life.augmented

UM2298
User manual

STM32Cube BSP drivers development guidelines

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, STM32CubeProgrammer (STM32CubeProg), a programming tool, and STM32CubeMonitor-Power (STM32CubeMonPwr) monitoring 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 2*.

The BSP (board support package) drivers are part of the STM32Cube MCU and MPU Packages based on the HAL drivers, and provide a set of high-level APIs relative to the hardware components and features on the Evaluation boards, Discovery kits and Nucleo boards delivered with the STM32Cube MCU and MPU Packages for a given STM32 microcontroller Series.

The BSP drivers allow quick access to the board services using high-level APIs, without any specific configuration as the link with the HAL and the external components is made intrinsically within the drivers. From the project-setting point of view, the user has only to add the necessary driver files in the workspace and call the functions needed from the examples in Level0, Level1 or Level2 in *Figure 2*. However, some low-level configuration functions may be overridden by the applications if the user wants to change the default behavior of the BSP drivers.

The purpose of this document is to provide the user with guidance for the development of BSP drivers.

It gives an overview of the architecture of BSP drivers (class, component, common and bus) and implementation examples.

Contents

1 **Documentation conventions 8**
1.1 About this document 8
1.2 Acronyms and definitions 8

2 **What is STM32Cube? 10**

3 **BSP driver architecture 11**
3.1 BSP drivers features 13
3.2 Hardware configuration 13
3.3 BSP driver repository 15
3.4 BSP based project repository 17
3.5 BSP inclusion model 19

4 **BSP naming rules 20**
4.1 General rules 20
4.2 Board naming rules 21

5 **BSP common driver 22**
5.1 BSP common driver functions 23
5.2 BSP common functions implementation 25
5.2.1 BSP_GetVersion 25
5.2.2 BSP_LED_Init 26
5.2.3 BSP_LED_DeInit 28
5.2.4 BSP_LED_On 28
5.2.5 BSP_LED_Off 28
5.2.6 BSP_LED_Toggle 29
5.2.7 BSP_LED_GetState 29
5.2.8 BSP_PB_Init 31
5.2.9 BSP_PB_GetState 33
5.2.10 BSP_COM_Init 34
5.2.11 BSP_COM_SelectLogPort 36
5.2.12 BSP_JOY_Init 37
5.2.13 BSP_JOY_DeInit 41

5.2.14 BSP_JOY_GetState . . . 42
5.2.15 BSP_POT_Init . . . 43
5.2.16 BSP_POT_GetLevel . . . 45
5.2.17 BSP_POT_DeInit . . . 45
5.2.18 BSP_POT_RegisterDefaultMspCallbacks . . . 46
5.2.19 BSP_POT_RegisterMspCallbacks . . . 47
5.2.20 Using IO expander . . . 48

6 BSP bus driver . . . 51
6.1 BSP bus driver APIs . . . 52
6.2 BSP bus APIs implementation . . . 53
6.2.1 BSP_PPPn_Init . . . 53
6.2.2 BSP bus MSP default functions . . . 54
6.2.3 BSP_PPPn_DeInit . . . 56
6.2.4 BSP_PPPn_ReadReg{16} . . . 57
6.2.5 BSP_PPPn_RegisterDefaultMspCallbacks . . . 57
6.2.6 BSP_PPPn_WriteReg{16} . . . 59
6.2.7 BSP_PPPn_ReadReg{16} . . . 60
6.2.8 BSP_PPPn_Send . . . 61
6.2.9 BSP_PPPn_Recv . . . 61
6.2.10 BSP_PPPn_SendRecv . . . 62
6.2.11 BSP_PPPn_IsReady . . . 62
6.3 BSP bus services . . . 63
6.4 BSP SPI bus specific services . . . 64
6.5 BSP I^2C bus specific services . . . 67
6.6 BSP bus multi configuration . . . 78
6.7 BSP bus customization . . . 79

7 BSP component driver . . . 81
7.1 BSP component register file . . . 82
7.1.1 Component register header file . . . 83
7.1.2 Component register source file . . . 83
7.2 BSP component core drivers . . . 85
7.2.1 Component core header file . . . 85
7.2.2 Component core source file . . . 91
7.3 Registering components . . . 93

7.4 BSP component class drivers 96
7.5 BSP memory component drivers 97
7.6 BSP component upgrade 101

8 BSP class driver 102
8.1 Generic common functions 103
8.1.1 BSP_CLASS_Init 103
8.1.2 BSP_CLASS_DeInit 111
8.1.3 BSP_CLASS_RegisterDefaultMspCallbacks 111
8.1.4 BSP_CLASS_RegisterMspCallbacks 111
8.1.5 RegisterMspCallbacks example 112
8.1.6 BSP_CLASS_Action 116
8.2 Specific functions 117
8.3 Extended functions 117
8.4 BSP classes 118
8.4.1 BSP AUDIO OUT class APIs 118
8.4.2 BSP AUDIO IN class APIs 121
8.4.3 BSP TS class APIs 123
8.4.4 BSP SD class APIs 125
8.4.5 BSP SDRAM class APIs 127
8.4.6 BSP SRAM class APIs 128
8.4.7 BSP NOR class APIs 129
8.4.8 BSP QSPI class APIs 130
8.4.9 BSP IO expander class APIs 133
8.4.10 BSP LCD class APIs 134
8.4.11 BSP camera class APIs 136
8.4.12 BSP EEPROM class APIs 139
8.4.13 BSP motion class APIs 139
8.4.14 BSP environmental class APIs 141
8.4.15 BSP OSPI class APIs 143
8.5 BSP class driver context 148
8.6 BSP class driver inter dependency 149
8.7 Using driver structure 150

9 BSP IRQ handlers 152
9.1 Generic rules 152

9.2 BSP IRQ Handlers implementation 154

10 BSP error management 156

10.1 Component drivers 157

10.2 BSP common drivers 157

10.3 BSP class drivers 157

10.4 BSP bus drivers 158

Revision history 159

List of tables

Table 1. Acronyms and definitions . . . 8
Table 2. General naming rules . . . 20
Table 3. Board naming rules . . . 21
Table 4. BSP common driver functions . . . 23
Table 5. BSP bus driver APIs . . . 52
Table 6. Most common communication services . . . 63
Table 7. BSP component drivers: object structure . . . 86
Table 8. BSP component drivers: IO structure . . . 88
Table 9. BSP class drivers: generic common functions . . . 103
Table 10. BSP class drivers: Specific functions . . . 117
Table 11. BSP class drivers: Extended functions . . . 117
Table 12. BSP AUDIO OUT class APIs . . . 118
Table 13. BSP AUDIO IN class APIs . . . 121
Table 14. Specific APIs for recording . . . 123
Table 15. BSP TS class APIs . . . 123
Table 16. BSP SD class APIs . . . 125
Table 17. BSP SDRAM class APIs . . . 127
Table 18. BSP SRAM class APIs . . . 128
Table 19. BSP NOR class APIs . . . 129
Table 20. BSP QSPI class APIs . . . 130
Table 21. BSP IOExpander class APIs . . . 133
Table 22. BSP LCD class APIs . . . 134
Table 23. BSP camera class APIs . . . 136
Table 24. BSP EEPROM class APIs . . . 139
Table 25. BSP motion class APIs . . . 139
Table 26. BSP environmental class APIs . . . 141
Table 27. BSP OSPI NOR class APIs . . . 143
Table 28. Advanced APIS for OSPI support additional features . . . 145
Table 29. BSP OSPI RAM class APIs . . . 146
Table 30. Examples of symbols . . . 152
Table 31. Document revision history . . . 159

List of figures

Figure 1. STM32Cube MCU and MPU Packages architecture . . . 11
Figure 2. STM32Cube MCU and MPU Packages detailed architecture . . . 11
Figure 3. BSP drivers overview . . . 12
Figure 4. BSP driver repository: single board . . . 13
Figure 5. BSP driver repository: multi-boards . . . 14
Figure 6. BSP driver repository: overview . . . 15
Figure 7. BSP driver repository: common class headers . . . 16
Figure 8. BSP driver repository: configuration files . . . 18
Figure 9. BSP inclusion model . . . 19
Figure 10. BSP driver repository: common drivers . . . 22
Figure 11. BSP driver repository: bus drivers . . . 51
Figure 12. BSP driver repository: component drivers . . . 81
Figure 13. BSP driver architecture . . . 82
Figure 14. BSP driver repository: memory component drivers . . . 97
Figure 15. BSP driver architecture: memory calling model . . . 98
Figure 16. Patchable component folder example . . . 101
Figure 17. BSP driver repository: function drivers . . . 102
Figure 18. BSP error management . . . 156

1 Documentation conventions

STM32Cube BSP drivers support Arm®(a)-based devices.

arm

1.1 About this document

This document mainly focuses on the BSP architecture, which is designed to be more generic by addressing multi instance components and multi buses and changing the initialization process to be based on the code generated by STM32CubeMX tool.

