

Development guidelines for STM32Cube Expansion Packages

Introduction

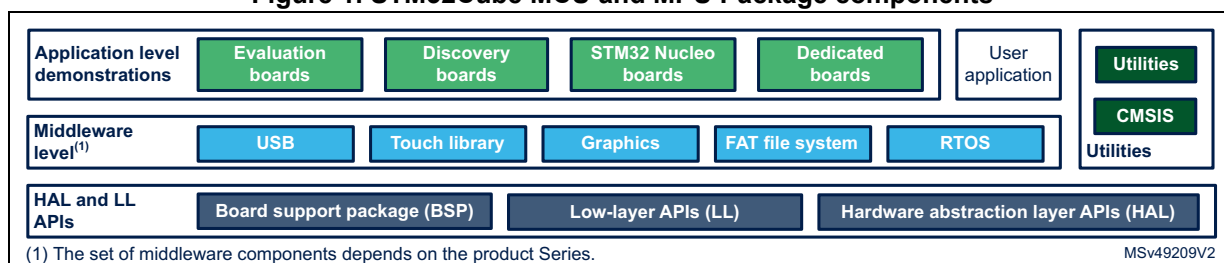
STM32Cube is an STMicroelectronics original initiative to significantly improve designer's productivity by reducing development effort, time and cost. STM32Cube covers the whole STM32 portfolio.

STM32Cube includes:

- A set of user-friendly software development tools to cover project development from the conception to the realization, among which STM32CubeMX, a graphical software configuration tool, STM32CubeIDE, an all-in-one development tool, and STM32CubeProgrammer (STM32CubeProg), a programming tool.
- STM32Cube MCU and MPU Packages, comprehensive embedded-software platforms specific to each microcontroller and microprocessor series (such as STM32CubeL4 for the STM32L4 Series), which include STM32Cube hardware abstraction layer (HAL), STM32Cube low-layer APIs, a consistent set of middleware components, and all embedded software utilities.
- STM32Cube Expansion Packages, which contain embedded software components that complement the functionalities of the STM32Cube MCU and MPU Packages with middleware extensions and applicative layers, and examples.

For a complete description of STM32Cube, refer to [Chapter 3](#).

Figure 1. STM32Cube MCU and MPU Package components



This document describes the compatibility requirements for STM32Cube Expansion Package development that ensure a proper match with STM32Cube MCU and MPU Package and tools, and overall consistency within the STM32Cube ecosystem, enabling rapid application development based on proven and validated software elements.

Readers of this document must be familiar with STM32Cube architecture, HAL and LL APIs, and programming model. A complete documentation set is available in the STM32Cube MCU and MPU Packages page on www.st.com.

Contents

1	General information	5
2	References and acronyms	5
3	What is STM32Cube?	6
4	STM32Cube Package and STM32Cube Expansion Package	7
4.1	Supported hardware	7
4.2	STM32Cube Package	8
4.3	STM32Cube Expansion Package	9
5	STM32Cube Expansion packaging requirements	10
5.1	Example development with STM32CubeMX	10
5.2	IDE and environment	10
5.3	Extension of STM32Cube Expansion Package drivers and middleware	13
6	New middleware integration in STM32Cube Expansion Package	15
6.1	Requirements	15
6.2	Middleware architecture	15
6.2.1	Middleware overview	15
6.2.2	Middleware architecture	16
6.2.3	Middleware repository	18
6.2.4	Middleware rules	21
6.2.5	Interfaces	21
6.2.6	High interface	22
6.2.7	Low interface	26
6.2.8	Application architecture	29
6.3	Delivery in object format	34
7	Software quality requirements	35
8	Revision history	36

List of tables

Table 1. List of acronyms 5

Table 2. Supported hardware 7

Table 3. Document revision history 36

List of figures

Figure 1.	STM32Cube MCU and MPU Package components	1
Figure 2.	STM32Cube Package content	9
Figure 3.	IDE folders in STM32Cube Package	11
Figure 4.	Building projects using CMSIS files	11
Figure 5.	Building projects using CMSIS files	12
Figure 6.	Building projects using CMSIS files	12
Figure 7.	Default IDE Flash loader selection	13
Figure 8.	STM32Cube drivers and middleware extension example.	14
Figure 9.	Examples of middleware in STM32Cube Packages	16
Figure 10.	Two middleware library categories	16
Figure 11.	Middleware component internal organization	17
Figure 12.	FatFS middleware folder organization	18
Figure 13.	FreeRTOS middleware folder organization	19
Figure 14.	STMicroelectronics middleware: typical folder structure.	20
Figure 15.	Middleware component interactions	22
Figure 16.	Calling middleware high interface: bare metal / main function	23
Figure 17.	Calling middleware high interface: bare metal / callback	24
Figure 18.	Calling middleware high interface: bare metal / static configuration.	24
Figure 19.	Calling middleware high interface: OS-based / main function	25
Figure 20.	Calling middleware high interface: OS-based / internal tasks created	26
Figure 21.	LwIP init low level function example	27
Figure 22.	low_level_init () function template pseudo code	27
Figure 23.	LwIP low-level init integration with STM32 example	28
Figure 24.	ETH_IRQHandler example	29
Figure 25.	Application entry point: main.c typical implementation	30
Figure 26.	Application entry point: main.h	31
Figure 27.	app_hw.c typical implementation	31
Figure 28.	mw_init() function implementation example	32
Figure 29.	LwIP middleware notification example	33
Figure 30.	LwIP middleware process implementation example	34

1 General information

In this user manual, both STM32Cube MCU Packages and STM32Cube MPU Packages are referred to as STM32Cube Packages.

The STM32Cube MCU and MPU Packages and STM32Cube Expansion Packages run on STM32 32-bit microcontrollers, based on the Arm^{®(a)} Cortex[®]-M processor.



2 References and acronyms

The following documents available on www.st.com are concurrently used for the development of STM32Cube Expansion Packages:

1. *Development checklist for STM32Cube Expansion Packages* (UM2312)
2. *Development guidelines for STM32Cube firmware Packs* (UM2388)
3. *How to create a software pack enhanced for STM32CubeMX using STM32 Pack Creator tool* (UM2739)
4. *STM32Cube BSP drivers development guidelines* (UM2298)

[Table 1](#) presents the definition of acronyms that are relevant for a better understanding of this document.

Table 1. List of acronyms

Term	Definition
API	Application programming interface
BSP	Board support package
CMSIS	Cortex [®] microcontroller system interface standard
DHCP	Dynamic host configuration protocol
FTP	File transfer protocol
HAL	Hardware abstraction layer
HTTP	Hypertext transfer protocol
LL	Low-layer
TCP/IP	Transmission control protocol / Internet protocol
TLS/SSL	Transport layer security / secure sockets layer

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

3 What is STM32Cube?

STM32Cube is an STMicroelectronics original initiative to significantly improve designer's 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 conception to the realization, among which are:
 - STM32CubeMX, a graphical software configuration tool that allows the automatic generation of C initialization code using graphical wizards
 - STM32CubeIDE, an all-in-one development tool with IP configuration, code generation, code compilation, and debug features
 - STM32CubeProgrammer (STM32CubeProg), a programming tool available in graphical and command line versions
 - STM32CubeMonitor (STM32CubeMonitor, STM32CubeMonPwr, STM32CubeMonRF, STM32CubeMonUCPD) powerful monitoring tools to fine-tune the behavior and performance of STM32 applications in real-time
- STM32Cube MCU and MPU Packages, comprehensive embedded-software platforms specific to each microcontroller and microprocessor series (such as STM32CubeL4 for the STM32L4 Series), which include:
 - STM32Cube hardware abstraction layer (HAL), ensuring maximized portability across the STM32 portfolio
 - STM32Cube low-layer APIs, ensuring the best performance and footprints with a high degree of user control over the HW
 - A consistent set of middleware components such as FAT file system, RTOS, USB Host and Device, TCP/IP, Touch library, and Graphics
 - All embedded software utilities with full sets of peripheral and applicative examples
- STM32Cube Expansion Packages, which contain embedded software components that complement the functionalities of the STM32Cube MCU and MPU Packages with:
 - Middleware extensions and applicative layers
 - Examples running on some specific STMicroelectronics development boards

4 STM32Cube Package and STM32Cube Expansion Package

The STM32Cube solution considered in this user manual is mostly composed of the STM32CubeMX, which is the tools part, and STM32Cube Package providing the software bricks needed to benefit from STM32 microcontroller features.



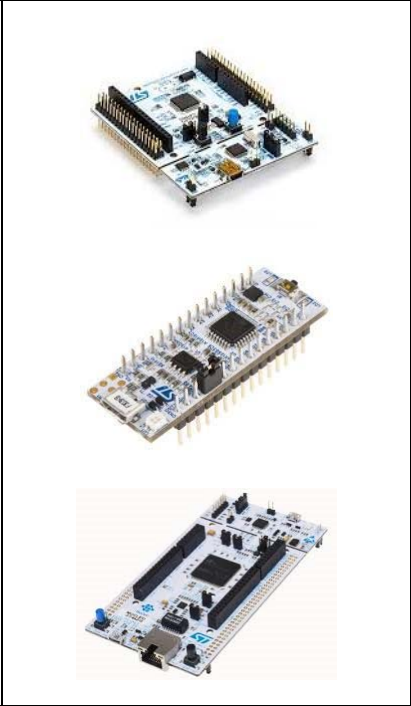
Additionally to the STM32CubeMX and STM32Cube Package, the STM32Cube Expansion Packages enrich the overall STM32Cube ecosystem with complementary add-ons.

