLIB	-	-
	RTOS	-	None (include) Keil RTX Keil RTX5
	RTOS2	-	None (include) Keil RTX5
	OS Tick	Private Timer	-
		Generic Physical Timer	-
Capiversion	a.b.c	-	-
Cversion	x.y.z	-	-
Condition	CMSIS_{CgroupCondition}	-	-

Component path

STM32Cube_FW_TN_VX.Y.Z\Drivers\CMSIS\

Folder organization


```
CMSIS
├─ .git
├─ Device
├─ docs
├─ DSP
├─ Include
├─ Lib
├─ RTOS
│  ├─ Template
│  └─ Include
│     └─ cmsis_os.h
├─ RTOS2
│  ├─ Include
│  │  ├─ cmsis_os2.h
│  │  └─ cmsis_os.h      CMSIS V1 APIs over V2 ones
│  └─ Template
│     ├─ cmsis_os.h
│     └─ cmsis_os1.c
├─ .gitignore
├─ LICENSE.txt
└─ README.md

Device
└─ ST
   └─ STM32F4xx
      ├─ Include
      ├─ Source
      │  └─ Templates
      │     ├─ arm
      │     ├─ gcc
      │     ├─ iar
      │     └─ system_stm32f4xx.c
      └─ Release_Notes.html
```

Figure 31. **CMSIS driver folder organization**

6.4.3 Description of BSP components

Element usage

Table 5. BSP element usage

Element	Value	Examples/Dependencies
Cbundle	<BSP_Class> (refer to section 7.4.4.5)	• MEMS • GNSS • AUDIO…etc.
Cvendor	STMicroelectronics	
Cclass	Board part	
Cgroup	{FeaturesGroup}	• AudioCodec • WIFI • LCDController …etc
Csub	{component part number}	• lsm6dsl, lsm303agr, lis2dw12 • cs42l51
Cvariant	Used bus to link the component to the board	• I2C • SPI ...etc.
Capiversion	a.b.c	2.5.0
Cversion	x.y.z	6.2.0
Condition	{<ComponentP/N>_Condition}	Board BSP Bus driver HAL

Component path

STM32Cube_FW_TN_VX.Y.Z\Drivers\BSP\Components\NNNxxxx

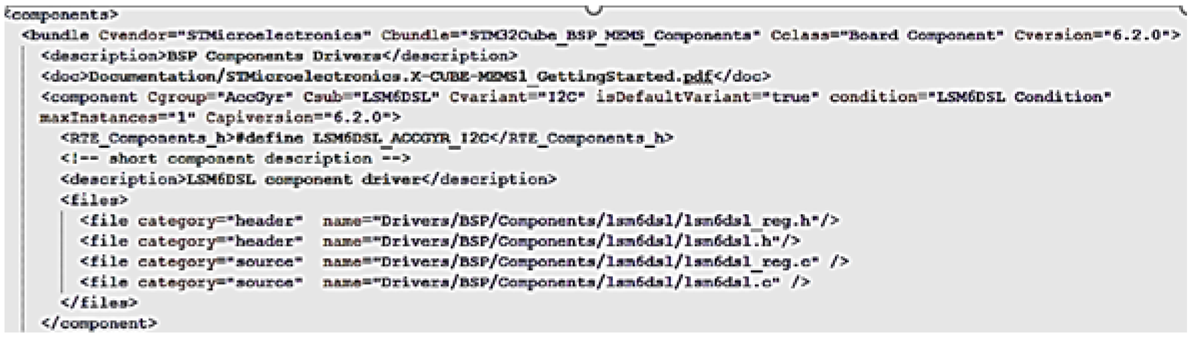
Cgroup list

Table 6. BSP CGroup list

Cgroup	Description
AudioCodec	Audio codec component (wm8994, cs42l51..etc)
WIFI	Wifi module component (esp8266, eswifi)
AccGyro	Accelerometer Gyroscope combo component (lsm6dsl)
GyroMag	Gyroscope magnetometer combo component
LCDController	LCD Controller component (st7735, otm8009a)
TouchScreen	Touchscreen controller component (ft5336, ts3510, exc7200)
IOExpander	IO Expander controller component (mfxstm32l152, stmpe811)
LCDDisplay	LCD display component (ampire480272, ampire640480, rk043fn48h)
ETHPhy	Ethernet Phy. controller component (lan8742)
Camera	Camera Controller component (ov9655, otm8009a)

PDSC format

Figure 32. PDSC format for BSP drivers



```
<components>
  <bundle Cvendor="STMicroelectronics" Cbundle="STM32Cube_BSP_MEMS_Components" Cclass="Board Component" Cversion="6.2.0">
    <description>BSP Components Drivers</description>
    <doc>Documentation/STMicroelectronics.X-CUBE-MEMS1_GettingStarted.pdf</doc>
    <component Cgroup="AccGyr" Csub="LSM6DSL" Cvariant="I2C" isDefaultVariant="true" condition="LSM6DSL Condition"
    maxInstances="1" Capiversion="6.2.0">
      <RTE_Components_h>#define LSM6DSL_ACCGYR_I2C</RTE_Components_h>
      <!-- short component description -->
      <description>LSM6DSL component driver</description>
      <files>
        <file category="header"  name="Drivers/BSP/Components/lsm6dsl/lsm6dsl_reg.h"/>
        <file category="header"  name="Drivers/BSP/Components/lsm6dsl/lsm6dsl.h"/>
        <file category="source"  name="Drivers/BSP/Components/lsm6dsl/lsm6dsl_reg.c" />
        <file category="source"  name="Drivers/BSP/Components/lsm6dsl/lsm6dsl.c" />
      </files>
    </component>
```

6.4.4 Description of BSP Class drivers

STMicroelectronics board element usage

Table 7. STMicroelectronics board element usage

Element	Value	Examples/Dependencies
Cbundle	STM32Cube_{SERIE}_BSP_Drivers	STM32Cube_L5_BSP_Drivers
Cvendor	STMicroelectronics	
Cclass	Board support	
Cgroup	{BoardName}	• STM32L5xx_Nucleo • STM32L552E-EVAL • STM32L562E-DK
Csub	{BSP Class}	• Audio • Common
Capiversion	a.b.c	2.5.0
Cversion	x.y.z	6.2.0
Condition	{<BoardName>_Condition}	Board Part HAL

Extension board element usage

Table 8. Extension board element usage

Element	Value	Examples
Cbundle	STM32Cube_Extension_BSP_Drivers	
Cvendor	STMicroelectronics	
Cclass	Board Extension	
Cgroup	{BoardName}	• IKS01A3
Csub	{BSP Class}	• Audio • Common

Element	Value	Examples
Capiversion	a.b.c	2.5.0
Cversion	x.y.z	6.2.0
Condition	{<BoardName>_Condition}	Board part HAL

Custom board element usage

Table 9. **Custom board element usage**

Element	Value	Examples/Dependencies
Cbundle	STM32Cube_Custom_BSP_Drivers	
Cvendor	STMicroelectronics	
Cclass	Board Support	Board Support
Cgroup	<UserBoardName>	• Custom
Csub	<BSP Class>	• Audio • Common
Capiversion	a.b.c	2.5.0
Cversion	x.y.z	6.2.0
Condition	<BoardName>_Condition	Board part HAL

Component path

STM32Cube_FW_TN_VX.Y.Z\Drivers\BSP\BoardName (see board naming rules below)

Csub list

Table 10. **Csub list for BSP Class drivers**

Cgroup	Description
Audio	Audio class
CAMERA	Camera class
MEMS	Generic sensors class
Env_sensor	Environement sensors class (Humidity, temperature, pressure)
Motion_sensor	Motion sensors class (magnetometer, accelerometer, gyroscope)
Pwr_mon	Power monitor class
TS	Touchscreen class
usb_typec_switch	USB type-C Switch class
Radio	Radio RF class (SubGHZ)
LCD	LCD class
IO	Input output extension class
IDD	Current and power measurement class

Cgroup	Description
EPD	E-paper display class
SD	SD memory class
SDRAM	SDRAM memory class
NOR	NOR flash driver
QSPI	QSPI memory class
SRAM	SRAM memory class
COMMON	Common class (COM, Buttons, LED, potentiometer)
EEPROM	EEPROM memory class

Board name rules

The following table summarizes the rulesfor naming boards as well as the difference between the part numbers and the name used in the firmware.