It details the architecture of BSP drivers (class, component, common and bus) and gives examples of implementation to guide the user developing BSP drivers on a given board.

The user of this document must be familiar with the STM32Cube MCU and MPU Packages.

1.2 Acronyms and definitions

Table 1. Acronyms and definitions

Acronym	Definition
API	application programming interface
BSP	board support package
CMSIS	Cortex microcontroller software interface standard
CPU	central processing unit
DMA	direct memory access
FMC	flexible memory controller
GPIO	general purpose IO
HAL	hardware abstraction layer
I2C	Inter-integrated circuit
LTDC	LCD TFT display controller
MSP	MCU specific package
SAI	serial audio interface
SD	secure digital
SDRAM	SDRAM external memory
SRAM	SRAM external memory
SMARTCARD	smartcard IC
SPI	serial peripheral interface

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

Table 1. Acronyms and definitions (continued)

Acronym	Definition
SysTick	System tick timer
PPP	Generic STM32 peripheral or block
Class/Function	A common functionality category with the same user interface (LCD, Touch Screen, SD…)
HMI	Human machine interface

2 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:
  - 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-Power (STM32CubeMonPwr), a monitoring tool to measure and help in the optimization of the power consumption of the MCU
- 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

3 BSP driver architecture

The STM32Cube MCU and MPU Packages architecture is described in *Figure 1* and *Figure 2*.

Figure 1. STM32Cube MCU and MPU Packages architecture

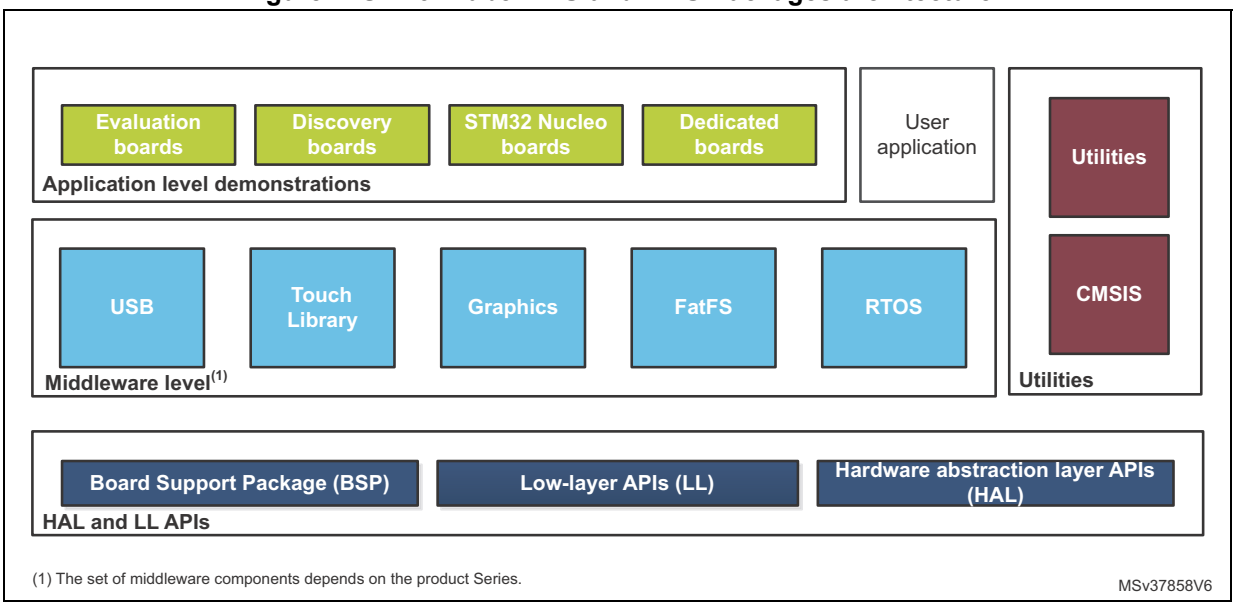
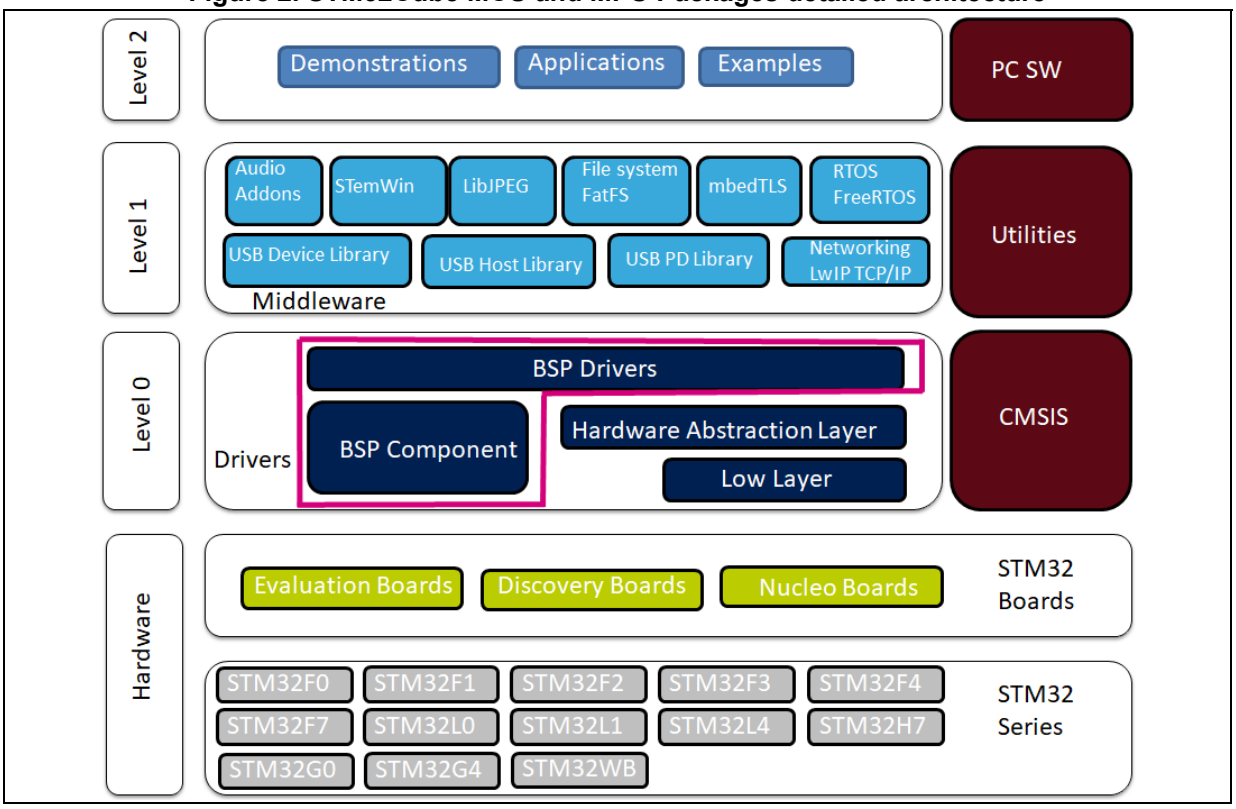