4.1 Supported hardware

For a given STM32 microcontroller series, the STM32Cube firmware runs on any STM32 board or user compatible hardware built around an STM32 device within this series. This means that the user can simply update the BSP drivers to port the STM32Cube firmware on his own board, if this latter has the same hardware functionalities (LED, LCD display, buttons, and others).

The projects enclosed in the STM32Cube Packages run on all the STM32 boards including those coming with several versions and patches. Typically, these STM32 boards are divided into three types as illustrated in [Table 2](#).

Table 2. Supported hardware

Evaluation boards	Discovery kits	Nucleo boards
		

4.2 STM32Cube Package

The STM32Cube Package (such as STM32CubeF4 for the microcontrollers in the STM32F4 Series) provides, in one single package per STM32 microcontroller series, all the generic embedded software components required to develop an application on an STM32 microcontrollers in this series. Following STM32Cube initiative, this set of components is highly portable across all the STM32 series.

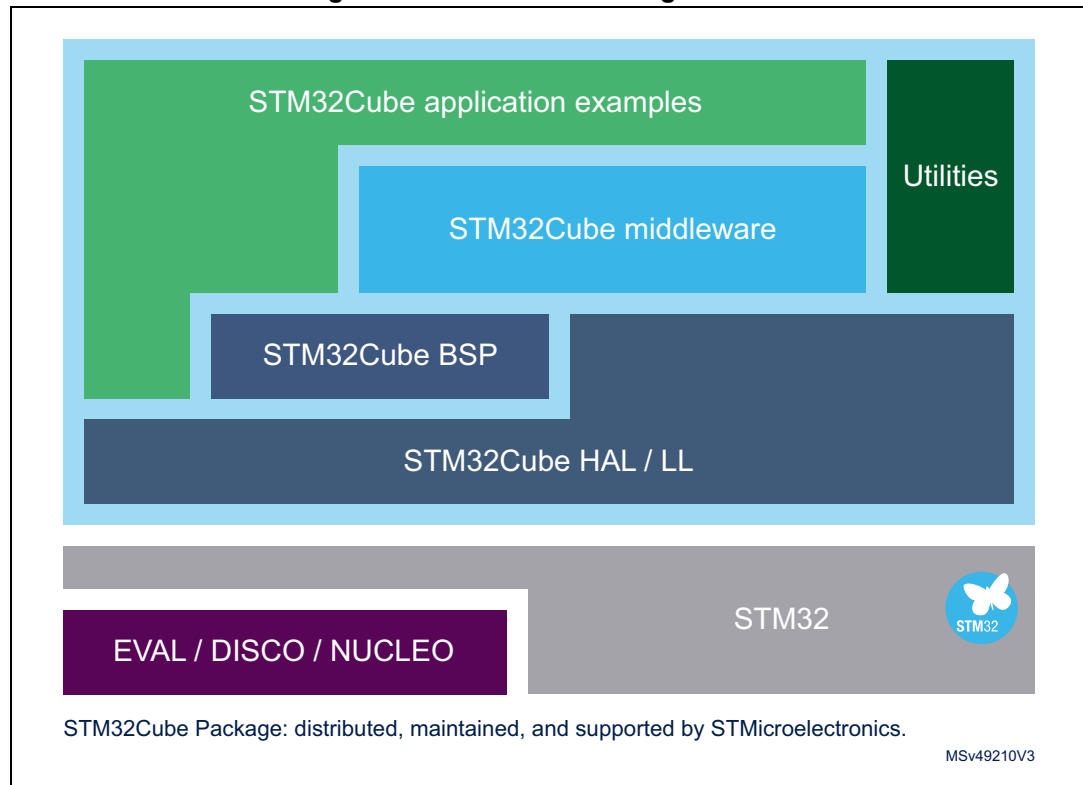
STM32Cube firmware is to a large extent compatible with the STM32CubeMX code generator that allows users to generate their initialization code. The STM32Cube Package is composed of:

- STM32 peripheral drivers covering the microcontroller hardware, together with an extensive set of examples running on all the STMicroelectronics boards
 - HAL (hardware abstraction layer)
 - LL (low-level API)
- Middleware components covering STM32 peripheral set with their 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 decode/Encoder library
- External components drivers (BSP) for STM32 boards
 - Evaluation boards
 - Discovery kits
 - Nucleo boards
- Global demonstrations per board exercising the different components (drivers and middleware)

The STM32Cube embedded software is distributed under the *Mix Liberty + OSS + 3rd party V1* mixed-licensed model as described on www.st.com.

Figure 2 presents the top-level structure of the STM32Cube Package as distributed, maintained and supported by STMicroelectronics. The structure of STM32Cube Packages is described more in detail in [2].

Figure 2. STM32Cube Package content



4.3 STM32Cube Expansion Package

STM32Cube Expansion Package contains add-on software components that complement the functionalities of the STM32Cube Package for:

- New middleware stack
- New hardware and board support (BSP)
- New examples
- Several of the items above

There are two categories of Expansion Packages: some STM32Cube Expansion Packages are developed, maintained and supported by STMicroelectronics, while some others are developed, maintained and distributed by third parties not affiliated to STMicroelectronics.

5 STM32Cube Expansion packaging requirements

The STM32Cube Package is the backbone of any STM32Cube Expansion Package. As a result, the folders and file structures must always be organized without modifying the original folder structures as described in "How to develop an STM32Cube Expansion Package"

(https://wiki.st.com/stm32mcu/wiki/How_to_develop_a_STM32Cube_Expansion_Package) in the STM32 MCU Wiki and within the *Development guidelines for STM32Cube firmware Packs* user manual [2].

The *STM32Cube Expansion Package development checklist* document [1] provides the list of requirements that must be respected by the STM32Cube Expansion Package.

5.1 Example development with STM32CubeMX

In an STM32Cube Expansion Package, examples must be developed using the STM32CubeMX tool. This requirement implies that the development satisfies to the following rules:

- The STM32CubeMX tool must be used to configure the STM32 device and board, and generate the corresponding initialization code
- In the generated *.h and *.c source files, the user must add the applicative code within the sections limited by the `/* USER CODE BEGIN */` and `/* USER CODE END */` markers
- The associated *.ioc file must be available at the example root
- If additional STM32CubeMX project settings are used, the *.extSettings* file must be kept at the same level as the *.ioc file

More details about the structure of the STM32Cube Expansion Package are provided within the *Development guidelines for STM32Cube firmware Packs* user manual [2].

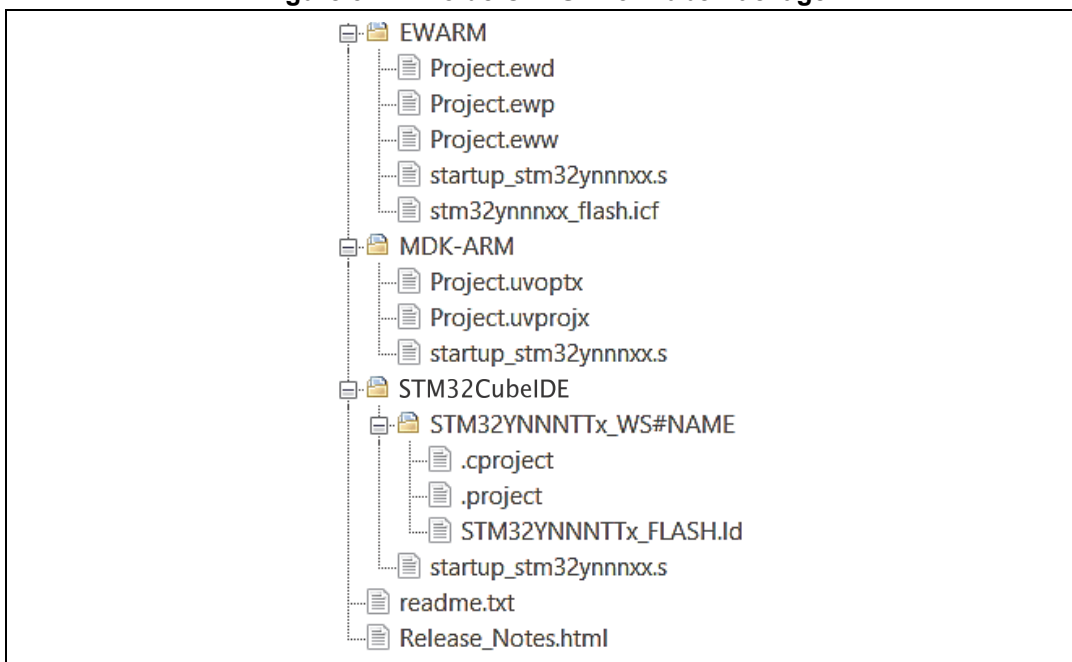
5.2 IDE and environment

The STM32Cube firmware projects are compliant with the requirements hereafter regarding the supported IDEs (Integrated Development Environments):

- All the projects are provided with IAR Systems EWARM
- All the projects are provided with Keil® MDK-ARM IDE
- All the projects are provided with STMicroelectronics STM32CubeIDE

All the IDE folders of the projects included in the STM32Cube Package have the scheme presented in [Figure 3](#).