ST Board	Board Name	Filename	Folder Name
Evaluation board	STM32TnnnM-EV/EVAL	stm32tnnnm_eval Example: stm32f769i_eval	STM32TnnnM-EVAL Example: STMF769I-EVAL
Discovery board	(Product focus) STM32TnnnM-DK/DISCO	stm32tnnnm_discovery Example: stm32f769i_discovery	STM32TnnnM-DISCO/DK Example: STM32F769I-DISCO

	(application focus)
	B-TxxxM-AAAyyT(z)
	B-TxxxM-AAAAAA(y)
		b_txxxm_aaayyt(z)
		Example: b_l475e_iot01a1
		b_txxxm_aaaaaa(y)
		Example: b_l072z_lrwan1
			B-TxxxM-AAAyyT(z)
			Example : b-l475e-iot01a1
			B-TxxxM-AAAAAA(y)
			Example: b-l072z-lrwan1

Nucleo board	NUCLEO-TnnnMM	stm32tnxx_nucleo Example: stm32l4xx_nucleo	STM32TNxx_Nucleo Example: STM32L4xx_Nucleo
Extension board	{X/I}-NUCLEO-NNNNN	nnnnn Example: iks01a2	NNNNN Example: iks01a2

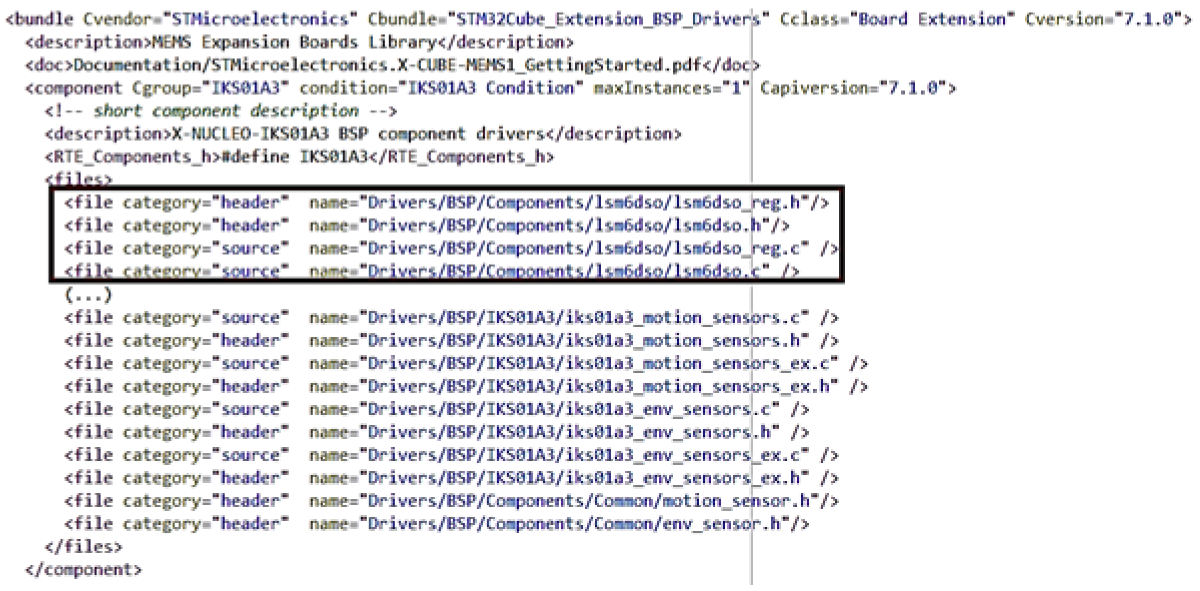
Note: *For Nucleo boards, the projects must refer to the exact board part number even if they use the same BSP drivers: STM32Cube_FW_TN_VX.Y.Z\Projects\STM32TNNN_Eval\Applications.*

Component driver management

In unitary Packs, BSP Class drivers (STMicroelectronics or extension boards) are based on the services of the component drivers using the CMSIS Pack conditions. As all the Pack managers, including STM32CubeMX, do not automatically resolve the dependencies (conditions), the users have to resolve them manually and select the required components when BSP drivers are used.

Standalone Packs propose an alternative implementation where the components are referenced internally. This enables hiding the components once a BSP component is selected. In this case the component files are part of the whole BSP component files and only the BSP bus services have to be resolved externally.

Figure 33. BSP Class driver: external component implementation



```
<bundle Cvendor="STMicroelectronics" Cbundle="STM32Cube_Extension_BSP_Drivers" Cclass="Board Extension" Cversion="7.1.0">
  <description>MEMS Expansion Boards Library</description>
  <doc>Documentation/STMicroelectronics.X-CUBE-MEMS1_GettingStarted.pdf</doc>
  <component Cgroup="IKS01A3" condition="IKS01A3 Condition" maxInstances="1" Capiversion="7.1.0">
    <!-- short component description -->
    <description>X-NUCLEO-IKS01A3 BSP component drivers</description>
    <RTE_Components_h>#define IKS01A3</RTE_Components_h>
    <files>
      <file category="header"  name="Drivers/BSP/Components/lsm6dso/lsm6dso_reg.h"/>
      <file category="header"  name="Drivers/BSP/Components/lsm6dso/lsm6dso.h"/>
      <file category="source"  name="Drivers/BSP/Components/lsm6dso/lsm6dso_reg.c" />
      <file category="source"  name="Drivers/BSP/Components/lsm6dso/lsm6dso.c" />
      (...)
      <file category="source"  name="Drivers/BSP/IKS01A3/iks01a3_motion_sensors.c" />
      <file category="header"  name="Drivers/BSP/IKS01A3/iks01a3_motion_sensors.h" />
      <file category="source"  name="Drivers/BSP/IKS01A3/iks01a3_motion_sensors_ex.c" />
      <file category="header"  name="Drivers/BSP/IKS01A3/iks01a3_motion_sensors_ex.h" />
      <file category="source"  name="Drivers/BSP/IKS01A3/iks01a3_env_sensors.c" />
      <file category="header"  name="Drivers/BSP/IKS01A3/iks01a3_env_sensors.h" />
      <file category="source"  name="Drivers/BSP/IKS01A3/iks01a3_env_sensors_ex.c" />
      <file category="header"  name="Drivers/BSP/IKS01A3/iks01a3_env_sensors_ex.h" />
      <file category="header"  name="Drivers/BSP/Components/Common/motion_sensor.h"/>
      <file category="header"  name="Drivers/BSP/Components/Common/env_sensor.h"/>
    </files>
  </component>
```

6.4.5 Description of middleware components

Middleware element usage

Table 11. Middleware element usage

Element	Value	Examples/Dependencies
Cbundle	MW Name	• BlueNRG-MS • LwIP
Cvendor	STMicroelectronics <Vendor> (Thirdparty)	
Cclass	Arm-defined Class	• Network
Cgroup	Middleware name	• LwIP
Csub	Middleware component as defined in the STM32Cube specification 1.2	• API • Core • Interface • Sample/Pattern(1)
Capiversion	a.b.c	2.0.3
Cversion	x.y.z	2.0.3
Condition	<Cgroup>_Condition	LL Driver PPP

1. *The Csub element must reflect the physical split of the middleware component folder. If no subfolder is defined, the Csub: Core must be used by default.*

Component path

STM32Cube_FW_TN_VX.Y.Z\Middleware\ST\MwName (Cgroup)
STM32Cube_FW_TN_VX.Y.Z\Middleware\ThirdParty\MwName (Cgroup)

Cclass list

Table 12. **Cclass list for middleware components**

Cclass	Description
Audio	Software components for audio processing
Device	Components containing device specific implementations of non-standard APIs (for example HAL drivers or CMSIS Startup files)
Data Exchange	Components implementing some kind of data exchange or data formatter
Extension Board	Drivers that support an extension board or shield
File System	Components implementing some kind of File Systems (for example Flash or RAM-based file systems)
Graphics	Components implementing some kind of Display and Graphics Software
IoT Client	Components implementing some kind of IoT cloud client connector
IoT Utility	IoT specific software utility
Network	Components implementing some kind of network communications (for example TCP/IP Stack)
Wireless	RF protocol based stacks (such as BLE and LoRa)
Sensors	Different MEMS sensor processing middlewares
RTOS	Components implementing some kind of real-time operating system (for example FreeRTOS, Micrium Real or Time Kernel)
Security	Components implementing some kind of encryption for secure communication or storage
USB	Components implementing some kind of USB interfaces (for example Host and Device interfaces)
Utility	Generic software utility components

6.4.6 Description of Utilities

Utility elements usage

Table 13. **Utility element usage**

Element	Value	Examples/Dependencies
Cbundle	STM32Cube_Utilities	
Cvendor	STMicroelectronics	
Cclass	Utility	
Cgroup	<UtilityGroupName>	• CPU • LPM • MISC
Csub	<UtilityName>	• List • Math • Mem
Capiversion	x.y.z	1.0.0
Cversion	a.b.c	1.0.0
Condition	<UtilityName>_Condition	HAL, Board specific (LCD)

Component path

STM32Cube_FW_TN_VX.Y.Z\Utilities\UtilityGroupName\UtilityName
STM32Cube_FW_TN_VX.Y.Z\Utilities\UtilityName (No Csub/UtilityGroupName defined)

