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 ..................................................................... 30
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\(^{\text{\textregistered}}\)-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>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>API</td>
<td>Application programming interface</td>
</tr>
<tr>
<td>BSP</td>
<td>Board support package</td>
</tr>
<tr>
<td>CMSIS</td>
<td>Cortex(^{\text{\textregistered}}) microcontroller system interface standard</td>
</tr>
<tr>
<td>DHCP</td>
<td>Dynamic host configuration protocol</td>
</tr>
<tr>
<td>FTP</td>
<td>File transfer protocol</td>
</tr>
<tr>
<td>HAL</td>
<td>Hardware abstraction layer</td>
</tr>
<tr>
<td>HTTP</td>
<td>Hypertext transfer protocol</td>
</tr>
<tr>
<td>LL</td>
<td>Low-layer</td>
</tr>
<tr>
<td>TCP/IP</td>
<td>Transmission control protocol / Internet protocol</td>
</tr>
<tr>
<td>TLS/SSL</td>
<td>Transport layer security / secure sockets layer</td>
</tr>
</tbody>
</table>

---

(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>
<thead>
<tr>
<th>Table 2. Supported hardware</th>
</tr>
</thead>
<tbody>
<tr>
<td>Evaluation boards</td>
</tr>
<tr>
<td><img src="image1" alt="Evaluation board" /></td>
</tr>
</tbody>
</table>
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](http://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**

```
- EWAR
  - Project.ewd
  - Project.ewp
  - Project.eww
  - startup stm32yminnoc.s
  - stm32yminnoc_flash.icf

- MDK-ARM
  - Project.uvcppx
  - Project.uvcppr
  - startup stm32yminnoc.s

- STM32CubeIDE
  - STM32YNNNTT_x_WS#NAME
    - .cproject
    - .project
    - STM32YNNNTT_x_FLASH.Id
    - startup stm32yminnoc.s
    - readme.txt
    - Release_Notes.html
```

**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

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

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.
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

Additional diskio pattern drivers based on several supported media.
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.
Middleware configuration files:
- The header file contains the different defines that can be modified by the user.
- The .c file contains the different wrapper functions required to link services with other middleware or utilities.

Middleware Core: protocol and stack kernel.

Interface patterns: full implementation of low-level interfaces that can be used directly in the project or through a thin interface wrapper layer.

Raw interface templates: templates of the low-layer interfaces. Must be edited by the user in the user space.

Middleware modules using the core APIs to add high-level additional features or protocols.

High-level patterns: ready-to-use application scenarios on top of the middleware high-level APIs.
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.
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

```c
int main(void)
{
    (...)
    lwip_init();
    Netif_Config();
    http_server_init();
    User_notification(&netif);
    while (1)
    {
        /* Read a received packet from the Ethernet buffers and send it to the lwIP for handling */
        ethernetif_input(&netif);
        /* Handle timeouts */
        sys_check_timeouts();
    }
    #if LWIP_NETIF_LINK_CALLBACK
    Ethernet_Link_Periodic_Handle(&netif);
    #endif
    #if LWIP_DHCP
    DHCP_Periodic_Handle(&netif);
    #endif
}
```

- **LwIP init function**
- **Net interface configuration function (static function calling several netif xxxx services)**
- **Starting HTTP daemon**
- **Registering user callbacks**
- **Background tasks**
Figure 17. Calling middleware high interface: bare metal / callback

```c
void ethernet_notify_conn_changed(struct netif *netif)
{
    if (netif_is_link_up(netif))
    {
        IP_ADDR4(sipaddr, IP_ADDR0, IP_ADDR1, IP_ADDR2, IP_ADDR3);
        IP_ADDR4(gwaddr, NETMASK_ADDR0, NETMASK_ADDR1, NETMASK_ADDR2, NETMASK_ADDR3);
        netif_set_addr(netif, sipaddr , gwaddr, 4); /* When the netif is fully configured this function must be called.*/
        netif_set_up(netif);
    } else
    { /* When the netif link is down this function must be called.*/
        netif_set_down(netif);
    }
}
```

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

```
#define LWIP_IPV4 1
#define LWIP_TCP 1
#define LWIP_ICMP 1
#define LWIP_HTTP 1
#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**

```c
int main(void)
{
    // Create start task
    osThreadDef(Start, StartThread, osPriorityNormal, 0, configMINIMAL_STACK_SIZE * 2);
    osThreadCreate(osThread(Start), NULL);
    // Start scheduler */
    osKernelStart();
    // We should never get here as control is now taken by the scheduler */
    for(;; )
    {
        // Initialize TCP/IP (LwIP and internal tasks)
        initializeTCP_IP(stack); // Initialize the LwIP stack */
        Netif_Config();
        // Initialize vserver demo */
        http_server_socket_init();
        // Notify user about the network interface config */
        User_notification(stack);)
        for(;; )
        {
            // Delete the Init Thread */
            osThreadTerminate(NULL);
        }
    }
}
```
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 \texttt{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 \texttt{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.
The `low_level_init()` in the template is provided with a pseudo-code as shown in Figure 22.

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

```c
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(uEthHandle);
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 **Figure 25**: 
Figure 25. Application entry point: `main.c` typical implementation

```c
#include "main.h"

/* Brief Main program */
int main(void)
{
    HAL_Init();
    /* Configure the system clock to 180 MHz */
    SystemClock_Config();

    /* Configure the application hardware resources */
    hw_Init();

    /* Initialize the middleware */
    MName_Init();

    /* Infinite loop */
    while (1)
    {
        /* Call middleware background task */
        MName_Process();
    }
}

/* Brief System Clock Configuration */
static void SystemClock_Config(void)
{
    /* ... */
    ret = HAL_RCC_OscConfig(&RCC_OscInitStruct);
    /* ... */
    ret = HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_5);
    if (ret != HAL_OK)
    {
        while (1) ;
    }
}

#ifdef USE_FULL_ASSERT
#endif
/* Brief Assert function. */
void assert_failed(uint8_t * file, uint32_t line)
{
    /* User can add his own implementation to report the file name and line number, 
    ex: printf("Wrong parameters value: file %s on line %d\n", file, line) */

    /* Infinite loop */
    while (1)
    {
    }
}
#endif
```
The *main.h* is calling the header files shown in **Figure 26**.