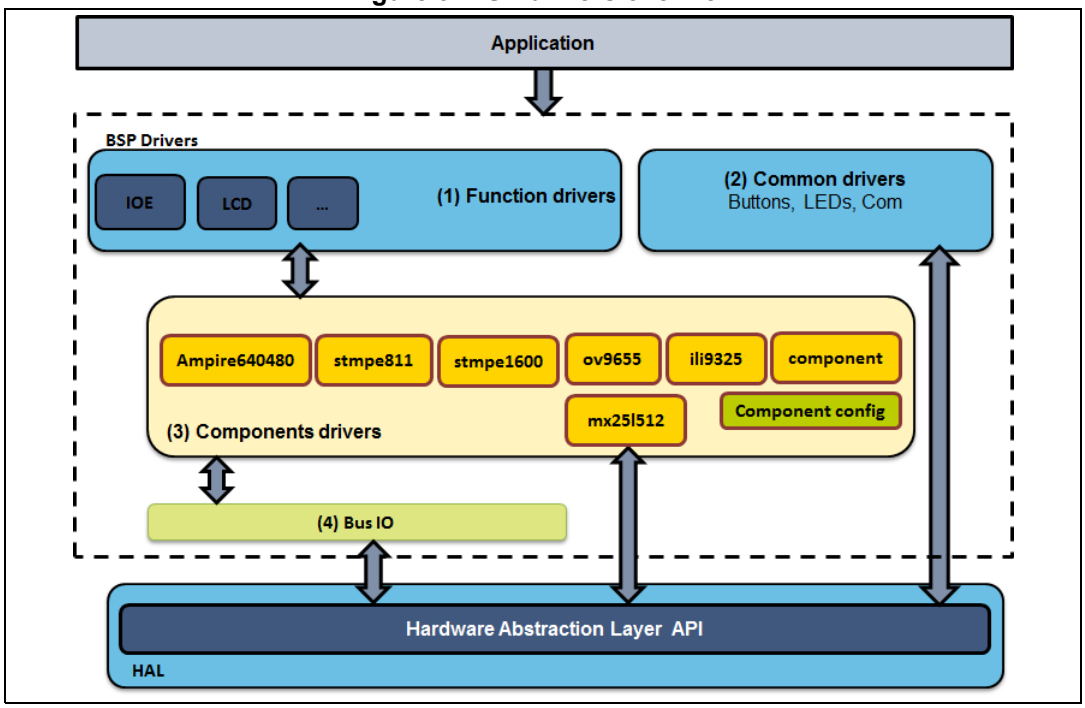
Figure 2. STM32Cube MCU and MPU Packages detailed architecture



The BSP drivers are generally composed of four parts, as presented in *Figure 3* and detailed here.

1. Function drivers: This part provides a set of high-level APIs for a specific class or functionality, such as LCD, Audio or Touchscreen.
2. Common driver: The common driver provides a set of friendly APIs for HMI (LEDs, buttons, and joysticks) and COM services.
3. Component drivers: This generic driver, for an external device on the board and independent of the HAL, is the component driver, which provides specific APIs to the external IC component and can be portable on any board. The component driver is composed of the component core files (*nnnxxxx.h* and *nnnxxxx.c*), component register files (*nnnxxxx_reg.h* and *nnnxxxx_reg.c*) and option configuration file (*nnnxxxx_conf.h*).
4. Bus IO driver: generic bus interface to provide the transport layer for the components, such as I^2C or SPI.

Figure 3. BSP drivers overview



3.1 BSP drivers features

The BSP architecture aims to provide the following features:

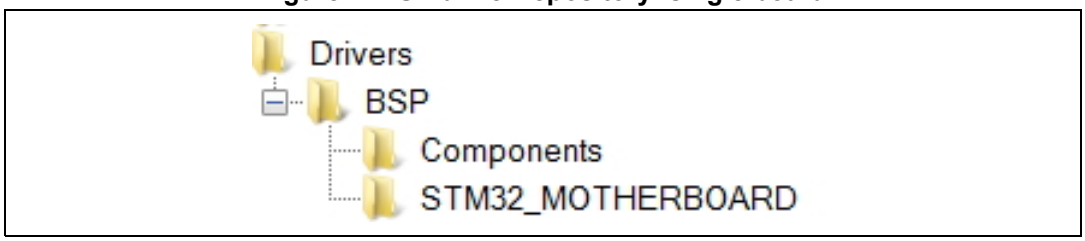
- Manage dynamically the MSP part of the generic class functions to customize the low-level part in order to prevent resources conflict between several class drivers.
- Support generic, specific and extended functions categories for BSP class drivers to fit in same time with native STM32 and expansion boards.
- Manage low power mode and allow either STM32 or external components to enter and exit low power mode defined by the user for more power control efficiency.
- Components driver flexibility allowing to customize the component class structure by adding more elementary functions for full external components control.
- Manage multi instance access to components drivers and guarantee unitary functions reentrancy by adding object handles
- Generic argument for each unitary component function to handle specific features offered by different components belonging to the same functionality class.
- Read register and Write registers basic component functions to allow handling more components features from application level.
- Generic BUS IO driver to centralize the low-level transport layer for the components
- Split the component driver into three layers:
  - Component core *driver.c* or *driver.h* files
  - Component register *driver_reg.c* or *driver_reg.h* files
  - Configuration file
- Flexible use DMA and IT HAL model based on the new HAL register callbacks features

3.2 Hardware configuration

The BSP architecture is designed to support the following hardware configurations:

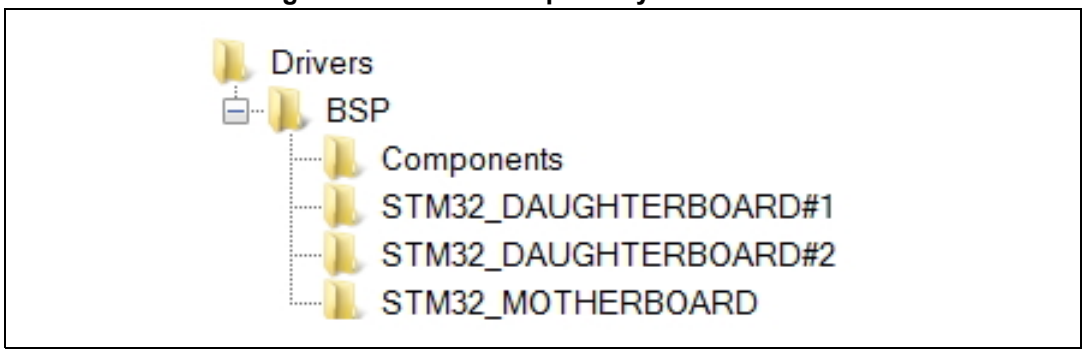
- Single board: The STM32 and all the used components are on the same board (mother board): Evaluation board, Discovery Kit, Nucleo, specific or user-defined board.

Figure 4. BSP driver repository: single board



- Multi boards: The STM32 and some used components are on the same board (mother board) while additional components are on one or several daughterboards linked to the mother board through GPIOs or communication buses.