Cgroup/Csub list

Table 14. **Cgroup/Csub list for Utilities**

Cgroup	Csub	Description
CPU	N/A	CPU load measurement utility
LPM	Advanced LPM	Advanced low-power manager utility
	Tiny LPM	Tiny low-power manager utility
MISC	List	Linked list utility
	Math	Mathematical services
	Mem	Memory manager utility
	Sys_time	Time handling services utility
	Tiny_scanf	Tiny standard I/O scanf utility
	tiny_vsnprintf	Tiny standard I/O vsprintf utility
	Strings	String operations utility
Sequencer	N/A	Bare metal scheduler utility
Timer	N/A	Time server utility
Trace	Adv_Trace	Advanced trace utility
	ITM_Trace	Instrumentation Trace Macrocell trace based utility
	LCD_Trace	LCD log/trace utility
ResMgr	N/A	Resource manager (multi core) utility
LCD	N/A	LCD basic drawing services utility
fonts	N/A	Fonts resources used by the LCD utility
JPEG	N/A	JPEG post processing utility
Media	Audio	Audio format files used for graphical applications and demonstrations
	Pictures	Pictures files in JPEG and BMP format used for Audio applications and demonstrations
	Video	Audio data used for Audio application and demonstrations
Software	Software name	PC software utility used for preprocessing, tests and monitoring

6.4.7 Description of applications

The PDSC provides an overview of the applications as well as the list of available projects. It is then up to the Code generation tool (STM32CubeMX) or to the Pack manager to describe the file organization and what must be generated in its internal configuration file.

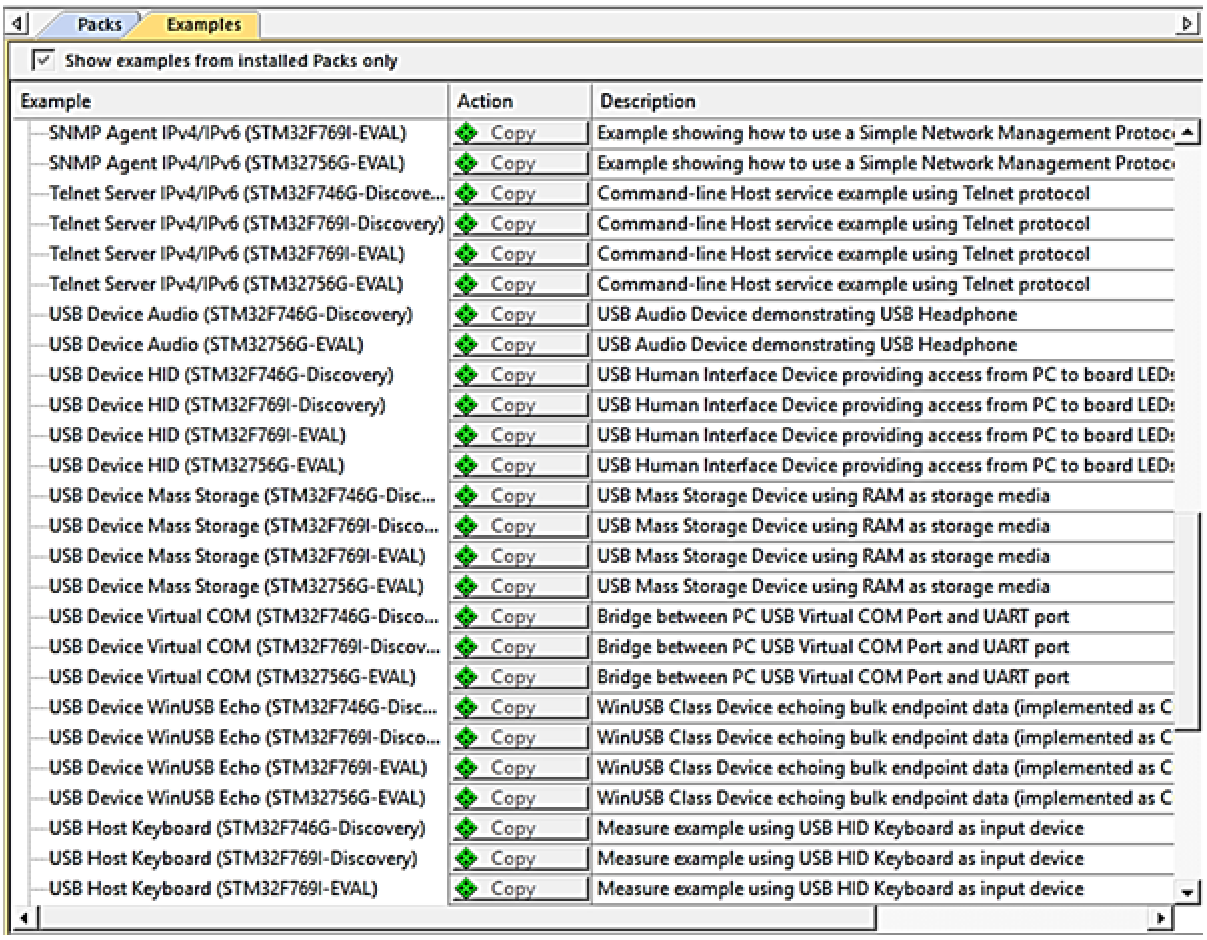
In standalone Packs, the application is generated by STM32CubeMX through the user interface and according the user configuration. The source files are either generated or referenced. They are described and listed in the STM32CubeMX Pack configuration file.

Unitary Packs

In unitary Packs, the projects (examples, applications and demonstration) are defined according to the example structure defined by Arm.

The */package/examples/example* element fully describes the examples contained in the Pack. Each example contains a list of files. The example itself and each individual file must refer to a *condition* that must resolve to true, otherwise the example or file is ignored. The board element is used to reference one or more board descriptions using the board vendor and name for which the example is targeted. Each example can specify attributes that list the related components using Class (Cclass=), Group (Cgroup=), Subgroup (Csub) and a Version (Cversion=).

Figure 34. **Arm CMSIS Pack project samples**



Packs | Examples

☑ Show examples from installed Packs only

Example	Action	Description
SNMP Agent IPv4/IPv6 (STM32F769I-EVAL)	Copy	Example showing how to use a Simple Network Management Protoc
SNMP Agent IPv4/IPv6 (STM32756G-EVAL)	Copy	Example showing how to use a Simple Network Management Protoc
Telnet Server IPv4/IPv6 (STM32F746G-Discove...	Copy	Command-line Host service example using Telnet protocol
Telnet Server IPv4/IPv6 (STM32F769I-Discovery)	Copy	Command-line Host service example using Telnet protocol
Telnet Server IPv4/IPv6 (STM32F769I-EVAL)	Copy	Command-line Host service example using Telnet protocol
Telnet Server IPv4/IPv6 (STM32756G-EVAL)	Copy	Command-line Host service example using Telnet protocol
USB Device Audio (STM32F746G-Discovery)	Copy	USB Audio Device demonstrating USB Headphone
USB Device Audio (STM32756G-EVAL)	Copy	USB Audio Device demonstrating USB Headphone
USB Device HID (STM32F746G-Discovery)	Copy	USB Human Interface Device providing access from PC to board LED
USB Device HID (STM32F769I-Discovery)	Copy	USB Human Interface Device providing access from PC to board LED
USB Device HID (STM32F769I-EVAL)	Copy	USB Human Interface Device providing access from PC to board LED
USB Device HID (STM32756G-EVAL)	Copy	USB Human Interface Device providing access from PC to board LED
USB Device Mass Storage (STM32F746G-Disc...	Copy	USB Mass Storage Device using RAM as storage media
USB Device Mass Storage (STM32F769I-Disco...	Copy	USB Mass Storage Device using RAM as storage media
USB Device Mass Storage (STM32F769I-EVAL)	Copy	USB Mass Storage Device using RAM as storage media
USB Device Mass Storage (STM32756G-EVAL)	Copy	USB Mass Storage Device using RAM as storage media
USB Device Virtual COM (STM32F746G-Disco...	Copy	Bridge between PC USB Virtual COM Port and UART port
USB Device Virtual COM (STM32F769I-Discov...	Copy	Bridge between PC USB Virtual COM Port and UART port
USB Device Virtual COM (STM32756G-EVAL)	Copy	Bridge between PC USB Virtual COM Port and UART port
USB Device WinUSB Echo (STM32F746G-Disc...	Copy	WinUSB Class Device echoing bulk endpoint data (implemented as C
USB Device WinUSB Echo (STM32F769I-Disco...	Copy	WinUSB Class Device echoing bulk endpoint data (implemented as C
USB Device WinUSB Echo (STM32F769I-EVAL)	Copy	WinUSB Class Device echoing bulk endpoint data (implemented as C
USB Device WinUSB Echo (STM32756G-EVAL)	Copy	WinUSB Class Device echoing bulk endpoint data (implemented as C
USB Host Keyboard (STM32F746G-Discovery)	Copy	Measure example using USB HID Keyboard as input device
USB Host Keyboard (STM32F769I-Discovery)	Copy	Measure example using USB HID Keyboard as input device
USB Host Keyboard (STM32F769I-EVAL)	Copy	Measure example using USB HID Keyboard as input device