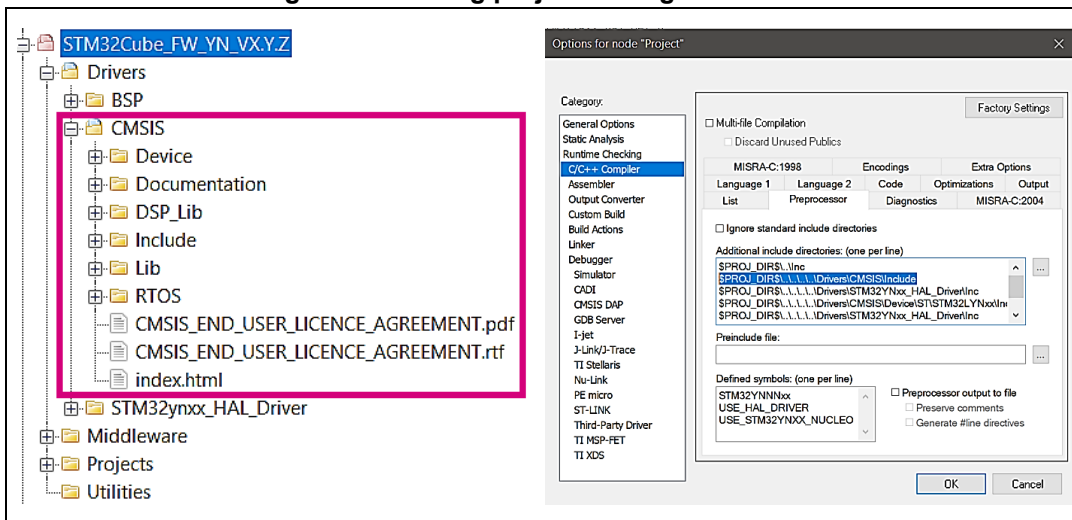
Figure 3. IDE folders in STM32Cube Package



Rules

- The projects must be built using the CMSIS files provided by the STM32Cube Package in folder *Drivers* as illustrated in [Figure 4](#).

Figure 4. Building projects using CMSIS files



- The following items must be added as symbols in the projects settings as illustrated in [Figure 5](#):
 - Device define (STM32YNNNxx)
 - USE_HAL_DRIVER (when the HAL is used)
 - USE_BOARD_NAME_PINOUT (Select the board/pinout to be used)

Figure 5. Building projects using CMSIS files

Additional include directories: (one per line)

```

$PROJ_DIR$\..\Inc
$PROJ_DIR$\..\..\Drivers\CMSIS\Device\ST\STM32YNNx\Inc
$PROJ_DIR$\..\..\Drivers\STM32YNNx_HAL_Driver\Inc
  
```

Preinclude file:

Defined symbols: (one per line)

```

STM32YNNNxx
USE_HAL_DRIVER
USE_STM32YNNX_NUCLEO_
  
```

☐ Preprocessor output to file

☐ Preserve comments

☐ Generate #line directives

- The optimization must be set high size (refer to [Figure 6](#)), except in LL projects where the footprint can drop in some cases when High Speed is selected.

Figure 6. Building projects using CMSIS files

Language 1 Language 2 Code Optimizations Output

Level

☐ None

☐ Low

☐ Medium

☒ High

Size

☐ No size constraints

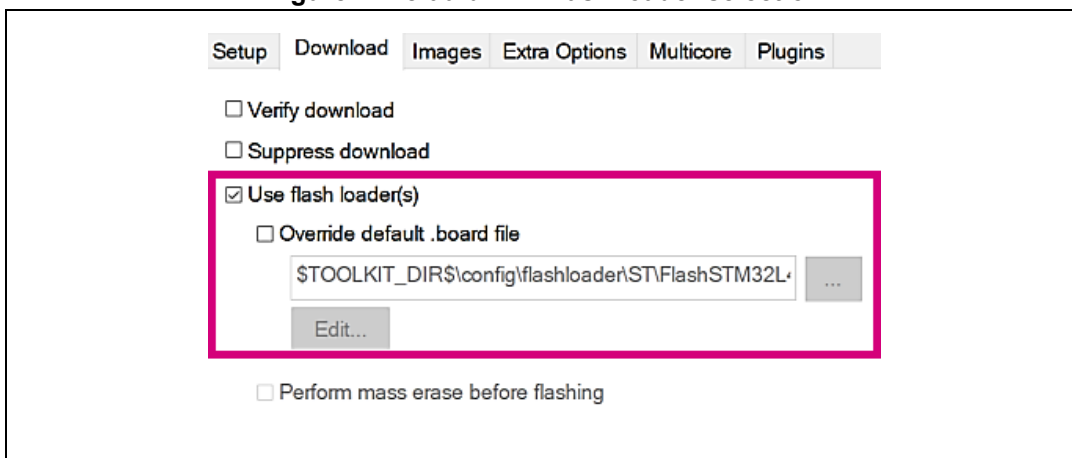
Enabled transformations:

- ☒ Common subexpression elimination
- ☒ Loop unrolling
- ☒ Function inlining
- ☒ Code motion
- ☒ Type-based alias analysis
- ☒ Static clustering
- ☒ Instruction scheduling
- ☐ Vectorization

- Multi configurations (workspaces) are allowed and must be used when the same project is provided with several different configurations.

- The IDE native Flash loader for a product must be selected by default as shown in [Figure 7](#).

Figure 7. Default IDE Flash loader selection



5.3 Extension of STM32Cube Expansion Package drivers and middleware

In some particular cases, it may happen that native drivers (HAL/LL, CMSIS, and BSP) or middleware provided within the STM32Cube Package, or both, need to be updated by the developer of the STM32Cube Expansion Package (for instance for extending the implemented features, early fixing of a bug prior to the official ST release, or others).

In such a situation, the user must proceed as follows:

1. Create extension components (files) for each native component that must be updated or overridden in the same folder.
2. Add the extension files into the project.

An example is provided in [Figure 8](#), where files outlined in pink are customized from files outlined in dark blue to implement specific features needed for an USB Device audio streaming application that is not natively supported by the STM32Cube Package.

Figure 8. STM32Cube drivers and middleware extension example.



6 New middleware integration in STM32Cube Expansion Package

6.1 Requirements

Middleware components are firmware layers that lie between the STM32 hardware and the user application. Basically, any new middleware component must comply to the following requirements:

- Must implement a modular architecture, breaking down complexity into simpler chunks or files.
- Must be hardware (device and board) independent.
- The connection with hardware drivers and other middleware stacks must be done through an interface file.
- The interface file must be provided as template within the middleware folder to be customized by the user.

6.2 Middleware architecture

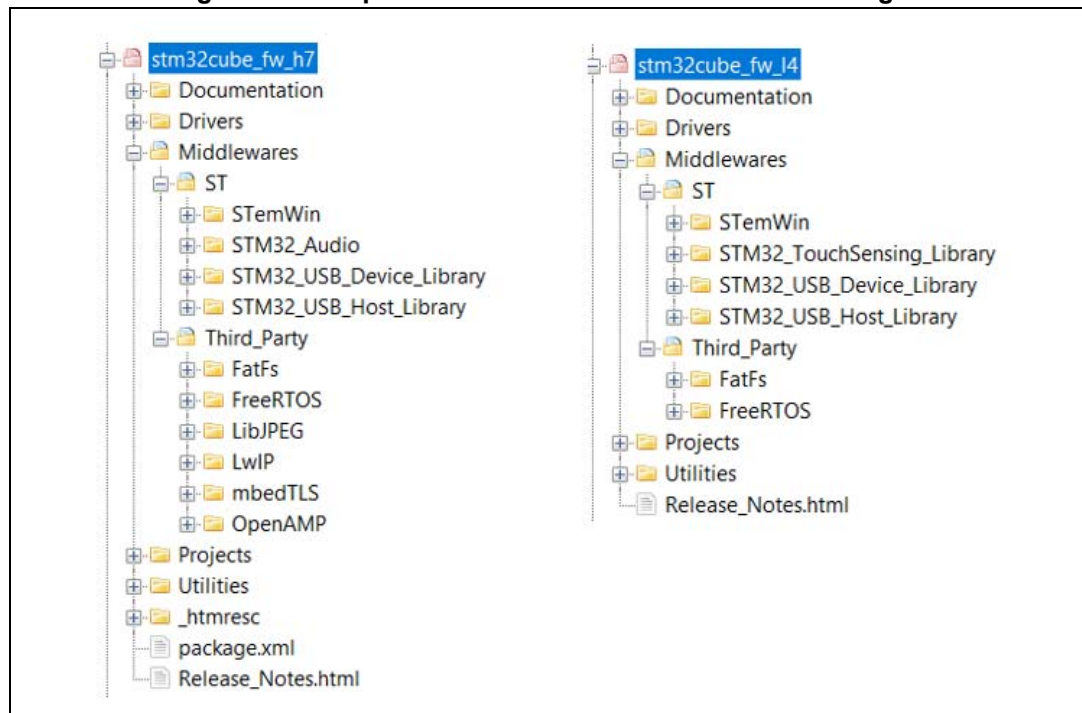
This section provides an overview of the middleware components architecture. The full architecture details are provided in the dedicated middleware components user manuals.

6.2.1 Middleware overview

The libraries and stacks are collections of non-volatile resources used by applications and examples. These may include configuration data, documentation, modules, pre-written code and subroutines, classes, values or type specifications, and eventually binary libraries.