Figure 5. BSP driver repository: multi-boards

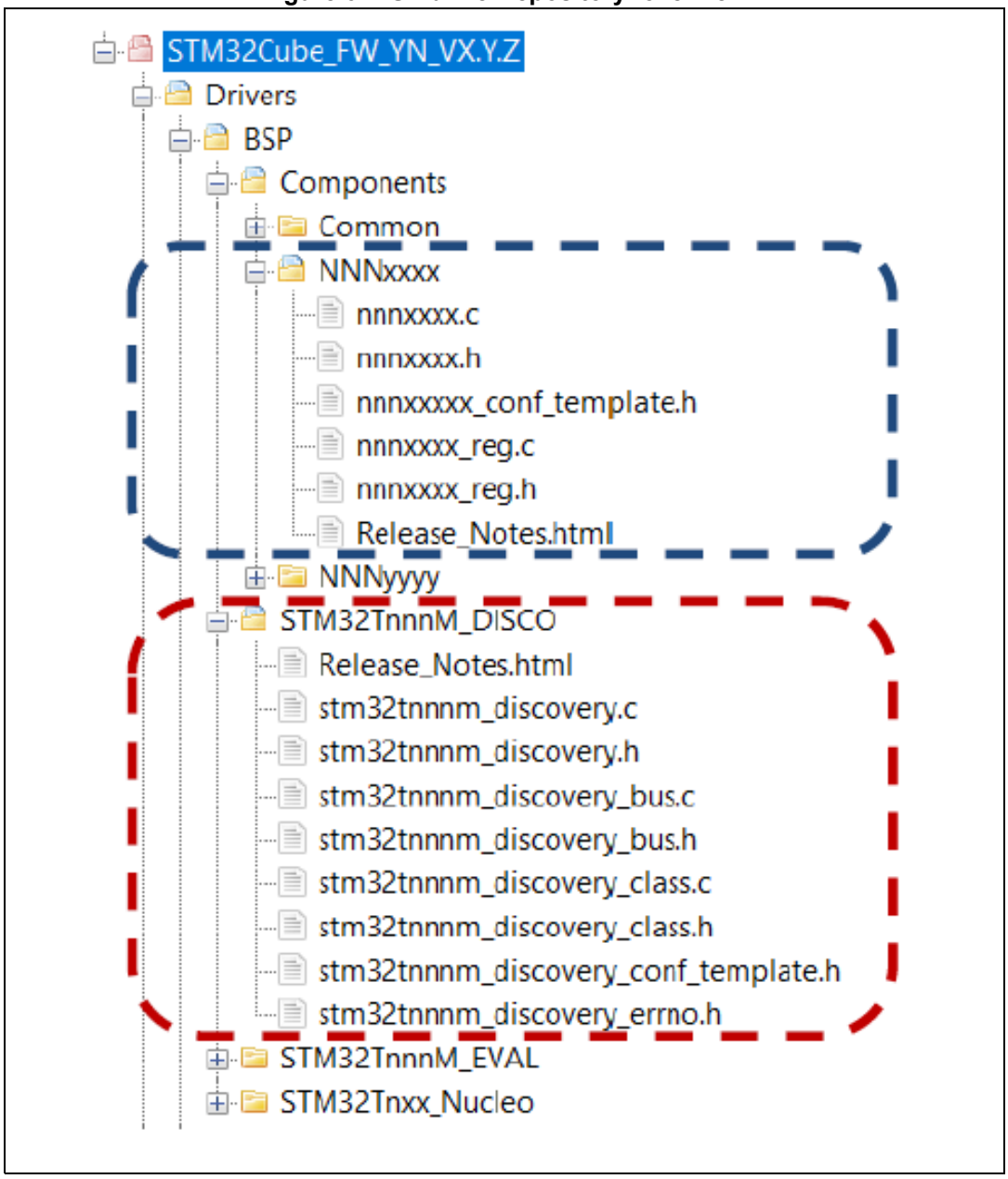


Note: The BSP class drivers that are exclusively provided by components on the daughterboard must be located in the daughterboard folder. The components on the daughterboard may be linked to the mother board through this formal bus IO services.

3.3 BSP driver repository

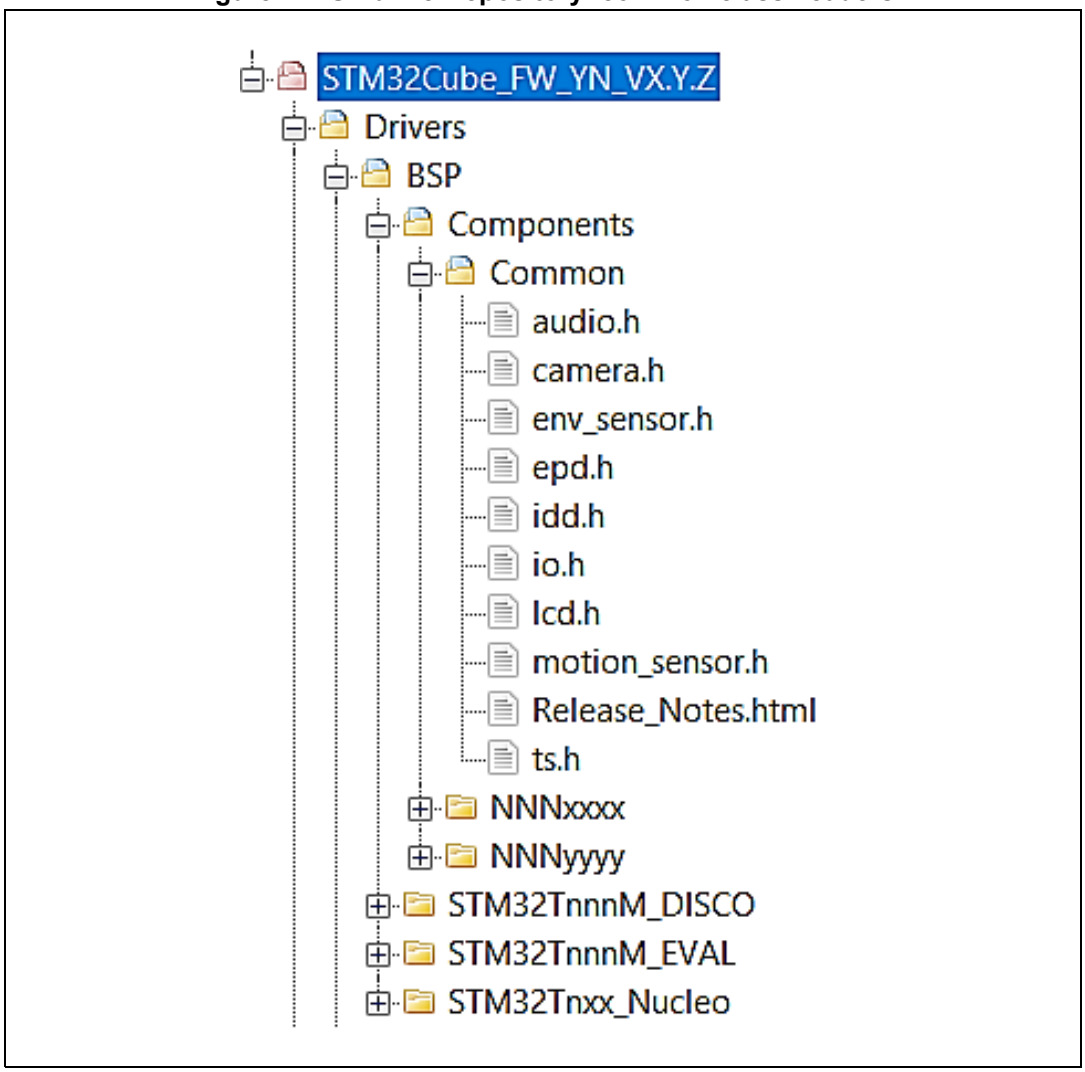
The BSP drivers are composed of component drivers that are located in a common folder for all the boards and board specific drivers that are located in a specific folder for each supported board within a STM32 family as shown in *Figure 6*.

Figure 6. BSP driver repository: overview



The components drivers of a same class are providing the same driver structure as defined in the component header of the class file (*class.h*) located in the *components**common* folder. The class corresponds to the generic functions categories, such as LCD, TS, IO and others.