Example 1 :

```
<examples>
<example name="Blinky" folder=" Projects\BoardName" doc="Abstract.txt" version="1.0">
<description>This is a basic example demonstrating the development flow and letting the LED
on the board blink</description>
<board vendor="STMicroelectronics" name="32F429IDISCOVERY"/>
<project>
<environment name="uv" load="ARM/Blinky.uvproj"/>
<environment name="iar" load="IAR/Blinky.ewarm" />
</project>
<attributes>
<component Cclass="CMSIS" Cgroup="Core"/>
<component Cclass="Device" Cgroup="Startup"/>
<keyword>Blinky</keyword>
<keyword>Getting Started</keyword>
</attributes>
</example>
</examples>
```

Standalone Packs

Table 15. **Application element usage**

Element	Value	Examples/Dependencies
Cbundle	<PackName>_Applications	
Cvendor	STMicroelectronics	
Cclass	Device	
Cgroup	Application category	• Demonstration • Application • Examples
Variant	<AppName>	• LWIP_WebServer
Capiversion		
Cversion	a.b.c	1.0.0
Condition	<AppName>_Condition	Applicative modules

In standalone Packs, the application is generated by STM32CubeMX through the user interface and according to the user configuration. As a result, only the top-level information of the application(s) are provided in the PSDC.

Example 2 :

```
<bundle Cvendor="STMicroelectronics" Cbundle="MEMS1_Application" Cclass="Device"
Cversion="7.0.0">
      <description>MEMS library applications</description>
      <doc>Documentation/STMicroelectronics.X-CUBE-MEMS1_GettingStarted.pdf</doc>
      <component  Cgroup="Application" Cvariant="IKS01A3_DataLogTerminal"
isDefaultVariant="true" condition="IKS01A3 Examples Condition" maxInstances="1">
        <!-- short component description -->
        <description>DataLogTerminal sample application for X-NUCLEO-IKS01A3</description>
        <RTE_Components_h>#define IKS01A3_DATALOGTERMINAL_DEMO</RTE_Components_h>
        <files/>
      </component>
(…)
</bundle>
```

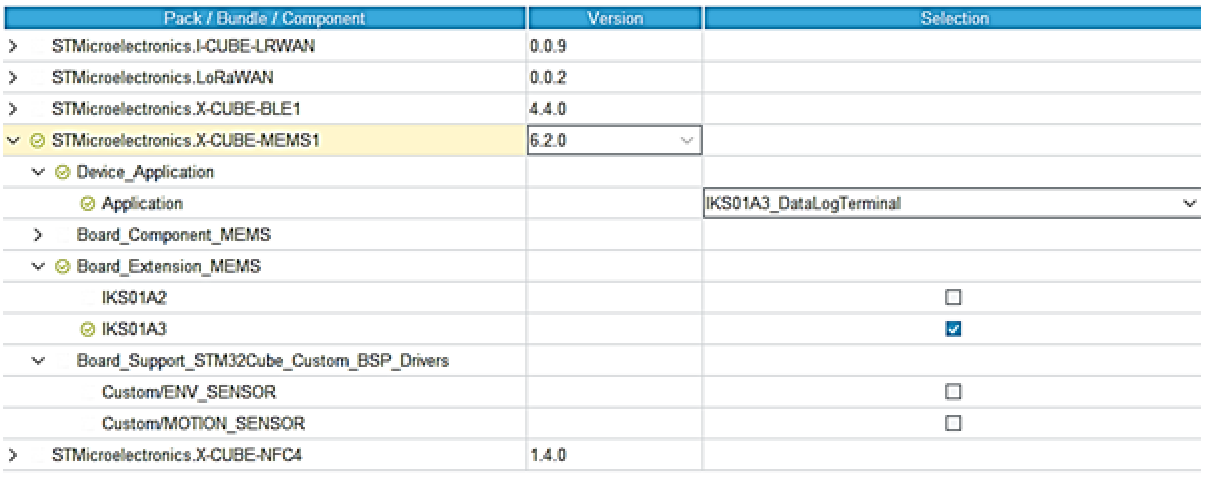
Note: The application element is only supported by STM32Cube and it is not seen by external or third-party Pack managers.

7 STM32Cube standalone Packs

An standalone Pack is mainly composed of a set of components offering a services that fulfil a common functionality (such as BSP Class or Middleware Class). Function Packs are typically:

- Packs based on a BSP driver high-level APIs and containing a set of applications built around the BSP Class services (such as MEMS Pack).
- Packs based on a single middleware class with a set of applications relative to the middleware class (such as Wireless or File system) and containing optionally a set of utilities and driver components required either by the middleware low-level interfaces or by the applications (such as Lora Pack).

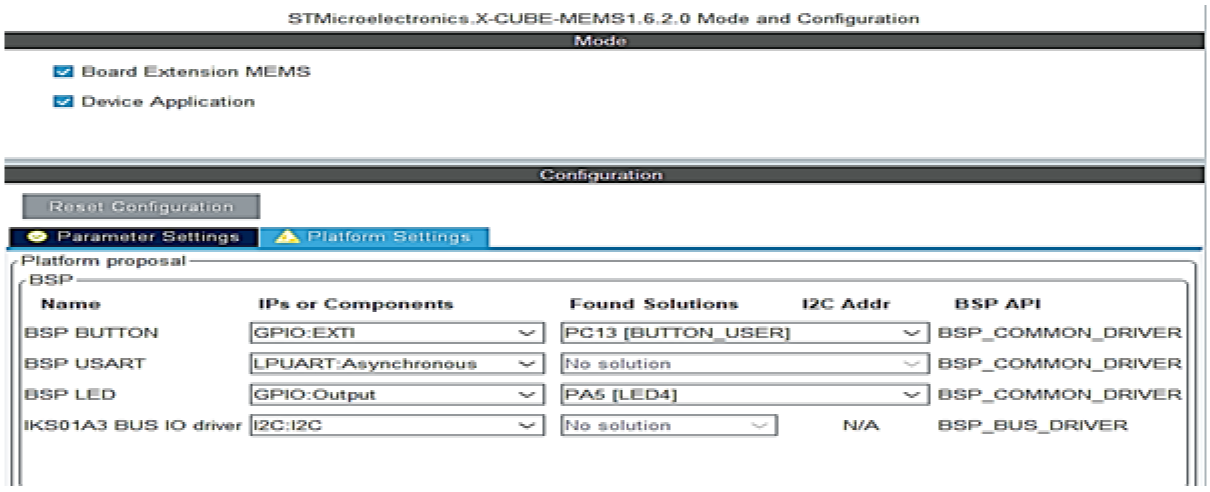
Figure 35. Selecting function Packs in STM32CubeMX



Pack / Bundle / Component	Version	Selection
> STMicroelectronics.I-CUBE-LRWAN	0.0.9	
> STMicroelectronics.LoRaWAN	0.0.2	
> STMicroelectronics.X-CUBE-BLE1	4.4.0	
⌄ STMicroelectronics.X-CUBE-MEMS1	6.2.0	
⌄ Device_Application		
Application		IKS01A3_DataLogTerminal
> Board_Component_MEMS		
⌄ Board_Extension_MEMS		
IKS01A2		☐
IKS01A3		☑
⌄ Board_Support_STM32Cube_Custom_BSP_Drivers		
Custom/ENV_SENSOR		☐
Custom/MOTION_SENSOR		☐
> STMicroelectronics.X-CUBE-NFC4	1.4.0	

After adding the function Pack in STM32CubeMX, it might be required to configure the different components through the parameter setting and platform setting menus, as shown below.

Figure 36. Configuring components in STM32CubeMX



STMicroelectronics.X-CUBE-MEMS1.6.2.0 Mode and Configuration

Mode

☑ Board Extension MEMS
☑ Device Application

Configuration

Reset Configuration

Parameter Settings | Platform Settings

Platform proposal

BSP

Name	IPs or Components	Found Solutions	I2C Addr	BSP API
BSP BUTTON	GPIO:EXTI	PC13 [BUTTON_USER]		BSP_COMMON_DRIVER
BSP USART	LPUART:Asynchronous	No solution		BSP_COMMON_DRIVER
BSP LED	GPIO:Output	PA5 [LED4]		BSP_COMMON_DRIVER
IKS01A3 BUS IO driver	I2C:I2C	No solution	N/A	BSP_BUS_DRIVER

STM32CubeMX then generates the selected application project with the required components, according to the conditions, the interface files and the virtual components (if any) and the selected platform.