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

```c
18 /* Define to prevent recursive inclusion --------------------------------*/
19 ifndef __MAIN_H
20 #define __MAIN_H
21 /*
22 Includes -----------------------------------------------*/
23 #include "stm32yxxx_hal.h"
24 #include "app_hw.h"
25 #include "app_mw.h"
26 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**

```c
56 void hw_init(void)
57 {
58     COM_InitTypeDef COM_Init;
59     /* Configure LEDs */
60     BSP_LED_Init(LED_GREEN);
61     BSP_LED_Init(LED_ORANGE);
62     BSP_LED_Init(LED_RED);
63     BSP_LED_Init(LED_BLUE);
64     /* Configure KEY Button & Joystick */
65     BSP_PB_Init(BUTTON_KEY, BUTTON_MODE_EXIT);
66     BSP_JOY_Init(JOY1, JOY_MODE_EXIT, JOY_ALL);
67     /* Configure COM1 : Use VCP */
68     COM_Init.BaudRate = 9600;
69     COM_Init.WordLength = 8;
70     COM_Init.StopBits = COM_STOPBITS_1;
71     COM_Init.Parity = COM_PARITY_NONE;
72     COM_Init.HandFlowCtl = COM_HWCONTROL_NONE;
73     BSP_COM_Init(COM1, &COM_Init);
74     /* Configure the Potentiometer */
75     BSP_FOI_Init(FOI1);
76 }
```

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

```c
void mw_init(void)
{
  ip_addr_t ipaddr;
  ip_addr_t netmask;
  ip_addr_t gw;

  /* Initialize the lwIP stack */
  lwip_init();

  /* Configure the Network interface */
  ifdef USE_DEC
  ip_addr_set_zero_ip4(&ipaddr);
  ip_addr_set_zero_ip4(&netmask);
  ip_addr_set_zero_ip4(&gw);
  else
  IP_ADDR4(ipaddr, IP_ADDR0, IP_ADDR1, IP_ADDR2, IP_ADDR3);
  IP_ADDR4(netmask, NETWORK_ADDR0, NETWORK_ADDR1, NETWORK_ADDR2, NETWORK_ADDR3);
  IP_ADDR4(gw, GW_ADDR0, GW_ADDR1, GW_ADDR2, GW_ADDR3);
  endif
  /* add the network interface */
  netif_add(ipnetif, ipaddr, netmask, gw, NULL, ethernet_init, ethernet_input);

  /* Registers the default network interface */
  netif_set_default(ipnetif);

  if (netif_is_link_up(ipnetif))
  { /* When the netif is fully configured this function must be called */
    netif_set_up(ipnetif);
  }
  else
  { /* When the netif link is down this function must be called */
    netif_set_down(ipnetif);
  }

  /* Set the link callback function, this function is called on change of link status*/
  netif_link_callback(ipnetif, ethernet_update_config);

  /* Http webservice Init */
  http_init();

  /* configure SSI handlers (ADC page SSI) */
  http_add_ssi_handler(ADC_handler, (char const **)TAGS, 1);

  /* configure CGI handlers (LEDs control CGI) */
  CGI_TAB[] = LEDS/cgi;
  http_add_cgi_handler(CGI_TAB, 1);
```

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**

```c
void ethenetif_notify_conn_changed(struct netif *netif)
{
    if (netif_is_link_up(netif))
    {
        #ifdef USE_LCD
            LCD_Printf("The network cable is now connected \n\n");
        #else
            BSP_LED_Off(LED0);
            BSP_LED_On(LED1);
        #endif
    }
}

/* Update DHCP state machine */
DHCP_state = DHCP_START;

/* Static IP address */
IP_ADDR4(ipaddr, TP_ADDR1, IP_ADDR2, IP_ADDR3);
IP_ADDR4(Netmask, NETMASK_ADDR1, NETMASK_ADDR2, NETMASK_ADDR3);
IP_ADDR4(gw, GW_ADDR1, GW_ADDR2, GW_ADDR3);
netif_set_addr(netif, ipaddr, Netmask, gw);

/* When the netif is fully configured this function must be called.*/
netif_up(netif);
}

/* Update DHCP state machine */
DHCP_state = DHCP_LINK_DOWN;

/* When the netif link is down this function must be called.*/
netif_down(netif);
```

```c
#ifdef USE_LCD
    uins32_t iptxt[20];
    sprintf((char *)iptxt, "%h", ipaddr_ntoa((const ip_addr_t *)netif->ip_addr));
    LCD_Printf("Static IP address: %s", iptxt);
    sendif /* USE_LCD */
#else
    #ifdef USE_DNSCP
    else
    #endif
    #endif
#endif
```
- 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**

```c
07 void lwip_process(void)
08 {
09    /* Read a received packet from the Ethernet buffers and send them to the stack */
10    ethernetif_input(&enetif);
11    /* Handle timeouts */
12    nva_check_timeouts();
13}
```

### 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
  - **Ot**: 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.
## Revision history

### Table 3. Document revision history

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