Figure 7. BSP driver repository: common class headers

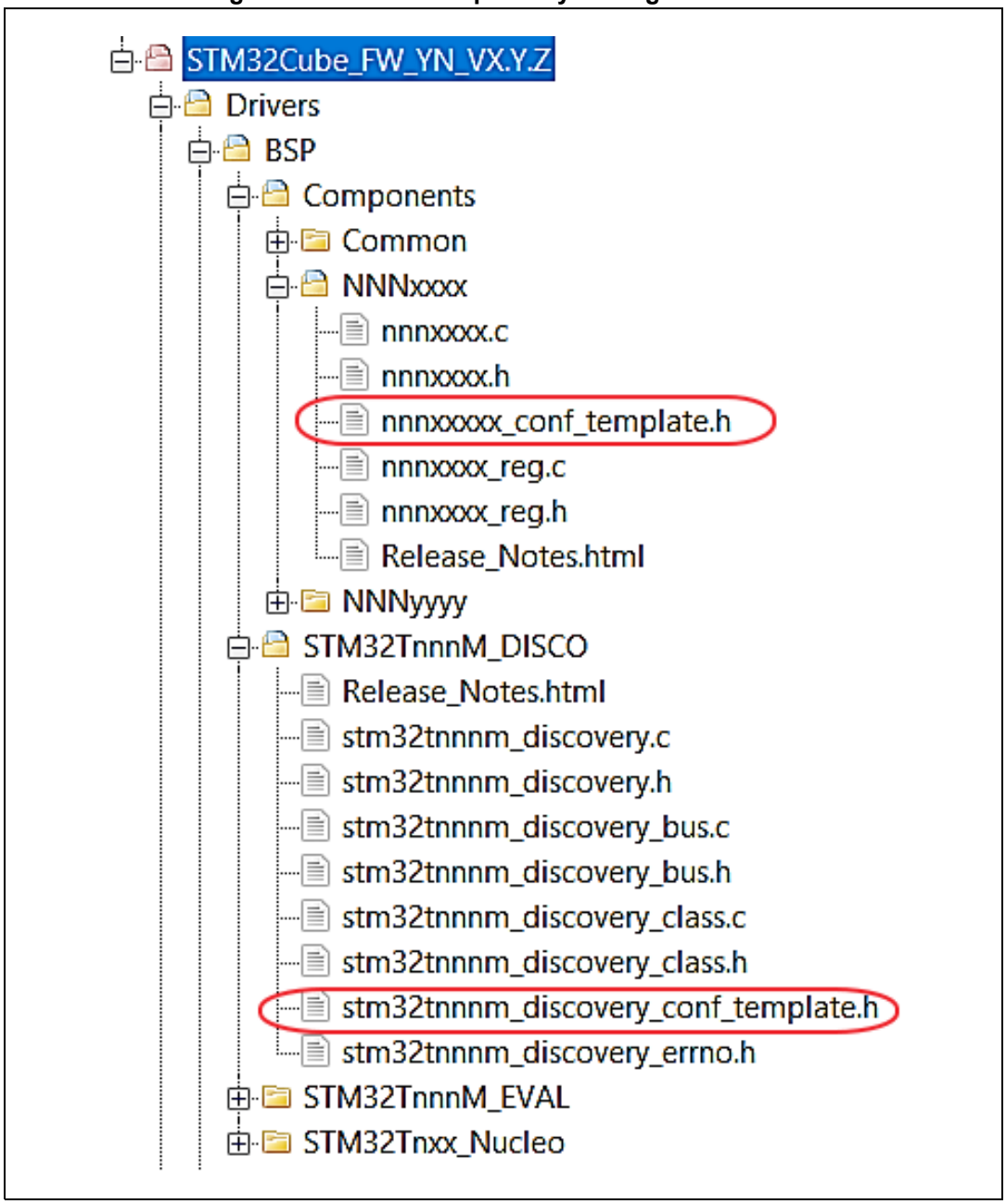


3.4 BSP based project repository

This BSP architecture offers two configuration file sets:

- Component configuration file: a component may require to be configured by the user to select a specific mode in compile time (Compile-time configurable drivers). In this case, the component folder contains a configuration file with the *_template* suffix. This file must be copied and renamed by removing the *_template* suffix in the user folder.
- BSP drivers configuration file: the common and the specific function drivers may need to be configured to select a specific mode or enable specific features. In this case, the board folder contains a configuration file with the *_template* suffix. This file must be copied and renamed by removing the *_template* suffix in the user folder.

Figure 8. BSP driver repository: configuration files



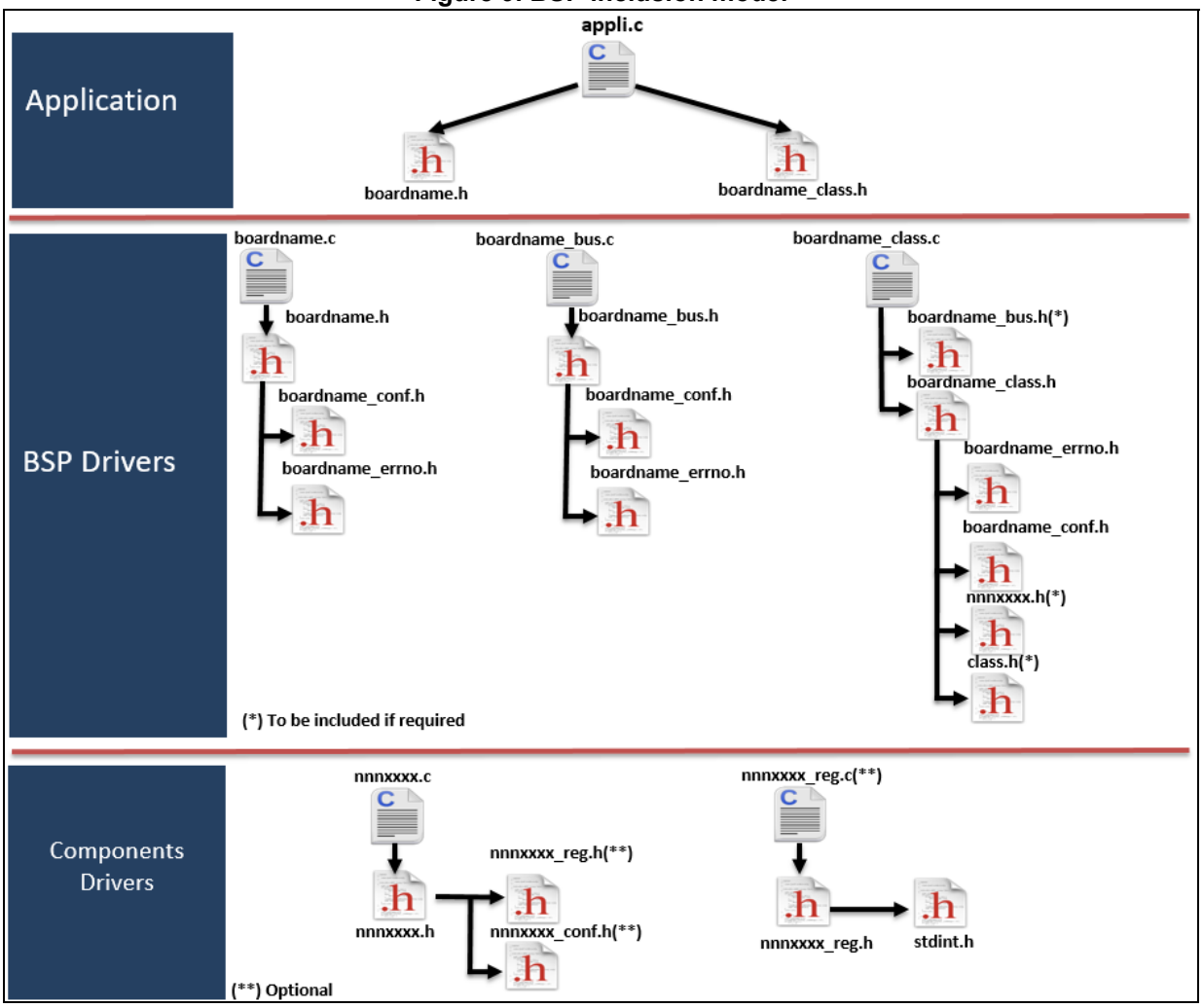
Note: 1 The component configuration files are optional. In some cases, they may be omitted when no specific configuration item needs to be customized.

Note: 2 The BSP drivers configuration file is mandatory and must be always added in the application include folder

3.5 BSP inclusion model

Figure 9 shows the different inclusion dependencies between the BSP drivers modules. From application side, the common and the class function drivers must be added independently in the workspace and the components and global BSP drivers configuration files must be added physically in the application include folder.

Figure 9. BSP inclusion model



Some BSP drivers services can be used by other BSP common or class driver, thus the former BSP drivers must be added as well in the project workspace. Example: IO BSP driver.

The component configuration files are not available for some components.

4 BSP naming rules

4.1 General rules

The following table summarizes the general naming rules regarding the files, functions and resources exported to the end user.

Table 2. General naming rules

-	ST Mother board	ST daughterboard	User specific board
File names	stm32tnnnm_boardtype{_Function} (c/h) like *stm32f413h_discovery.c/h, stm32f4xx_nucleo.c/h, stm32f769i_eval.c*	shield name {_Function} (c/h) like *iks01a2_env_sensors.c/h*	user board name{_Function} (c/h) like *sx1276mb1mas.c/h*
Exported Function name	BSP_CLASS_Function like `BSP_AUDIO_OUT_Init()`, `BSP_LED_Init()`	SHIELDNAME _Function like `ADAFRUIT_802_JOY_Init()` `IKS01A2_ENV_SENSOR_Init()`	USERBOARDNAME _CLASS_Function like `USERBOARDNAME_AUDIO_OUT_Init()` `USERBOARDNAME_LED_Init()`
Structure name	`CLASS_STRUCTNAME_Ctx_t` (for example `AUDIO_IN_Ctx_t`)		
Enum name	`{CATEGORY/CLASS}_EnumName_t` (for example `Button_TypeDef/COM_TypeDef`)		
defines	`CLASS_DEFINENAME` (for example `AUDIO_OUT_XXX`)		

Note: 1 The category refers to common BSP drivers feature classification ex: JOY, PB, COM, POT, LED…

Note: 2 For expansion/extension BSP drivers, the _ex suffix must be added in the files name and ex suffix must be added to the extension driver resources (functions, data structures, enumeration, defines…).