7.1 Application architecture

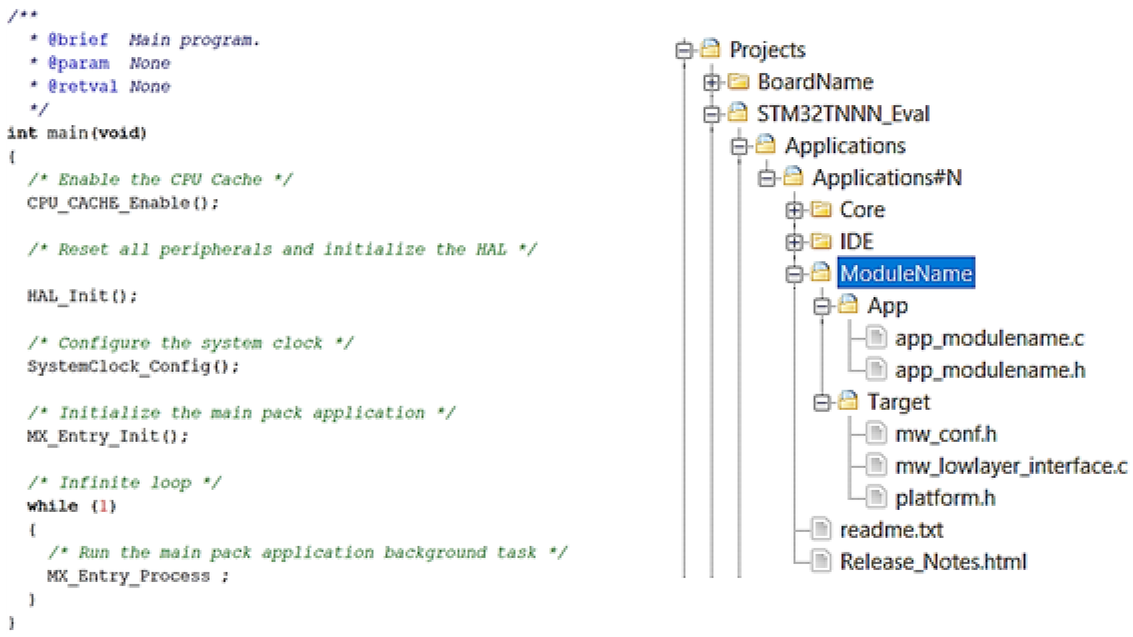
The applications are following a modular architecture reflected in the folder organization and based on the advanced structure. Applications come with a simple reduced set of high-level APIs. The *main.c* file is located in the *core* folder and calls the application entry APIs:

- *void MX_Entry_Init (void)*
- *void MX_Entry_Process (void)*

The content of the entry APIs (*MX_Entry_Init ()* and *MX_Entry_Process ()*) is defined by the Pack designer and provided in the Pack application template inside the Pack. The template is used by STM32CubeMX to copy the entry APIs in the main file.

The modules services are hidden by the above entry APIs that are called from the core, as described below:

Figure 37. **Module folders and calls**



```
/**
  * @brief  Main program.
  * @param  None
  * @retval None
  */
int main(void)
{
  /* Enable the CPU Cache */
  CPU_CACHE_Enable();

  /* Reset all peripherals and initialize the HAL */

  HAL_Init();

  /* Configure the system clock */
  SystemClock_Config();

  /* Initialize the main pack application */
  MX_Entry_Init();

  /* Infinite loop */
  while (1)
  {
    /* Run the main pack application background task */
    MX_Entry_Process ;
  }
}
```

Note:
1. *The pack application services are centralized in a single entry point that manages the internal modules services.*
2. *The MX_Entry_Process () can be omitted for some applications where the processes are either managed by an RTOS or handled in an asynchronous way through interrupts.*

7.2 Module definition

A module is a software component or a part of a program that contains one or more routines. One or several independently developed modules make up a program. A high-level software application may contain several different modules, each module serving unique separate functional operations.

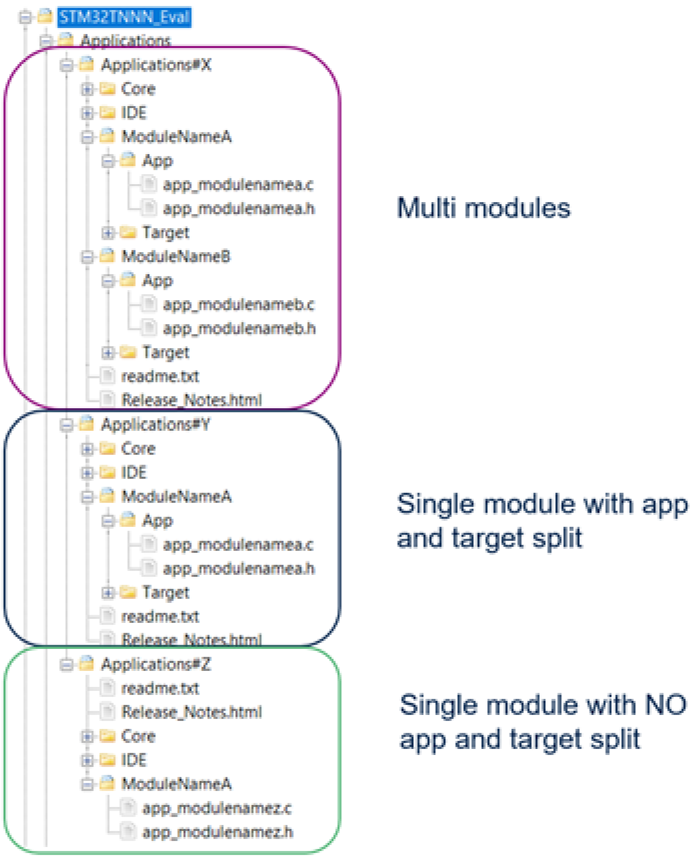
Modules make programmer's job easier by allowing the programmer to focus on one single area of the software application functionality. Modules are typically incorporated into the program (software) through interfaces.

As defined in the STM32Cube firmware specification, the interface defines how other modules can use that module. Note that more than one interface can be defined for a given module to allow identifying which services are offered to the other modules. In the application folder structure, the advanced folder model has to be used:

- *Core* folder : main.c/h stm32nnxx_it.c/h, HAL config and msp files
- *ModuleName/App* folder: platform independent application files relative to the module.

- *ModuleName/target* folder: platform dependent applications files (such as middleware interfaces or configuration files) relative to the module.

Figure 38. **Application folder structure**



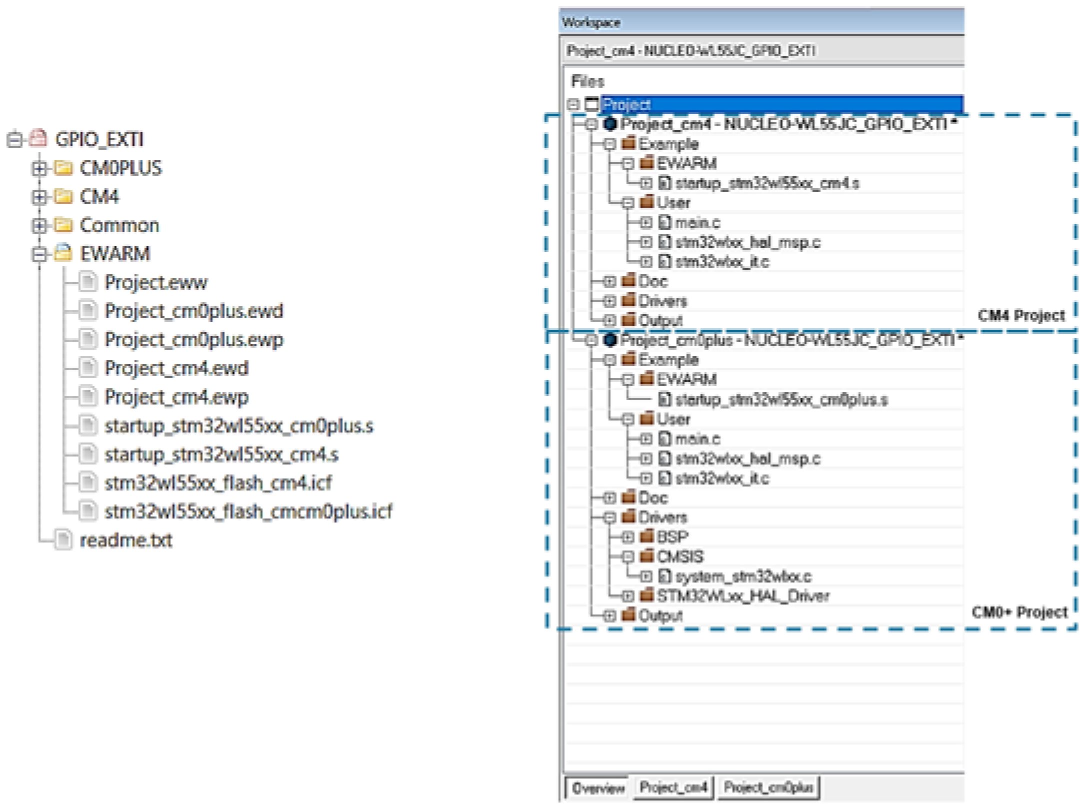
7.3 Multicore considerations

Dual-core projects are organized in one single workspace file (*.eww), two project files (*.ewp) and two debugger settings files (*.ewd). The cores related subproject and project file names must be postfixed with "_cmx" and "cmy", respectively, as listed below:

- *Project.eww*
- *Project_cmx.ewp*
- *Project_cmy.ewp*
- *Project_cmx.ewd*
- *Project_cmy.ewd*
- *stm32ynnnxx_flash{sram}_cmx.icf*
- *stm32ynnnxx_flash{sram}_cmy.icf*
- *startup_stm32ynnnxx_cmx.s*
- *startup_stm32ynnnxx_cmy.s*

The dual-core subprojects (CMx and CMy) for a given project are visible in the same IDE instances, thus the debugger can be launched at the same time for the two cores subprojects in a single workspace.Figure 39 shows the multicore project structure for the STM32WL Series:

Figure 39. **Example of multicore project structure for STM32WL**



```
GPIO_EXTI
├─ CM0PLUS
├─ CM4
├─ Common
├─ EWARM
│  ├─ Project.eww
│  ├─ Project_cm0plus.ewd
│  ├─ Project_cm0plus.ewp
│  ├─ Project_cm4.ewd
│  ├─ Project_cm4.ewp
│  ├─ startup_stm32wl55xx_cm0plus.s
│  ├─ startup_stm32wl55xx_cm4.s
│  ├─ stm32wl55xx_flash_cm4.icf
│  └─ stm32wl55xx_flash_cmcm0plus.icf
└─ readme.txt
```

```
Workspace
Project_cm4 - NUCLEO-WL55JC_GPIO_EXTI
Files
Project
├─ Project_cm4 - NUCLEO-WL55JC_GPIO_EXTI *
│  ├─ Example
│  │  ├─ EWARM
│  │  │  └─ startup_stm32wl55xx_cm4.s
│  │  └─ User
│  │     ├─ main.c
│  │     ├─ stm32wlxx_hal_msp.c
│  │     └─ stm32wlxx_it.c
│  ├─ Doc
│  ├─ Drivers
│  └─ Output                                        CM4 Project
└─ Project_cm0plus - NUCLEO-WL55JC_GPIO_EXTI *
   ├─ Example
   │  ├─ EWARM
   │  │  └─ startup_stm32wl55xx_cm0plus.s
   │  └─ User
   │     ├─ main.c
   │     ├─ stm32wlxx_hal_msp.c
   │     └─ stm32wlxx_it.c
   ├─ Doc
   ├─ Drivers
   │  ├─ BSP
   │  ├─ CMSIS
   │  │  └─ system_stm32wlxx.c
   │  └─ STM32WLxx_HAL_Driver
   └─ Output                                        CM0+ Project
Overview | Project_cm4 | Project_cm0plus
```

Multicores source files are duplicated for each core leading to two different binaries and sources split into two folders (*CORE_CMx* and *CORE_CMy*). When only one core is running (the other core being off after reset), the CMx folder level can be omitted as only the master core source files are used (single-core structure).

Multicores project source files are provided for each core. However, the two projects might share some common code (such as IPCC, Clock or System Configuration). In this case, an additional folder can be added at CMx folder level. It must contain several subfolders to group together the functional sources in an explicit way. Refer to Figure 41 and Figure 40 for examples.

Figure 40. **Multicore project structure with common files - example 1**

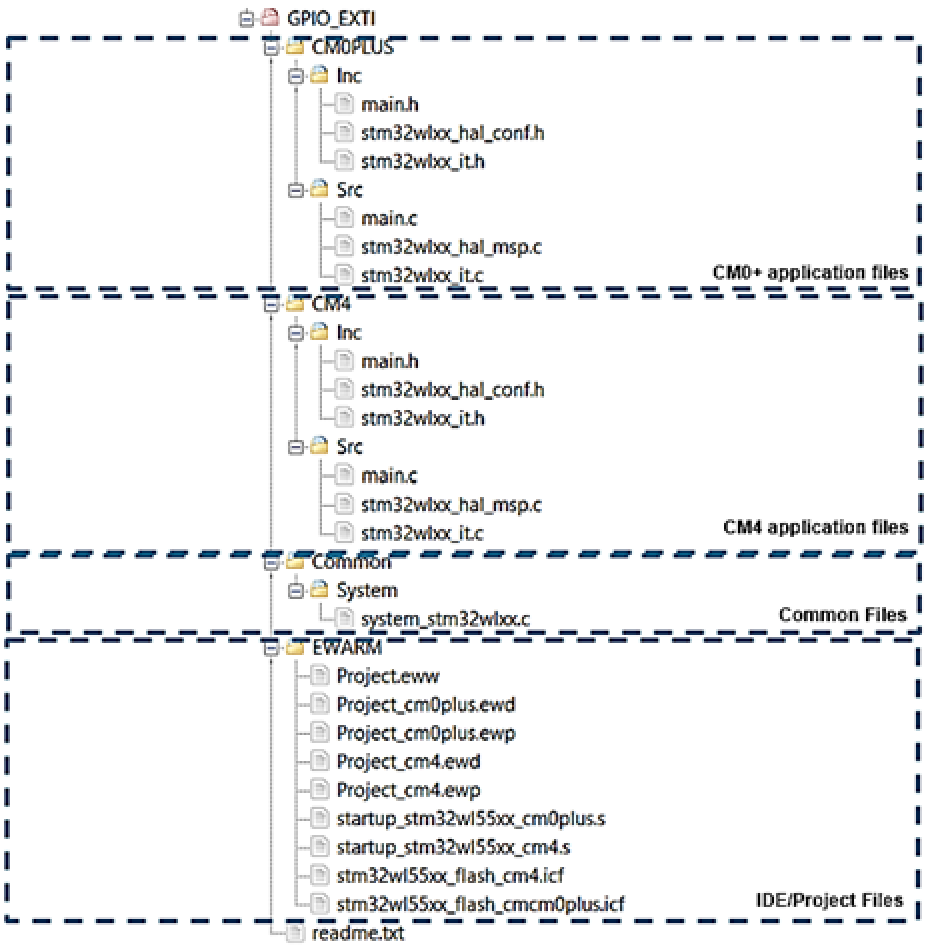
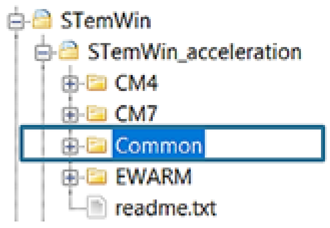
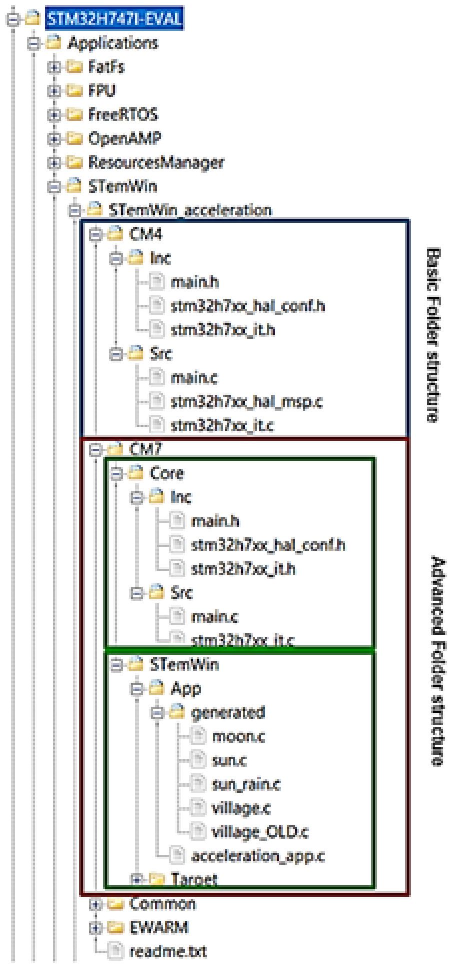

```
GPIO_EXTI
├─CM0PLUS
│ ├─Inc
│ │ ├─main.h
│ │ ├─stm32wlxx_hal_conf.h
│ │ └─stm32wlxx_it.h
│ └─Src
│   ├─main.c
│   ├─stm32wlxx_hal_msp.c
│   └─stm32wlxx_it.c
├─CM4
│ ├─Inc
│ │ ├─main.h
│ │ ├─stm32wlxx_hal_conf.h
│ │ └─stm32wlxx_it.h
│ └─Src
│   ├─main.c
│   ├─stm32wlxx_hal_msp.c
│   └─stm32wlxx_it.c
├─Common
│ └─System
│   └─system_stm32wlxx.c
├─EWARM
│ ├─Project.eww
│ ├─Project_cm0plus.ewd
│ ├─Project_cm0plus.ewp
│ ├─Project_cm4.ewd
│ ├─Project_cm4.ewp
│ ├─startup_stm32wl55xx_cm0plus.s
│ ├─startup_stm32wl55xx_cm4.s
│ ├─stm32wl55xx_flash_cm4.icf
│ └─stm32wl55xx_flash_cmcm0plus.icf
└─readme.txt
```