The libraries contain code and data that provide services to independent applications and middleware components. They are either in open source code or binary format as pre-compiled libraries.

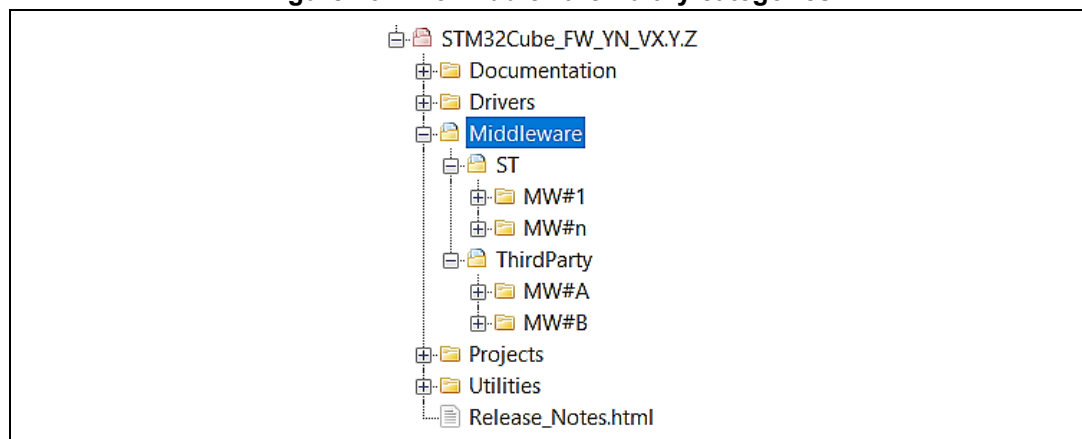
Middleware in the STM32Cube Package depends on the STM32 Series, its packaging policy, and the features to be promoted. Examples of middleware in two different STM32Cube Packages are shown in [Figure 9](#).

Figure 9. Examples of middleware in STM32Cube Packages

6.2.2 Middleware architecture

The libraries are divided into two categories (refer to [Figure 10](#)):

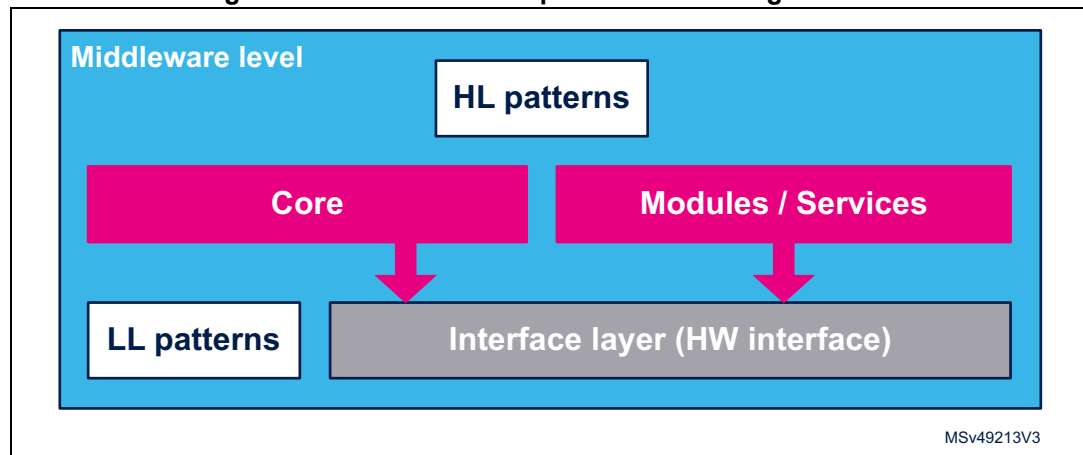
- STMicroelectronics libraries such as USB Host and Device library
- Third-party libraries such as JPEG decoders, FatFS file system, TCP/IP stacks and others

Figure 10. Two middleware library categories

Each library and middleware component is mainly composed of:

- **Modules:** a module is a common-functionality sub-layer, which can be added or removed individually in the library firmware. For example the TCP/IP stack is composed of the TCP/IP core and DHCP, HTTP, and FTP components. Each component is considered as a module. It can be added or removed by a specific define/macro in the configuration file, which enables or disables it.
- **Library core:** this is the kernel of a component; it manages the main library state machine and the data flow between the various modules.
- **Interface layer:** the interface layer is generally used to link the middleware core component with lower layers like the HAL and BSP drivers.
- **High-level (HL) patterns:** generic applicative code (platform agnostic) that is built on the middleware high-level services and can be used by the projects for common functionalities.
- **Low-level (LL) patterns:** specific hardware interface on generic hardware configuration that can be used with several STM32 series sharing the same hardware features.

Figure 11. Middleware component internal organization



6.2.3 Middleware repository

The third-party middleware folder organization derives from the packaging scheme of the original middleware provider. Nevertheless, additional folders and files can be added to match STM32Cube middleware architecture and sub-layer requirements. A typical example is illustrated in the FatFS and FreeRTOS™ middleware components as shown in [Figure 12](#) and [Figure 13](#).

Figure 12. FatFS middleware folder organization

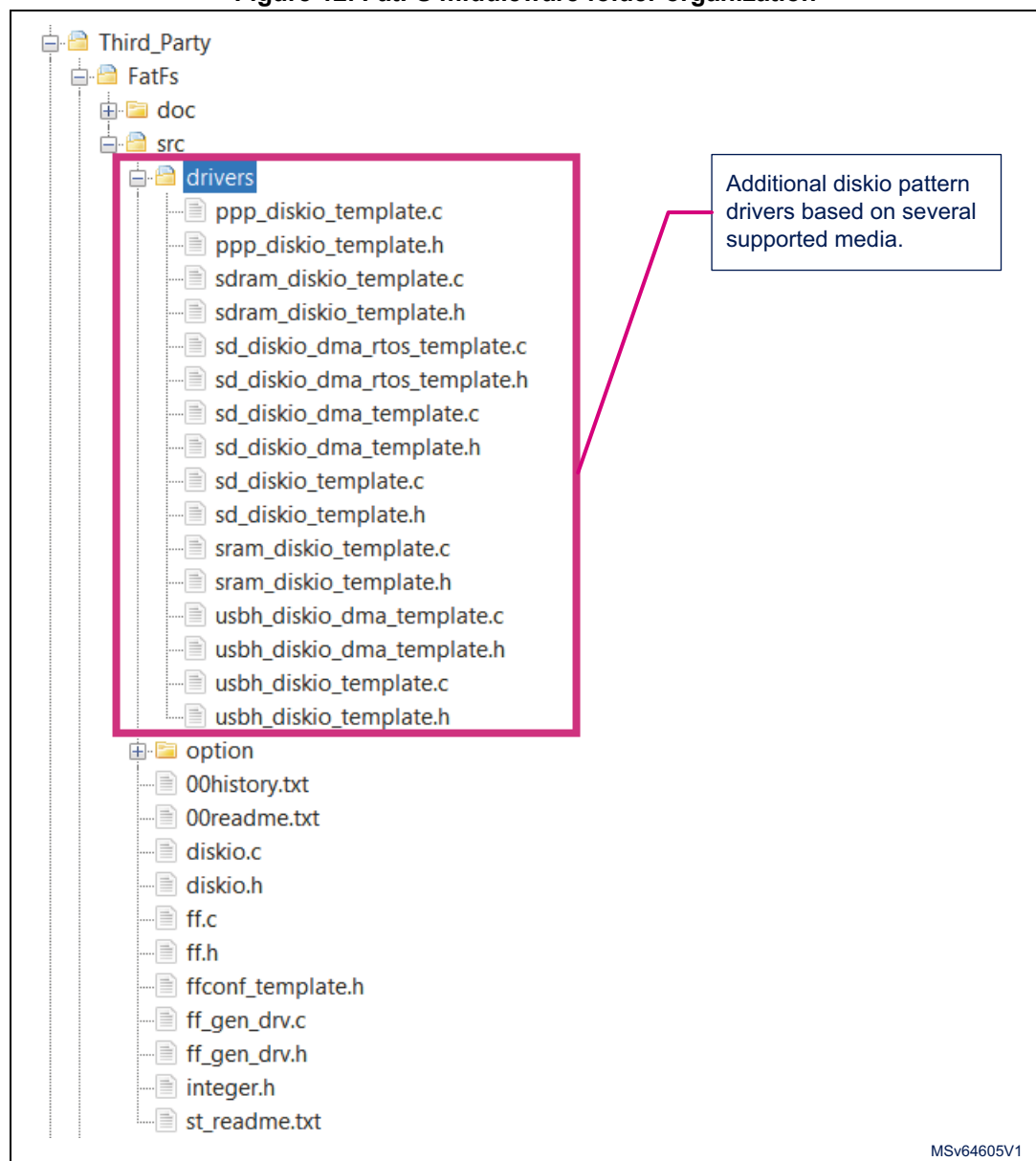
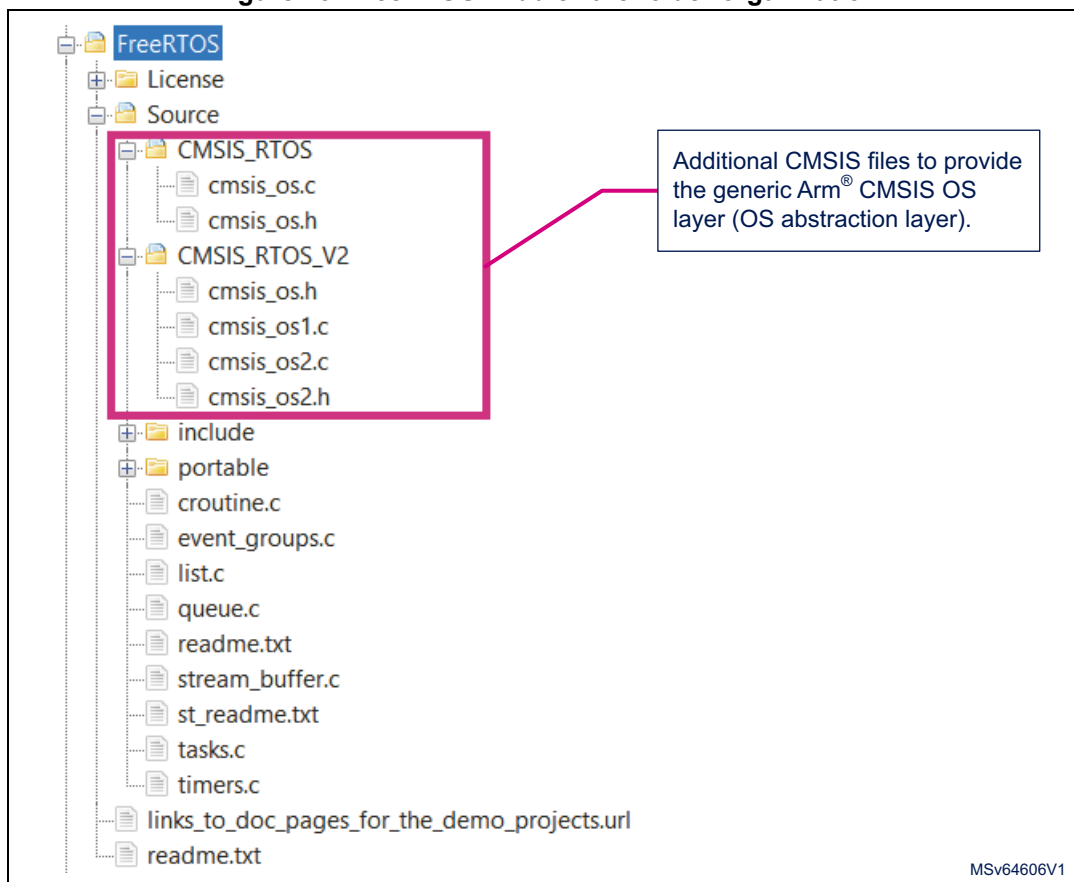