Note: 3 For Nucleo, whatever the type (32, 64, 144), we must always use the same common driver, if some resources are different, they must be delimited using the USE_NUCLEO_NN defined in the configuration file.

Note: 4 In case several variants of a Nucleo board type are available, a second level of delimitation is used based on the device name (USE_NUCLEO-TnnnMM), for example USE_NUCLEO_H7A3ZI.

Note: 5 The environment and motion sensors class are considered as macro classes as they support several unitary sensors subclasses (gyro, magneto...), thus the file names are ending with s but not the name of the class itself. For example, b_l475e_iot01a1_env_sensors.c or b_l475e_iot01a1_env_sensors.h are the file names, and the class name is ENV_SENSOR.

4.2 Board naming rules

The following table summarizes the rules and the difference between the CPN and the firmware used name.

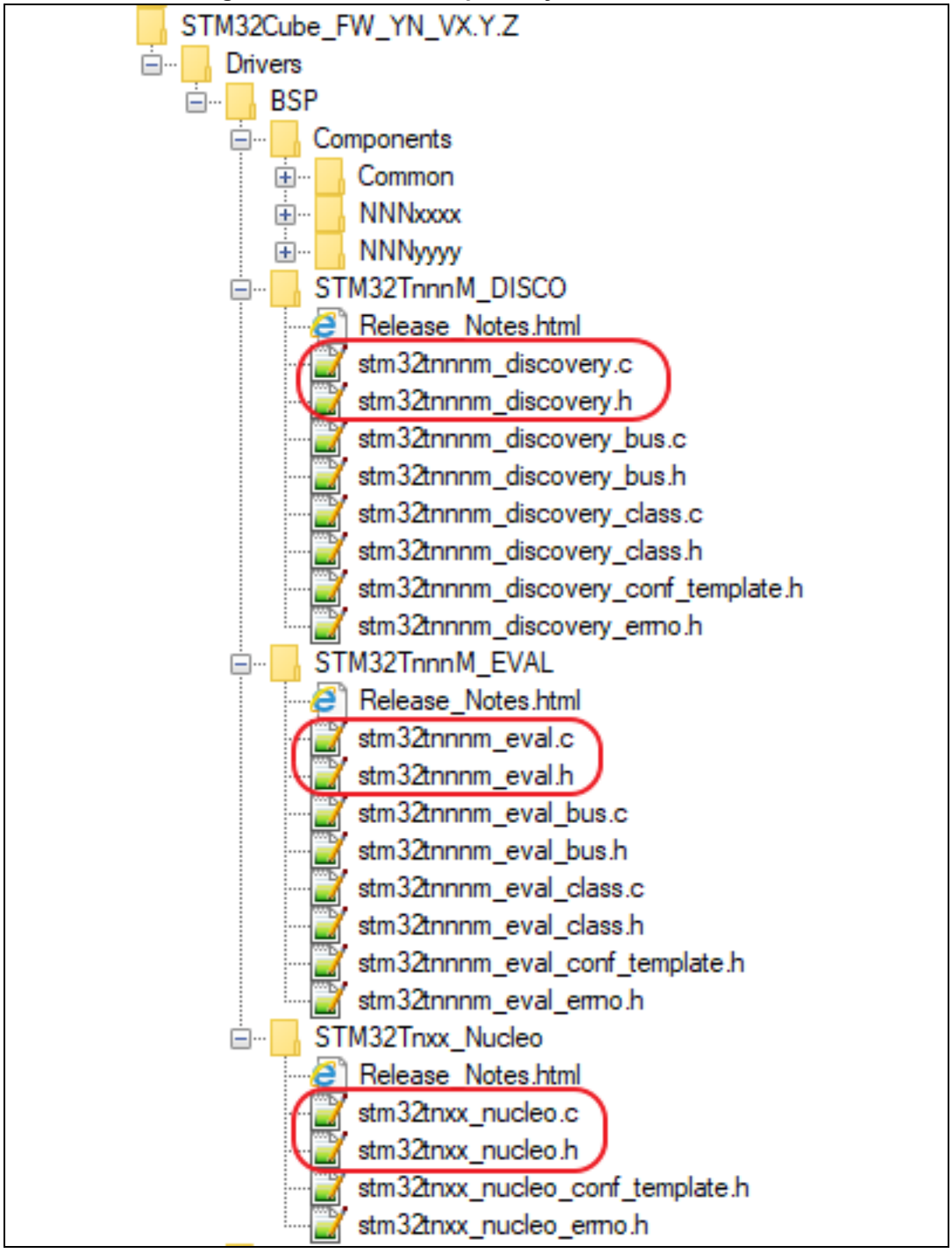
Table 3. Board naming rules

ST Board	Board Name	Filename	BSP Folder Name
Evaluation board	STM32TnnnM-EV/EVAL	stm32tnnnm_eval like *stm32f769i_eval*	STM32TnnnM-EVAL like *STMF769I-EVAL*
Discovery board	(Product focus) STM32TnnnM-DK/DISCO	stm32tnnnm_discovery like *stm32f769i_discovery*	STM32TnnnM-DISCO/DK like *STM32F769I-DISCO*
	(application focus) B-TxxxM-AAAyyT(z) B-TxxxM-AAAAAA(y)	b_txxxm_aaayyt(z) like *b_l475e_iot01a1* b_txxxm_aaaaaa(y) like *b_l072z_lrwan1*	B-TxxxM-AAAyyT(z) like B-L475E-IOT01A1 B-TxxxM-AAAAAA(y) like B-L072Z-LRWAN1
Nucleo board	NUCLEO-TnnnMM	stm32tnxx_nucleo like *stm32l4xx_nucleo*	STM32TNxx_Nucleo like *STM32L4xx_Nucleo*
Expansion board	{X/I}-NUCLEO-NNNNN	nnnnn like *iks01a2*	NNNNN like *IKS01A2*

5 BSP common driver

The common driver provides a set of friendly used APIs for HMI devices (LEDs, buttons, and joystick) and the board COM ports. It is available for all the supported boards within a specific series of STM32Cube MCU and MPU Packages.

Figure 10. BSP driver repository: common drivers



5.1 BSP common driver functions

The BSP common services are defined only for the corresponding hardware available on the board. *Table 4* gives the list of common services that must be implemented depending on the hardware:

Table 4. BSP common driver functions

Service	Description
int32_t BSP_GetVersion (void)	Return BSP version
int32_t BSP_LED_Init (Led_TypeDef Led)	Initialize a led
int32_t BSP_LED_DeInit (Led_TypeDef Led)	De-initialize a led
int32_t BSP_LED_On (Led_TypeDef Led)	Turn On a led
int32_t BSP_LED_Off (Led_TypeDef Led)	Turn Off a led
int32_t BSP_LED_Toggle (Led_TypeDef Led)	Toggle a led
int32_t BSP_LED_GetState (Led_TypeDef Led)	Return a led state
int32_t BSP_PB_Init (Button_TypeDef Button, ButtonMode_TypeDef Button_Mode)	Initialize and configure a button
int32_t BSP_PB_DeInit (Button_TypeDef Button)	De-initialize a Button
int32_t BSP_PB_GetState (Button_TypeDef Button)	Return a button state
void BSP_PB_Callback (Button_TypeDef Button)	Button callback when used in EXTI mode
void BSP_PB_IRQHandler (Button_TypeDef Button)	handles Button interrupt requests
int32_t BSP_COM_Init (COM_TypeDef COM, COM_InitTypeDef *COM_Init)	Initialize and configure COM port
int32_t BSP_COM_DeInit (COM_TypeDef COM)	De-Initialize COM port
int32_t BSP_COM_SelectLogPort (COM_TypeDef COM)	Select a COM port for the printf output Note: default COM Port is the first one (COM1)
int32_t BSP_COM_RegisterDefaultMspCallbacks (COM_TypeDef COM)	Register Default COM Msp Callbacks
int32_t BSP_COM_RegisterMspCallbacks (COM_TypeDef COM, BSP_COM_Cb_t *Callback)	Register specific user COM Msp Callback
int32_t BSP_JOY_Init (JOY_TypeDef JOY, JOYMode_TypeDef JoyMode, JOYPin_TypeDef JoyPins)	Initialize and configure the joystick
int32_t BSP_JOY_DeInit (JOY_TypeDef JOY, JOYPin_TypeDef JoyPins)	De-initialize a Joystick
Int32_t BSP_JOY_GetState (JOY_TypeDef JOY)	Return the joystick state
void BSP_JOY_Callback (JOY_TypeDef JOY, JOYPin_TypeDef JoyPins)	Joystick callback when used in EXTI mode
void BSP_JOY_IRQHandler (JOY_TypeDef JOY, JOYPin_TypeDef JoyPin)	handles JOY keys interrupt requests
int32_t BSP_POT_Init (POT_TypeDef POT)	Initialize and configure the Potentiometer resources