Figure 41. **Multicore project structure with common files - example 2**


```
STemWin
└─STemWin_acceleration
  ├─CM4
  ├─CM7
  ├─Common
  ├─EWARM
  └─readme.txt
```

Note:
- *CORE_CMx and CORE_CMy define statements need to be added in the preprocessor symbols.*
- *The codes running on the two cores can be cooperative or non-cooperative, meaning that they can either communicate together and share the applications processes or completely work in a standalone way without interacting.*
- *When the codes running on each core are not cooperative, they must not alter each other. For example, they must not share and use common resources without resource management policy.*
- *When one single core is running (when the other core is off after Reset), the CMx folder level can be omitted as only the Master core sources files are used.*
- *When the advanced folder structure is applied (application level), the repository model is applied for each core when need be (see Figure 42).*

***Figure 42.* Example of multicore-project advanced-folder structure**



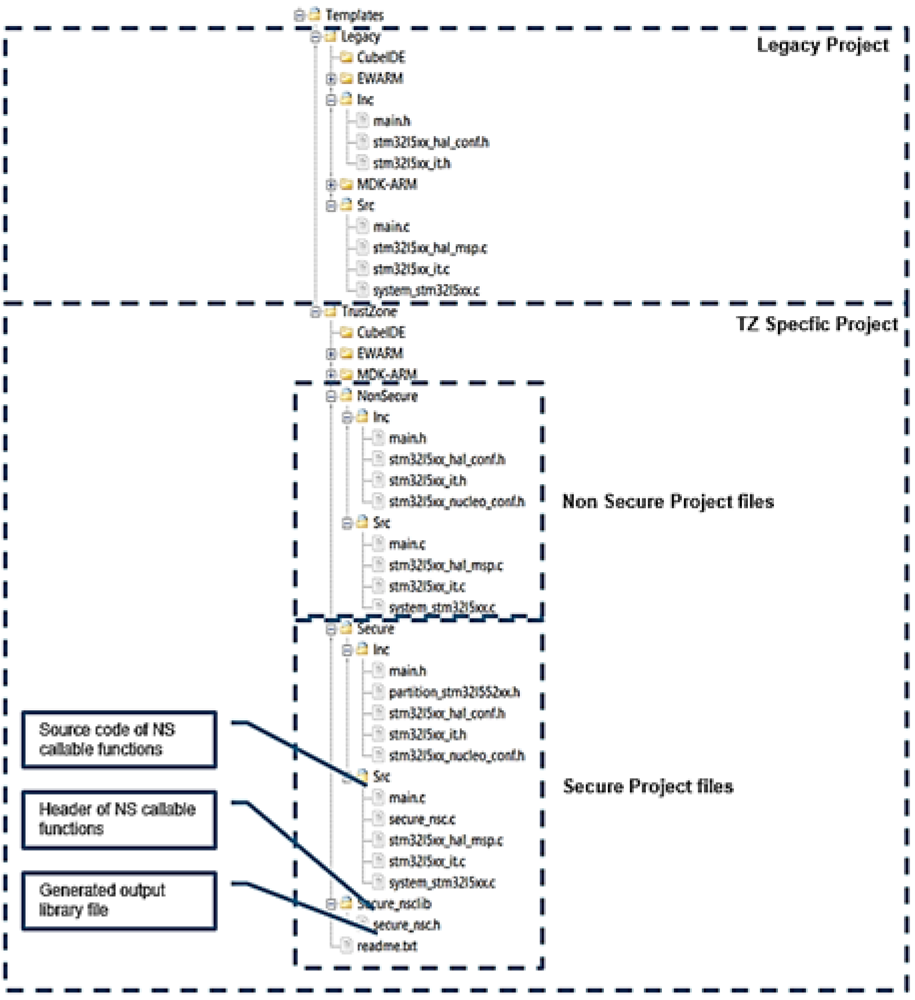
```
STM32H747I-EVAL
  Applications
    FatFs
    FPU
    FreeRTOS
    OpenAMP
    ResourcesManager
    STemWin
      STemWin_acceleration
        CM4
          Inc
            main.h
            stm32h7xx_hal_conf.h
            stm32h7xx_it.h
          Src
            main.c
            stm32h7xx_hal_msp.c
            stm32h7xx_it.c
        CM7
          Core
            Inc
              main.h
              stm32h7xx_hal_conf.h
              stm32h7xx_it.h
            Src
              main.c
              stm32h7xx_it.c
          STemWin
            App
              generated
                moon.c
                sun.c
                sun_rain.c
                village.c
                village_OLD.c
              acceleration_app.c
            Target
        Common
        EWARM
        readme.txt
```

7.4 Arm®TrustZone® considerations

In Arm®TrustZone® core-based firmware packages, the projects are enriched with examples, applications and demonstrations for Arm®TrustZone®. These new projects are added under directories with the name '*TrustZone*. They are targeted to be executed on devices where security is enabled (TZEN = 1) contrary to legacy projects with no security (TZEN = 0).

The projects are provided in legacy and Trustzone subprojects:
- The legacy project follows the single core project architecture.
- The secure project generates the non-secure callable library file, *secure_nsclib.o*. Note that the *secure_nsclib.h* header file and *secure_nsclib.o* library file are available for the non-secure project to include non-secure callable function prototypes and link with a non-secure callable entry point, respectively.

Figure 43. TrustZone® project high-level folder structure

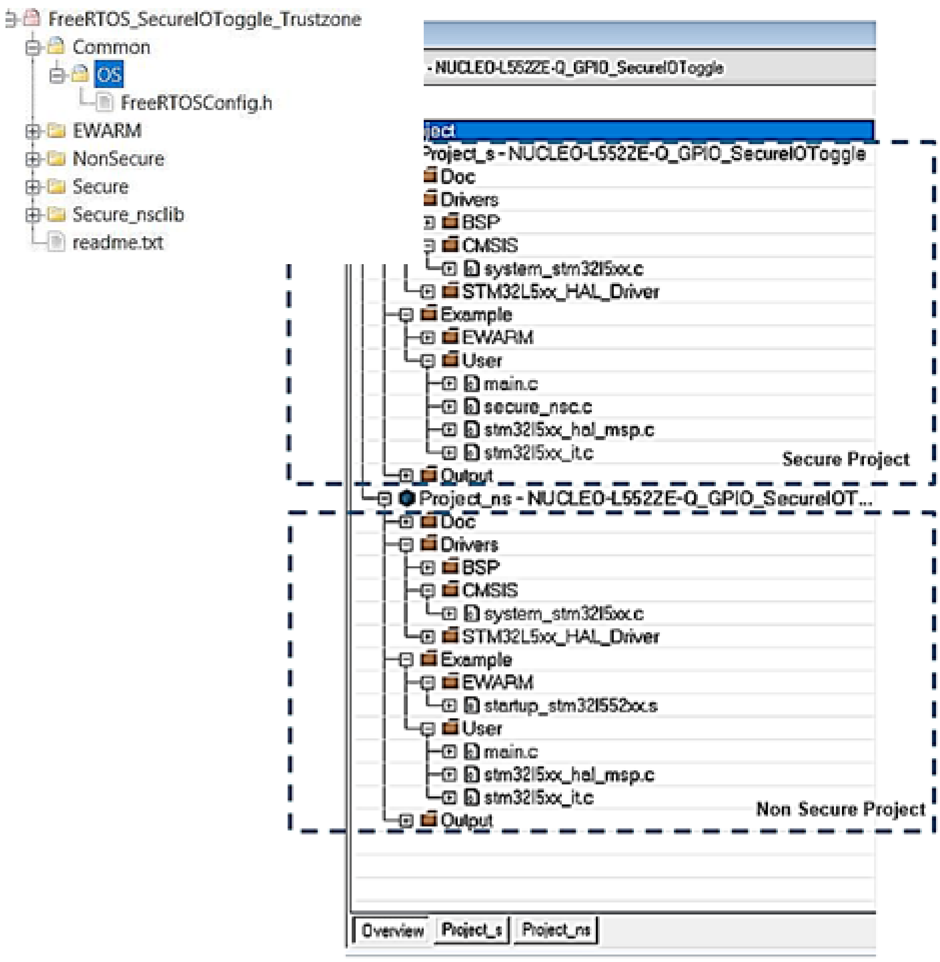


From project settings point of view, the TrustZone® projects are organized into one single workspace file (*.eww), one single startup file, two project file (*.ewp) and two debugger settings file (*.ewd). The secure and non secure sub-project related project files must be postfixed with the “_s” and “_ns” postfix, respectively:

- *Project.eww*
- *Project_s.ewp*
- *Project_ns.ewp*
- *Project_s.ewd*
- *Project_ns.ewd*
- *stm32ynnnxx_flash{sram}_s.icf*
- *stm32ynnnxx_flash{sram}_ns.icf*
- *startup_stm32ynnnxx.s*

The TrustZone® secure and non-secure subprojects for a given project are visible in the same IDE instance, and thus the debugger can be launched at the same time for the secure and the non secure subprojects in the same IDE instance.