Figure 13. FreeRTOS middleware folder organization



For STMicroelectronics middleware components, the repository scheme is organized after the structure presented in [Section 6.2.2](#):

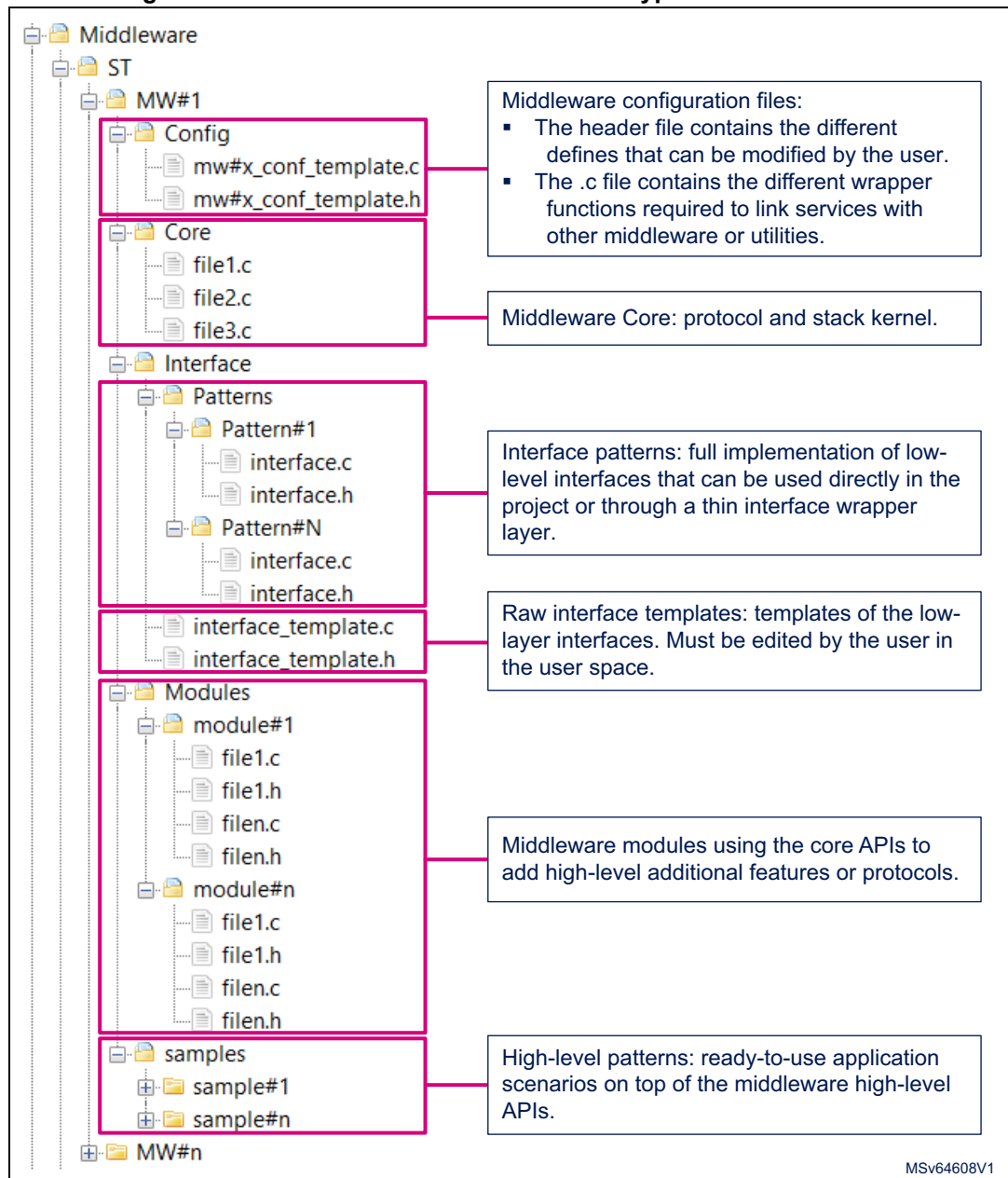
- Modules
- Library core
- Interface layer
- High-level patterns
- Low-level patterns

Note:

It is possible to omit some layers inside the middleware structure according to the middleware architecture and requirement. For instance, modules, high-level patterns and low-level patterns are not mandatory.

[Figure 14](#) shows a typical repository architecture for STMicroelectronics native middleware components that contains the different layers described above. In this figure, the different folder and file names are generic ones; they must be adapted to the middleware specificities.

Figure 14. STMicroelectronics middleware: typical folder structure



6.2.4 Middleware rules

Middleware libraries are considered as standalone components that must interact with other components through dedicated interfaces and configuration files. Related mandatory rules are defined:

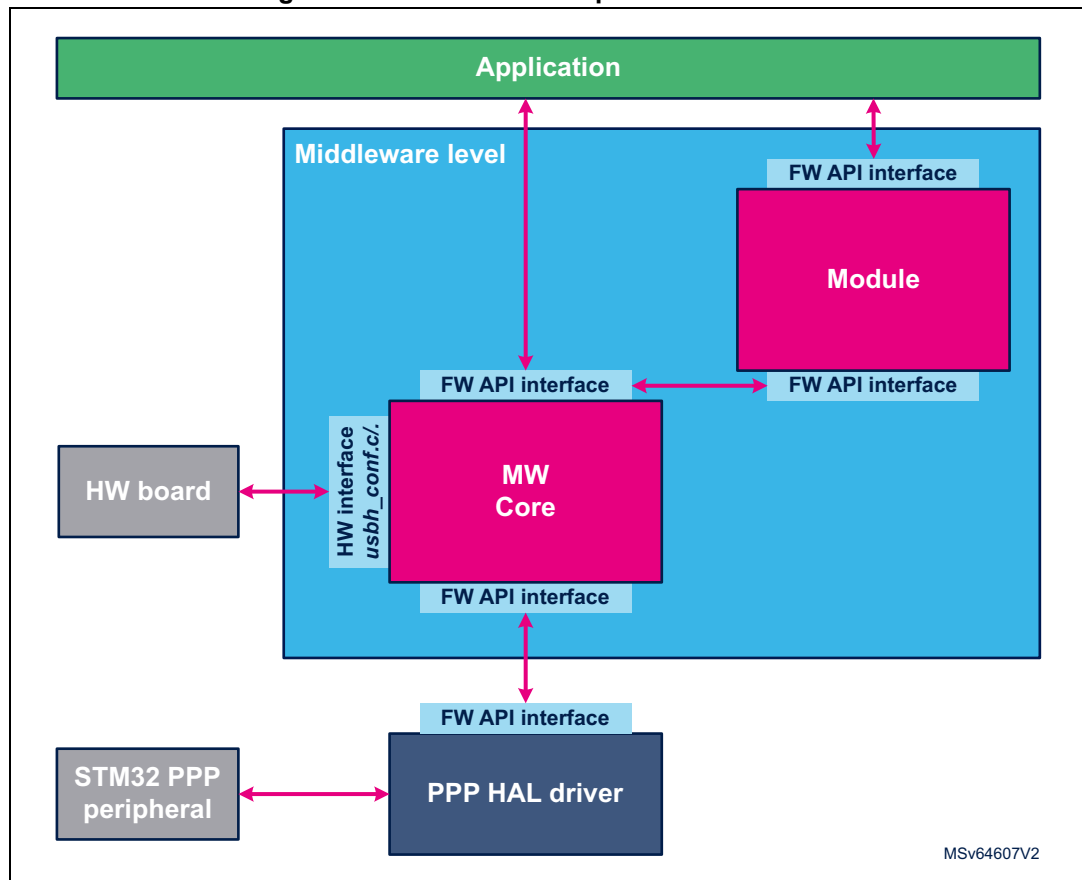
- The applications on top of middleware must comply to the advanced folder structure.
- The application core that calls the middleware APIs must be platform agnostic. Platform dependent applicative files must be located in the *Core* folder while the interfaces must be located in the target folder.
- Do not use IDE native library services directly (malloc, random, string operations or others). If needed, call them through dedicated macros or wrappers defined in the configuration files.
- The folder names for the patterns must refer neither to a specific STM32 series nor STM32 board, but rather to a common feature or module.