Table 4. BSP common driver functions (continued)

Service	Description
int32_t BSP_POT_DeInit (POT_TypeDef POT)	De-Initialize the Potentiometer resources
int32_t BSP_POT_RegisterDefaultMspCallbacks (POT_TypeDef POT)	Register Default Potentiometer Msp Callbacks
int32_t BSP_POT_RegisterMspCallbacks (POT_TypeDef POT, BSP_POT_Cb_t *Callback)	Register Potentiometer Msp Callbacks
int32_t BSP_POT_GetLevel (POT_TypeDef POT)	Return the Potentiometer Level [0 to 100%]

Note: The COM BSP driver must be always defined when the board embeds RS232 connector or when the board supports the VCP USB class through STLINK.

5.2 BSP common functions implementation

This section provides examples of common services code implementation. This must be adapted to match the board components.

5.2.1 BSP_GetVersion

This API returns the BSP driver revision. The version format is as follows: 0xXYZR (8 bits for each decimal, R for RC)

```
int32_t BSP_GetVersion (void)
{
  return (int32_t)BOARDNAME_BSP_VERSION;
}
```

The BOARDNAME_BSP_VERSION is defined in the common BSP driver header file as follows:

```
#define BOARDNAME_BSP_VERSION_X (int32_t) (0x02) /*!< [31:24] main version */
#define BOARDNAME_BSP_VERSION_Y (int32_t) (0x06) /*!< [23:16] sub1 version */
#define BOARDNAME_BSP_VERSION_Z (int32_t) (0x02) /*!< [15:8] sub2 version */
#define BOARDNAME_BSP_VERSION_RC (int32_t) (0x00) /*!< [7:0] release candidate */
#define BOARDNAME_BSP_VERSION ((BOARDNAME_BSP_VERSION_X << 24)\
|(BOARDNAME_BSP_VERSION_Y << 16)\
|(BOARDNAME_BSP_VERSION_Z << 8)\
|(BOARDNAME_BSP_VERSION_RC))
```

5.2.2 BSP_LED_Init

This API configures the user LED GPIOs (Power and components monitoring LEDs are not included). The number of LEDs (LEDn) is defined by the number of physical LEDs on the board.

```
int32_t BSP_LED_Init (Led_TypeDef Led)
{
  GPIO_InitTypeDef gpio_init_structure;

  /* Enable the GPIO_LED Clock */
    switch (Led)
    {
    case LED1:
      LED1_GPIO_CLK_ENABLE ();
      break;
    (…)
    case LEDN:
      LEDN_GPIO_CLK_ENABLE ();
      break;

    default:
      break;
}

  /* configure the GPIO_LED pin */
  gpio_init_structure.Pin = LED_PIN [Led];
  gpio_init_structure.Mode = GPIO_MODE_OUTPUT_PP;
  gpio_init_structure.Pull = GPIO_PULLUP;
  gpio_init_structure.Speed = GPIO_SPEED_FREQ_VERY_HIGH;
  HAL_GPIO_Init (LED_PORT [Led], &gpio_init_structure);

  HAL_GPIO_WritePin (LED_PORT [Led], LED_PIN [Led], GPIO_PIN_RESET);

  return BSP_ERROR_NONE;
}
```

The LEDs enumeration must be defined in the common BSP driver header file as follows:

```
typedef enum
{  LED1 = 0,
   LED_GREEN = LED1,
   LED2 = 1,
   LED_ORANGE = LED2,
   LED3 = 2,
   LED_RED = LED3,
   LED4 = 3,
   LED_BLUE = LED4,
   LEDn
} Led_TypeDef;
```

The different literals and defines must be defined in the common BSP driver header file as follows:

```
#define LEDi_LED_PORT                ((GPIO_TypeDef*) GPIOx)
#define LEDi_GPIO_CLK_ENABLE ()      __HAL_RCC_GPIOj_CLK_ENABLE ()
#define LEDi_PIN                     ((uint32_t) GPIO_PIN_M)
Where 0<= i < N and j = {A, B …}
N: Number of LEDs
{A, B …} GPIO Port}
```

The PIN and GPIO arrays are defined in the BSP common driver source file as follows:

```
int32_t LED_PIN [LEDn] = {LED1_PIN,
                          LEDi_PIN,
                          …,
                          LEDN_PIN};

GPIO_TypeDef* LED_PORT [LEDn] = {LED1_GPIO_PORT,
                                 LEDi_GPIO_PORT,
                                 …,
                                 LEDN_GPIO_PORT};
```

5.2.3 BSP_LED_DeInit

This API turns off the user LED and de-initialize the GPIO peripheral registers to their default reset values.

```
int32_t BSP_LED_DeInit(Led_TypeDef Led)
{
   GPIO_InitTypeDef gpio_init_structure;

   /* DeInit the GPIO_LED pin */
   gpio_init_structure.Pin = LED_PIN [Led];

   /* Turn off LED */
   HAL_GPIO_WritePin (LED_PORT [Led], LED_PIN[Led], GPIO_PIN_SET);
   HAL_GPIO_DeInit (LED_PORT [Led], gpio_init_structure.Pin);
   return BSP_ERROR_NONE;
}
```

Note: The BSP_LED_DeInit must not disable the clock of the GPIO port, because it may be used by other modules in the application

5.2.4 BSP_LED_On

This API turns the selected LED on.

```
int32_t BSP_LED_On(Led_TypeDef Led)
{
    HAL_GPIO_WritePin (LED_PORT [Led], LED_PIN [Led], GPIO_PIN_RESET);
    return BSP_ERROR_NONE;
}
```

5.2.5 BSP_LED_Off

This API turns the selected LED off.

```
int32_t BSP_LED_On(Led_TypeDef Led)
{
    HAL_GPIO_WritePin (LED_PORT [Led], LED_PIN [Led], GPIO_PIN_SET);
    return BSP_ERROR_NONE;
}
```

Note: The GPIO PIN state, GPIO_PIN_SET / GPIO_PIN_RESET, used to turn on/off the LEDs depends on the schematics of the board (the GPIO may drive the anode or the cathode of the LED).

5.2.6 BSP_LED_Toggle

This API toggles the state of the selected LED.

```
int32_t BSP_LED_Toggle (Led_TypeDef Led)
{
   HAL_GPIO_TogglePin (LED_PORT [Led], LED_PIN [Led]);
   return BSP_ERROR_NONE;
}
```

5.2.7 BSP_LED_GetState

This API returns the state of the selected LED.

```
int32_t BSP_LED_GetState (Led_TypeDef Led)
{
     return int32_t)(HAL_GPIO_ReadPin (LED_PORT [Led], LED_PIN [Led]) ==
GPIO_PIN_RESET);
}
```

Note: The GPIO PIN state, GPIO_PIN_SET / GPIO_PIN_RESET, used to turn on/off the LEDs depends on the schematics of the board (the GPIO may drive the anode or the cathode of the LED).