Figure 44. Example of TrustZone® project folder structure for STM32L5

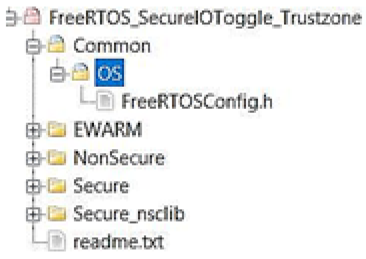


```
FreeRTOS_SecureIOToggle_Trustzone
  Common
    OS
      FreeRTOSConfig.h
  EWARM
  NonSecure
  Secure
  Secure_nsclib
  readme.txt
```

```
- NUCLEO-L552ZE-Q_GPIO_SecureIOToggle
ject
Project_s - NUCLEO-L552ZE-Q_GPIO_SecureIOToggle
  Doc
  Drivers
    BSP
    CMSIS
      system_stm32l5xx.c
    STM32L5xx_HAL_Driver
  Example
    EWARM
    User
      main.c
      secure_nsc.c
      stm32l5xx_hal_msp.c
      stm32l5xx_it.c
  Output
Project_ns - NUCLEO-L552ZE-Q_GPIO_SecureIOT...
  Doc
  Drivers
    BSP
    CMSIS
      system_stm32l5xx.c
    STM32L5xx_HAL_Driver
  Example
    EWARM
      startup_stm32l552xx.s
    User
      main.c
      stm32l5xx_hal_msp.c
      stm32l5xx_it.c
  Output
```

The TrustZone® projects source files are provided for secure and non-secure projects. However, the two projects might share some common code (such as OS services or System). In this case, an additional folder can be added at the IDE folder level. It must contains several subfolders to group together the functional sources in explicit way.

Figure 45. TrustZone® project folder structure with common files



```
FreeRTOS_SecureIOToggle_Trustzone
  Common
    OS
      FreeRTOSConfig.h
  EWARM
  NonSecure
  Secure
  Secure_nsclib
  readme.txt
```

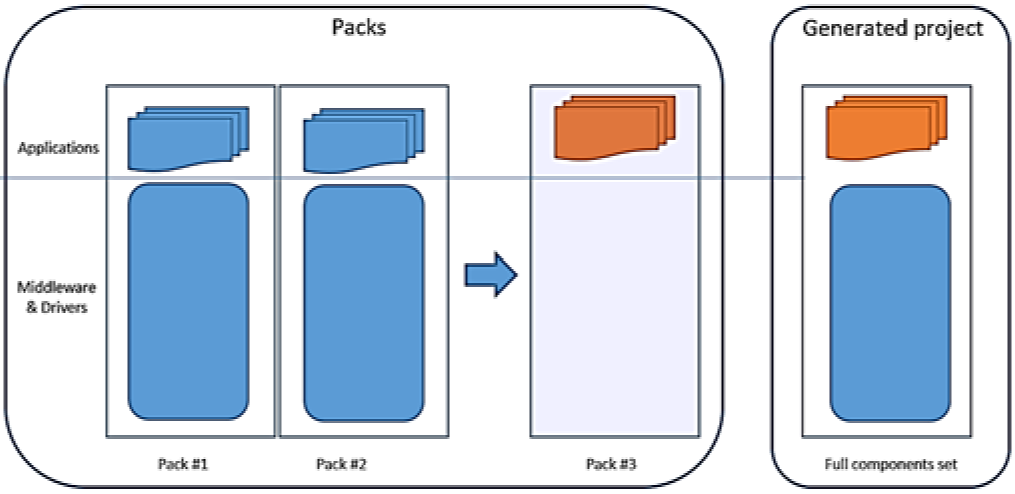
8 STM32Cube multipacks

8.1 Overview of STM32Cube multipacks

Several Packs can be configured together through STM32CubeMX to build an application based on the different components available in each Pack and enrich the applicative code. However, as the applications are always exclusive, putting the different Packs together prevents from directly using the applications initially integrated in the two initial function Packs.

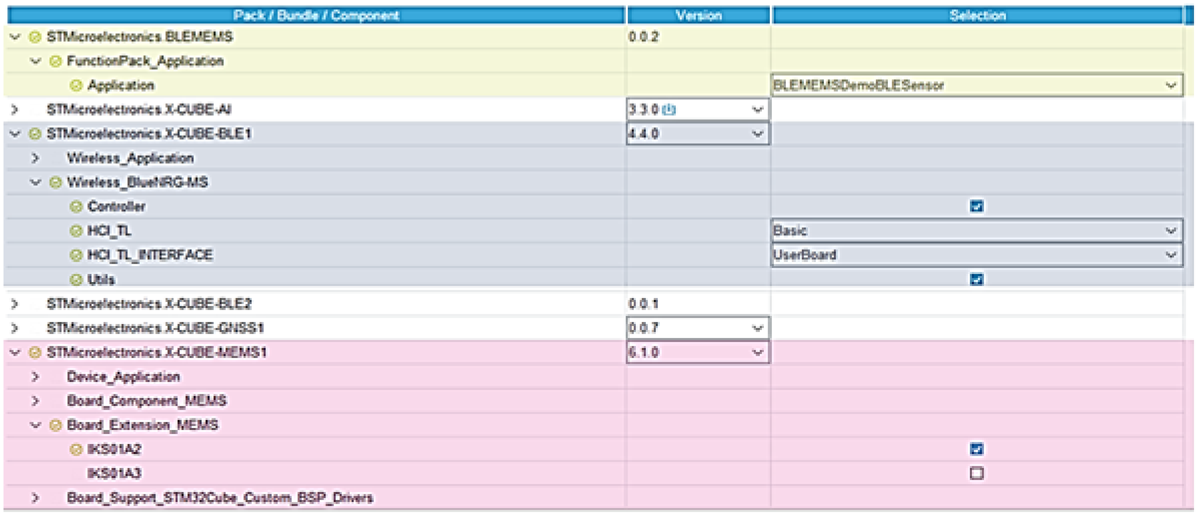
The new application Pack built on the two initial Packs must come with its own application set, while the remaining firmware components are inherited from the initials Packs.

Figure 46. **Building multipacks**



This approach prevents from having Pack interdependencies at application level and handling the common system and BSP resources for each Pack. The multipack integration leads to the creation of a new applicative code (new Pack) with a dependency with the two initial Function Pack components resolved through intrinsic conditions. The following example shows how MEMS and BLE standalone Packs can coexist.

Figure 47. **Example of MEMS and BLE coexistence**



Pack / Bundle / Component	Version	Selection
STMicroelectronics.BLEMEMS	0.0.2	
FunctionPack_Application		
Application		BLEMEMSDemoBLESensor
STMicroelectronics.X-CUBE-AI	3.3.0	
STMicroelectronics.X-CUBE-BLE1	4.4.0	
Wireless_Application		
Wireless_BlueNRG-MS		
Controller		☑
HCI_TL		Basic
HCI_TL_INTERFACE		UserBoard
Utils		☑
STMicroelectronics.X-CUBE-BLE2	0.0.1	
STMicroelectronics.X-CUBE-GNSS1	0.0.7	
STMicroelectronics.X-CUBE-MEMS1	6.1.0	
Device_Application		
Board_Component_MEMS		
Board_Extension_MEMS		
IKS01A2		☑
IKS01A3		☐
Board_Support_STM32Cube_Custom_BSP_Drivers		

8.2 Multipack usage policy

- Always one application can be selected.
- The selected application can either be one of the standalone Pack applications or the application Pack.
- The application is always referenced by the two high-level API entry:
  – *void MX_Entry_Init (void)*
  – *void MX_Entry_Process (void)*

8.3 Duplicated components management

Since the multipacks are based on the different components available inside each Pack, one or more middleware components can be present in several standalone Packs. This may lead to some integration issues.

Since the duplicated components (that have the same middleware) cannot be referenced together in the applicative Packs (multipacks), STM32CubeMX must offer the possibility to select one of the component (through its path).

Note that for middleware components, STM32CubeMX generates the middleware interfaces during project generation, starting from the templates.

Revision history

Table 16. Document revision history

Date	Revision	Changes
27-Jul-2020	1	Initial release.

Contents

List of tables

List of figures

Disclaimer

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.

© 2020 STMicroelectronics – All rights reserved