6.2.5 Interfaces

Similarly to the firmware components, the middleware components (core and modules) are interacting with other layers and firmware component through the three types of interfaces described below and illustrated in [Figure 15](#):

- **High interface:** application-programming interface based on a set of defined API init/configuration services and operation services (transfer, action and background tasks).
- **Low interface:** low-level services for a component that needs to be linked to one or more other components to get working. Generally, the low interface is composed of a set of `MW_Low_Level_Function` routines that must be filled from high-level services of other components or set of function pointers (fops) that need to be linked.
- **Hardware Interface:** allow to have access to hardware through the BSP, HAL, CMSIS, LL, or direct register access for user application needs.

Figure 15. Middleware component interactions



6.2.6 High interface

The middleware high interface is composed of the application programming services provided by the core and standalone supported modules. The services are classified into the following categories:

- Initialization functions
- Configuration functions
- One shot operations
- Background tasks
- RTOS based tasks
- Callbacks/Notification
- Static configuration (.h file)

The extracts of code in [Figure 16](#), [Figure 17](#) and [Figure 18](#) show how to call the middleware high interface, based on the LwIP in bare metal model:

Figure 16. Calling middleware high interface: bare metal / main function

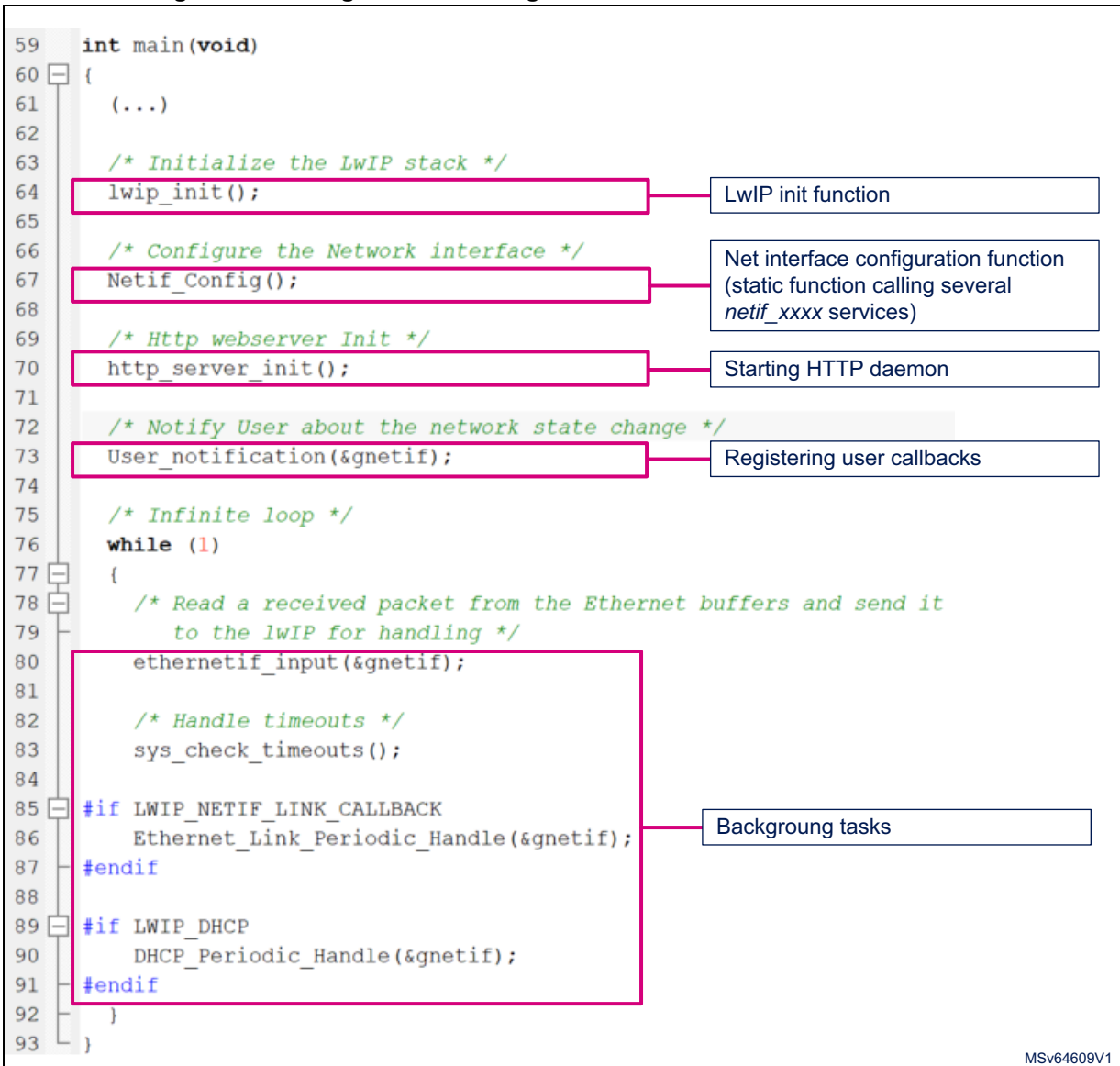


Figure 17. Calling middleware high interface: bare metal / callback

```

112 void ethernetif_notify_conn_changed(struct netif *netif)
113 {
114     if(netif_is_link_up(netif))
115     {
116         IP_ADDR4(&ipaddr, IP_ADDR0, IP_ADDR1, IP_ADDR2, IP_ADDR3);
117         IP_ADDR4(&netmask, NETMASK_ADDR0, NETMASK_ADDR1, NETMASK_ADDR2, NETMASK_ADDR3);
118         IP_ADDR4(&gw, GW_ADDR0, GW_ADDR1, GW_ADDR2, GW_ADDR3);
119         netif_set_addr(netif, &ipaddr, &netmask, &gw);
120
121         /* When the netif is fully configured this function must be called.*/
122         netif_set_up(netif);
123     }
124     else
125     {
126         /* When the netif link is down this function must be called.*/
127         netif_set_down(netif);
128     }
129 }

```

Figure 18. Calling middleware high interface: bare metal / static configuration

```

104 #define LWIP_IPV4          1
105 #define LWIP_TCP           1
106 #define LWIP_ICMP          1
107 #define LWIP_DHCP          1
108 #define LWIP_UDP           1

```


The next extracts of code in [Figure 19](#) and [Figure 20](#) show how to call the middleware high interface, based on the LwIP in OS-based model:

Figure 19. Calling middleware high interface: OS-based / main function

```

90 int main(void)
91 {
92     (...)
93     osThreadDef(Start, StartThread, osPriorityNormal, 0, configMINIMAL_STACK_SIZE * 2);
94
95     osThreadCreate (osThread(Start), NULL);
96
97     /* Start scheduler */
98     osKernelStart();
99
100    /* We should never get here as control is now taken by the scheduler */
101    for( ;; );
102 }
103
104 /**
105  * @brief Start Thread
106  * @param argument not used
107  * @retval None
108  */
109 static void StartThread(void const * argument)
110 {
111     /* Create tcp_ip stack thread */
112     tcpip_init(NULL, NULL);
113
114     /* Initialize the LwIP stack */
115     Netif_Config();
116
117     /* Initialize webserver demo */
118     http_server_socket_init();
119
120     /* Notify user about the network interface config */
121     User_notification(&gnetif);
122
123     for( ;; )
124     {
125         /* Delete the Init Thread */
126         osThreadTerminate(NULL);
127     }
128 }

```

Diagram annotations:

- A box labeled "Create start task" points to line 93: `osThreadDef(Start, StartThread, osPriorityNormal, 0, configMINIMAL_STACK_SIZE * 2);`
- A box labeled "Initialize TCP/IP (LwIP and internal tasks)" points to line 112: `tcpip_init(NULL, NULL);`

MSv64612V1

Figure 20. Calling middleware high interface: OS-based / internal tasks created

```

130 void tcpip_init(tcpip_init_done_fn initfunc, void *arg)
131 {
132     lwip_init();
133
134     tcpip_init_done = initfunc;
135     tcpip_init_done_arg = arg;
136     if (sys_mbox_new(&mbox, TCPIP_MBOX_SIZE) != ERR_OK) {
137         LWIP_ASSERT("failed to create tcpip_thread mbox", 0);
138     }
139     #if LWIP_TCPIP_CORE_LOCKING
140     if (sys_mutex_new(&lock_tcpip_core) != ERR_OK) {
141         LWIP_ASSERT("failed to create lock_tcpip_core", 0);
142     }
143     #endif /* LWIP_TCPIP_CORE_LOCKING */
144
145     sys_thread_new(TCPIP_THREAD_NAME, tcpip_thread, NULL, TCPIP_THREAD_STACKSIZE, TCPIP_THREAD_PRIO);
146 }

```

Internal tasks creation

MSv64613V1

6.2.7 Low interface

The middleware low interface is located in the middleware folder as template. It must be filled when linking middleware to the low level services (HAL, LL and others). The low interface provides a set of `MW_Low_Level_Function ()` routines that must be filled from high level services of other components or set of function pointers (fops) that must be linked.

As low interface resources (HAL drivers handles, local data structures) are defined in the interface file, all the user or middleware access methods to these resources must be defined in the same interface.

The low-level interface template provides the `MW_Low_Level_Function ()` routines with a generic body code that must be adapted to the low-level components to be used. The following example based on LwIP describe the init low level function in the template and after adapting them for the STM32 using the HAL Ethernet drivers.

Figure 21. LwIP init low level function example

```

537 err_t ethernetif_init(struct netif *netif)
538 {
539     LWIP_ASSERT("netif != NULL", (netif != NULL));
540
541     #if LWIP_NETIF_HOSTNAME
542         /* Initialize interface hostname */
543         netif->hostname = "lwip";
544     #endif /* LWIP_NETIF_HOSTNAME */
545
546     netif->name[0] = IFNAME0;
547     netif->name[1] = IFNAME1;
548     /* We directly use etharp_output() here to save a function call.
549      * You can instead declare your own function and call etharp_output()
550      * from it if you have to do some checks before sending (e.g. if link
551      * is available...) */
552     netif->output = etharp_output;
553     netif->linkoutput = low_level_output;
554
555     /* initialize the hardware */
556     low_level_init(netif);
557
558     return ERR_OK;
559 }

```

Link output method to the net interface

MSv64614V1

The `low_level_init ()` in the template is provided with a pseudo-code as shown in [Figure 22](#).

Figure 22. `low_level_init ()` function template pseudo code

```

84 static void low_level_init(struct netif *netif)
85 {
86     struct ethernetif *ethernetif = netif->state;
87
88     /* set MAC hardware address length */
89     netif->hwaddr_len = ETHARP_HWADDR_LEN;
90
91     /* set MAC hardware address */
92     netif->hwaddr[0] = ;
93     ...
94     netif->hwaddr[5] = ;
95
96     /* maximum transfer unit */
97     netif->mtu = 1500;
98
99     /* device capabilities */
100     /* don't set NETIF_FLAG_ETHARP if this device is not an ethernet one */
101     netif->flags = NETIF_FLAG_BROADCAST | NETIF_FLAG_ETHARP | NETIF_FLAG_LINK_UP;
102
103
104     /* Do whatever else is needed to initialize interface. */
105 }

```

The LwIP low-level init integration with the STM32 is done by linking the net interface to the ETH Ethernet HAL driver as shown in [Figure 23](#).

Figure 23. LwIP low-level init integration with STM32 example

```

242 static void low_level_init(struct netif *netif)
243 {
244     uint32_t regvalue = 0;
245     uint8_t macaddress[6] = { MAC_ADDR0, MAC_ADDR1, MAC_ADDR2, MAC_ADDR3, MAC_ADDR4, MAC_ADDR5 };
246
247     EthHandle.Instance = ETH;
248     EthHandle.Init.MACAddr = macaddress;
249     EthHandle.Init.AutoNegotiation = ETH_AUTONEGOTIATION_ENABLE;
250     EthHandle.Init.Speed = ETH_SPEED_100M;
251     EthHandle.Init.DuplexMode = ETH_MODE_FULLDUPLEX;
252     EthHandle.Init.MediaInterface = ETH_MEDIA_INTERFACE_MII;
253     EthHandle.Init.RxMode = ETH_RXPOLLING_MODE;
254     EthHandle.Init.ChecksumMode = ETH_CHECKSUM_BY_HARDWARE;
255     EthHandle.Init.PhyAddress = DP83848_PHY_ADDRESS;
256
257     /* configure ethernet peripheral (GPIOs, clocks, MAC, DMA) */
258     if (HAL_ETH_Init(&EthHandle) == HAL_OK)
259     {
260         netif->flags |= NETIF_FLAG_LINK_UP;
261     }
262
263     /* Initialize Tx Descriptors list: Chain Mode */
264     HAL_ETH_DMATxDescListInit(&EthHandle, DMATxDescTab, &Tx_Buff[0][0], ETH_TXBUFNB);
265
266     /* Initialize Rx Descriptors list: Chain Mode */
267     HAL_ETH_DMARxDescListInit(&EthHandle, DMARxDescTab, &Rx_Buff[0][0], ETH_RXBUFNB);
268
269     /* set MAC hardware address length */
270     netif->hwaddr_len = ETH_HWADDR_LEN;
271
272     /* set MAC hardware address */
273     netif->hwaddr[0] = MAC_ADDR0;
274     netif->hwaddr[1] = MAC_ADDR1;
275     netif->hwaddr[2] = MAC_ADDR2;
276     netif->hwaddr[3] = MAC_ADDR3;
277     netif->hwaddr[4] = MAC_ADDR4;
278     netif->hwaddr[5] = MAC_ADDR5;
279
280     /* maximum transfer unit */
281     netif->mtu = 1500;
282
283     /* device capabilities */
284     netif->flags |= NETIF_FLAG_BROADCAST | NETIF_FLAG_ETHARP;
285
286     /* Enable MAC and DMA transmission and reception */
287     HAL_ETH_Start(&EthHandle);
288
289     /**** Configure PHY to generate an interrupt when Eth Link state changes ****/
290     HAL_ETH_ReadPHYRegister(&EthHandle, PHY_MICR, &regvalue);
291     regvalue |= (PHY_MICR_INT_EN | PHY_MICR_INT_OE | PHY_MISR_LINK_INT_EN);
292     HAL_ETH_WritePHYRegister(&EthHandle, PHY_MISR, regvalue);
293 }

```

Init the HAL ETH driver

Assign MAC address

Start the HAL ETH process

Configure the PHY

MSv64616V1

The full integration of the net interface is done with LwIP through the following low level functions:

```

err_t ethernetif_init(struct netif *netif);
void ethernetif_input(struct netif *netif);

```

Caution: The Ethernet HAL MSP function and handle (driver resources) must be defined in the interface file and must not be accessed directly outside this file except to export the handle to the *stm32ynxx_it.c* file.

Figure 24. ETH_IRQHandler example

```
149 void ETH_IRQHandler(void)
150 {
151     HAL_ETH_IRQHandler(&EthHandle);
152 }
```

6.2.8 Application architecture

The application built on top of a middleware component calls the high interface services. It must not refer to low services (low interface); Such accesses must be handled internally by the middleware itself. In addition to the middleware high interface APIs, the application might require additional user oriented services such as debug console, buttons and miscellaneous HMI services.

The main file is implementing the system clock configuration, the `assert_failed()` function (delimited by the `USE_FULL_ASSERT` define), and the main routine that calls the middleware HL APIs, background processes and the application hardware services. The *main.c* must include only the *main.h* file, where only the application and configuration files must be included. The *main.c* file is the application entry point. A typical implementation is shown in the code extract shown in [Figure 25](#):

Figure 25. Application entry point: *main.c* typical implementation

```

41  /* Includes -----*/
42  #include "main.h"
43
44  /**
45   * @brief Main program
46   */
47  int main(void)
48  {
49      HAL_Init();
50
51      /* Configure the system clock to 180 MHz */
52      SystemClock_Config();
53
54      /* Configure the application hardware resources */
55      hw_Init();
56
57      /* Initialize the middleware */
58      MWName_Init();
59
60      /* Infinite loop */
61      while (1)
62      {
63          /* Call middleware background task */
64          MWName_Process();
65      }
66  }
67
68  /**
69   * @brief System Clock Configuration
70   */
71  static void SystemClock_Config(void)
72  {
73      (...)
74      ret = HAL_RCC_OscConfig(&RCC_OscInitStruct);
75
76      (...)
77      ret = HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_5);
78      if(ret != HAL_OK)
79      {
80          while(1) { ; }
81      }
82  }
83
84  #ifdef USE_FULL_ASSERT
85  /**
86   * @brief Assert function.
87   */
88  void assert_failed(uint8_t* file, uint32_t line)
89  {
90      /* User can add his own implementation to report the file name and line number,
91       ex: printf("Wrong parameters value: file %s on line %d\r\n", file, line) */
92
93      /* Infinite loop */
94      while (1)
95      {
96      }
97  }
98  #endif

```

Initialization of common application: clock and HAL

Initialization of application common services

Middleware initialization

Middleware background task

MSv64618V1

The *main.h* is calling the header files shown in [Figure 26](#).

Figure 26. Application entry point: *main.h*

```

38  /* Define to prevent recursive inclusion -----*/
39  #ifndef __MAIN_H
40  #define __MAIN_H
41
42  /* Includes -----*/
43  #include "stm32ynxx_hal.h"
44  #include "app_hw.h"
45  #include "app_mw.h"
46
47  #endif /* __MAIN_H */

```

The application additional user services such as debug console, buttons and miscellaneous and HMI services are located in the in the *app_hw.c* files. The BSP drivers cover most of these services, but they can be built using the HAL and LL APIs in user space as part of the application. A typical implementation of the *app_hw.c* file is shown in the code extract in [Figure 27](#).

Figure 27. *app_hw.c* typical implementation

```

56  void hw_init(void)
57  {
58      COM_InitTypeDef COM_Init;
59
60      /* Configure LEDs */
61      BSP_LED_Init(LED_GREEN);
62      BSP_LED_Init(LED_ORANGE);
63      BSP_LED_Init(LED_RED);
64      BSP_LED_Init(LED_BLUE);
65
66      /* Configure KEY Button & Joystick */
67      BSP_PB_Init(BUTTON_KEY, BUTTON_MODE_EXTI);
68      BSP_JOY_Init(JOY1, JOY_MODE_EXTI, JOY_ALL);
69
70      /* Configure COM1 : Use VCP */
71      COM_Init.BaudRate = 9600;
72      COM_Init.WordLength = 8;
73      COM_Init.StopBits = COM_STOPBITS_1;
74      COM_Init.Parity = COM_PARITY_NONE;
75      COM_Init.HwFlowCtl = COM_HWCONTROL_NONE;
76      BSP_COM_Init(COM1, &COM_Init);
77
78      /* Configure the Potentiometer */
79      BSP_POT_Init (POT1);
80  }

```

The middleware application processes are located in the *app_mw.c* file (such as the *app_lwip.c* for instance) and three services must be provided:

- Middleware initialization: full sequence of the middleware initialization including resources allocation (memory management) and modules configurations. The code extract in [Figure 28](#) shows an example of the *mw_init ()* function.

Figure 28. *mw_init()* function implementation example

```

3 void lwip_init(void)
4 {
5     ip_addr_t ipaddr;
6     ip_addr_t netmask;
7     ip_addr_t gw;
8
9     /* Initialize the LwIP stack */
10    lwip_init();
11
12    /* Configure the Network interface */
13    #ifndef USE_DHCP
14        ip_addr_set_zero_ip4(&ipaddr);
15        ip_addr_set_zero_ip4(&netmask);
16        ip_addr_set_zero_ip4(&gw);
17    #else
18        IP_ADDR4(&ipaddr, IP_ADDR0, IP_ADDR1, IP_ADDR2, IP_ADDR3);
19        IP_ADDR4(&netmask, NETMASK_ADDR0, NETMASK_ADDR1, NETMASK_ADDR2, NETMASK_ADDR3);
20        IP_ADDR4(&gw, GW_ADDR0, GW_ADDR1, GW_ADDR2, GW_ADDR3);
21    #endif /* USE_DHCP */
22
23    /* add the network interface */
24    netif_add(&netif, &ipaddr, &netmask, &gw, NULL, &ethernetif_init, &ethernet_input);
25
26    /* Registers the default network interface */
27    netif_set_default(&netif);
28
29    if (netif_is_link_up(&netif))
30    {
31        /* When the netif is fully configured this function must be called */
32        netif_set_up(&netif);
33    }
34    else
35    {
36        /* When the netif link is down this function must be called */
37        netif_set_down(&netif);
38    }
39
40    /* Set the link callback function, this function is called on change of link status */
41    netif_set_link_callback(&netif, ethernetif_update_config);
42
43    /* Http webserver Init */
44    /* Httpd Init */
45    httpd_init();
46
47    /* configure SSI handlers (ADC page SSI) */
48    http_set_ssi_handler(ADC_Handler, (char const **)TAGS, 1);
49
50    /* configure CGI handlers (LEDs control CGI) */
51    CGI_TAB[0] = LEDS_CGI;
52    http_set_cgi_handlers(CGI_TAB, 1);
53 }

```


- Middleware notifications (hooks/callbacks): The middleware notification callbacks provide a means to inform the user about the internal middleware component state and send dedicated events notification to let the user interact with the application. An example of notification is shown below in [Figure 29](#) for the LwIP middleware notification.

Figure 29. LwIP middleware notification example

```

112 void ethernetif_notify_conn_changed(struct netif *netif)
113 {
114 #ifndef USE_DHCP
115     ip_addr_t ipaddr;
116     ip_addr_t netmask;
117     ip_addr_t gw;
118 #endif
119
120     if(netif_is_link_up(netif))
121     {
122 #ifndef USE_LCD
123         LCD_UsrLog ("The network cable is now connected \n");
124 #else
125         BSP_LED_Off(LED2);
126         BSP_LED_On(LED1);
127 #endif /* USE_LCD */
128
129 #ifndef USE_DHCP
130         /* Update DHCP state machine */
131         DHCP_state = DHCP_START;
132 #else
133         IP_ADDR4(&ipaddr, IP_ADDR0, IP_ADDR1, IP_ADDR2, IP_ADDR3);
134         IP_ADDR4(&netmask, NETMASK_ADDR0, NETMASK_ADDR1, NETMASK_ADDR2, NETMASK_ADDR3);
135         IP_ADDR4(&gw, GW_ADDR0, GW_ADDR1, GW_ADDR2, GW_ADDR3);
136         netif_set_addr(netif, &ipaddr, &netmask, &gw);
137 #endif
138 #ifndef USE_LCD
139         uint8_t iptxt[20];
140         sprintf((char *)iptxt, "%s", ip4addr_ntoa((const ip4_addr_t *)&netif->ip_addr));
141         LCD_UsrLog ("Static IP address: %s\n", iptxt);
142 #endif /* USE_LCD */
143 #endif /* USE_DHCP */
144
145         /* When the netif is fully configured this function must be called.*/
146         netif_set_up(netif);
147     }
148     else
149     {
150 #ifndef USE_DHCP
151         /* Update DHCP state machine */
152         DHCP_state = DHCP_LINK_DOWN;
153 #endif /* USE_DHCP */
154
155         /* When the netif link is down this function must be called.*/
156         netif_set_down(netif);
157
158 #ifndef USE_LCD
159         LCD_UsrLog ("The network cable is not connected \n");
160 #else
161         BSP_LED_Off(LED1);
162         BSP_LED_On(LED2);
163 #endif /* USE_LCD */
164     }
165 }

```

- Middleware background tasks and processes: depending on the middleware usage model (bare metal or RTOS based), the background processes handle the internal middleware protocols and state machines inside a loop or a task:

Figure 30. LwIP middleware process implementation example

```

87 void lwip_process(void)
88 {
89     /* Read a received packet from the Ethernet buffers and send them to the stack */
90     ethernetif_input(&gnetif);
91
92     /* Handle timeouts */
93     sys_check_timeouts();
94
95     #ifndef USE_DHCP
96     /* handle periodic timers for DHCP */
97     DHCP_Periodic_Handle(&gnetif);
98     #endif
99 }

```

6.3 Delivery in object format

When the middleware library is delivered in binary or object format, it must comply to a minimum set of requirements:

- A header file must be provided to export the library interface API to end applications
- A release note must be made available
- The library must be provided in object format for all supported compilers. In case the library object is compiler dependent, the supported compilers have to be clearly indicated in the object file name.

As an example, **LibraryNameV_CMx_C_O.a** is a library object file name where:

- **V**: module version (for instance V=01 for a V0.1 release)
- **x**: the CMx core class (CM0, CM3, CM4, CM7, CM23, CM33)
- **C**: compiler (IAR™, Keil®, GCC)
- **O**: specify the compiler optimization
- **<empty>**: high size optimization
- **Of**: high-speed optimization
- **Otnsc**: high-speed optimization with No Size constraints
- **Ob**: high-balanced optimization

7 Software quality requirements

BSP drivers, middleware and projects developed within the STM32Cube Expansion Package (add-ons with respect to STM32Cube Package) have to meet the minimum set of requirements below:

- Ensure compilation with all supported compilers (EWARM, MDK-ARM and STM32CubeIDE) on Windows® and Linux® platforms, without errors nor warnings (warnings are accepted only if present on SW components not owned by the developer of the Expansion Package).
- Functional tests are performed with no known bugs left (minor bugs are accepted provided they are documented in the component release note), with evidence reports.

BSP drivers and middleware must comply to these additional requirements:

- MISRA C® compliance and static code analysis, with evidence reports
- MISRA C® non-compliance rules deviation must be justified.

8 Revision history

Table 3. Document revision history

Date	Revision	Changes
14-Nov-2017	1	Initial release.
9-Sep-2020	2	Updated the description of STM32Cube on the cover page and in Chapter 3: What is STM32Cube? , Section 4.1: Supported hardware , and Section 4.2: STM32Cube Package . Replaced Figure 3 by Section 5.2: IDE and environment and referred to UM2388 for STM32Cube Package structure and content. Updated Section 5.3: Extension of STM32Cube Expansion Package drivers and middleware . Replaced Organization by Section 6.2: Middleware architecture .

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