The LEDs are generally driven by GPIOs, on the same port or on different ports. If the same GPIO port is used, the switch case may be simply omitted.

```
int32_t BSP_LED_Init(Led_TypeDef Led)
{
  GPIO_InitTypeDef gpio_init_structure = {0};

  /* Enable the GPIO_LED Clock */
  LEDx_GPIO_CLK_ENABLE ();

  /* configure the GPIO_LED pin */
  gpio_init_structure.Pin = LED_PIN [Led];
  gpio_init_structure.Mode = GPIO_MODE_OUTPUT_PP;
  gpio_init_structure.Pull = GPIO_PULLUP;
  gpio_init_structure.Speed = GPIO_SPEED_FREQ_VERY_HIGH;
  HAL_GPIO_Init (LED_PORT [Led], &gpio_init_structure);

  HAL_GPIO_WritePin (LED_PORT [Led], LED_PIN [Led], GPIO_PIN_RESET);
  return BSP_ERROR_NONE;
}
```

Where the LEDx_GPIO_CLK_ENABLE is defined in the common BSP driver header file as follows:

```
#define LEDx_LED_PORT                       ((GPIO_TypeDef*) GPIOj)
#define LEDx_GPIO_CLK_ENABLE ()             __HAL_RCC_GPIOx_CLK_ENABLE ()
#define LEDi_PIN                            ((uint32_t) GPIO_PIN_M)

Where 0<= i < N and j = {A, B …}
N: Number of LEDs
{A, B …} GPIO Port
```

5.2.8 BSP_PB_Init

This API configures the push-button GPIOs and EXTI Line if the button must be used in interrupt mode.

```
int32_t BSP_PB_Init(Button_TypeDef Button, ButtonMode_TypeDef ButtonMode)
{
  GPIO_InitTypeDef gpio_init_structure;
  static BSP_EXTI_LineCallback ButtonCallback[BUTTONn] =
{BUTTON_USER_EXTI_Callback};
  static uint32_t  BSP_BUTTON_PRIO [BUTTONn] =
{BSP_BUTTON_USER_IT_PRIORITY};
  static const uint32_t BUTTON_EXTI_LINE[BUTTONn]    =
{BUTTON_USER_EXTI_LINE};

  /* Enable the GPIO_LED Clock */
  switch (Button)
  {
  case BUTTON_TAMPER:
   BUTTON_TAMPER_GPIO_CLK_ENABLE ();
  break;
  (…)
  case BUTTON_USER:
   BUTTON_USER_GPIO_CLK_ENABLE ();
  break;
  default:
  break;
  }

gpio_init_structure.Pin = BUTTON_PIN [Button];
  gpio_init_structure.Pull = GPIO_NOPULL;
  gpio_init_structure.Speed = GPIO_SPEED_FREQ_HIGH;

  if (ButtonMode == BUTTON_MODE_GPIO)
  {
    /* Configure Button pin as input */
    gpio_init_structure.Mode = GPIO_MODE_INPUT;
    HAL_GPIO_Init(BUTTON_PORT [Button], &gpio_init_structure);
  }
  else /* (ButtonMode == BUTTON_MODE_EXTI) */
  {
/* Configure Button pin as input with External interrupt */
    gpio_init_structure.Mode = GPIO_MODE_IT_RISING;

HAL_GPIO_Init(BUTTON_PORT[Button], &gpio_init_structure);
```

```
    (void)HAL_EXTI_GetHandle(&hpb_exti[Button], BUTTON_EXTI_LINE[Button]);
    (void)HAL_EXTI_RegisterCallback(&hpb_exti[Button],
HAL_EXTI_COMMON_CB_ID, ButtonCallback[Button]);

    /* Enable and set Button EXTI Interrupt to the configurable priority */
    HAL_NVIC_SetPriority((BUTTON_IRQn[Button]), BSP_BUTTON_PRIO[Button],
0x00);
    HAL_NVIC_EnableIRQ((BUTTON_IRQn[Button]));
  }

  return BSP_ERROR_NONE;
}
```

The buttons enumerations must be defined in the common BSP driver header file as follows:

```
typedef enum
{
  BUTTON_WAKEUP = 0,
  BUTTON_TAMPER = 1,
  BUTTON_USER = 2,
  BUTTONn
} Button_TypeDef;
Typedef enum
{
 BUTTON_MODE_GPIO = 0,
 BUTTON_MODE_EXTI = 1
} ButtonMode_TypeDef;
```

The different literals and defines must be defined in the common BSP driver header file as follows:

```
#define BUTTON_NAME_LED_PORT                ((GPIO_TypeDef*) GPIOx)
#define BUTTON_NAME_GPIO_CLK_ENABLE ()       __HAL_RCC_GPIOj_CLK_ENABLE ()
#define BUTTON_NAME_PIN                     ((uint32_t) GPIO_PIN_M)
BUTTON_NAME = BUTTON_WAKEUP or
BUTTON_TAMPER or
BUTTON_USER
N: Number of Buttons
x = {A, B …} GPIO Port
```

The buttons EXTI Lines IRQ priorities are defined in the in the *stm32YNNN_eval_conf.h* file.

```
#define BSP_BUTTON_NAME_IT_PRIORITY           0x0FU
```

The priority, PIN and GPIO arrays are defined in the BSP common driver C file:

```
uint32_t BSP_BUTTON_PRIO [BUTTONn] = {
                              …,
                              BSP_BUTTON_NAME_IT_PRIORITY,
                              …,
                                    };
```

5.2.9 BSP_PB_GetState

This API returns the state of the selected button.

```
int32_t BSP_PB_GetState (Button_TypeDef Button)
{
     return (int32_t)HAL_GPIO_ReadPin(BUTTON_PORT[Button],
BUTTON_PIN[Button]);
}
```

Note: The GPIO PIN state, GPIO_PIN_SET / GPIO_PIN_RESET, used to return the button state depends on the schematics of the board.

5.2.10 BSP_COM_Init

This API configures the COM port available on the board. This function may be used to initialize the LOG via the COM port feature when the `USE_BSP_COM_FEATURE` and `USE_COM_LOG` definitions are defined in the BSP configuration file, or prepares one of the board COM port for the user transfer.

```
UART_HandleTypeDef hcom_uart [COMn];
#if (USE_BSP_COM_FEATURE > 0)
#if (USE_COM_LOG > 0)
static COM_TypeDef COM_ActiveLogPort = COM1;
#endif
#endif
int32_t BSP_COM_Init (COM_TypeDef COM, COM_InitTypeDef *COM_Init)
{
int32_t ret = BSP_ERROR_NONE;

  if(COM >= COMn)
  {
    ret = BSP_ERROR_WRONG_PARAM;
  }
  else
  {
#if (USE_HAL_UART_REGISTER_CALLBACKS == 0)
    /* Init the UART Msp */
    UARTx_MspInit(&hcom_uart[COM]);
#else
    if(IsComMspCbValid[COM] == 0U)
    {
      if(BSP_COM_RegisterDefaultMspCallbacks(COM) != BSP_ERROR_NONE)
      {
        return BSP_ERROR_MSP_FAILURE;
      }
    }
#endif

    if(MX_USARTx_Init(&hcom_uart[COM], COM_Init) != HAL_OK)
    {
      ret = BSP_ERROR_PERIPH_FAILURE;
    }
  }

  return ret;
}
```

The COMs enumeration must be defined in the common BSP driver header file as follows:

```
typedef enum
{
  COM1 = 0, (…)
  COMn
} COM_TypeDef;
```

The different literals and defines must be defined in the common BSP driver header file as follows:

```
#define COMi_UART                              USARTi
#define COMi_TX_GPIO_PORT                      ((GPIO_TypeDef*) GPIOx)
#define COMi_TX_GPIO_CLK_ENABLE ()              __HAL_RCC_GPIOj_CLK_ENABLE ()
#define COMi_TX_PIN                            ((uint32_t) GPIO_PIN_M)
#define COMi_TX_AF                             GPIO_AFx_USARTy
#define COMi_RX_GPIO_PORT                      ((GPIO_TypeDef*) GPIOx)
#define COMi_RX_GPIO_CLK_ENABLE ()              __HAL_RCC_GPIOj_CLK_ENABLE ()
#define COMi_RX_PIN                            ((uint32_t) GPIO_PIN_M)
#define COMi_CLK_ENABLE ()                      __HAL_RCC_GPIOk_CLK_ENABLE ()
#define COM1_TX_AF                             GPIO_AFx_USARTYy

Where 0<= i < N and j, k = {A, B …}
N: Number of COMs
{A, B …} GPIO Port
```

The arrays' definition are defined in the BSP common driver source file as follows:

```
USART_TypeDef* COM_USART  [COMn] = {COMi_UART};
GPIO_TypeDef* COM_TX_PORT [COMn] = {COMi_TX_GPIO_PORT};
GPIO_TypeDef* COM_RX_PORT [COMn] = {COMi_RX_GPIO_PORT};
const uint16_t COM_TX_PIN [COMn] = {COMi_TX_PIN};
const uint16_t COM_RX_PIN [COMn] = {COMi_RX_PIN};
const uint16_t COM_TX_AF  [COMn] = {COMi_TX_AF};
const uint16_t COM_RX_AF  [COMn] = {COMi_RX_AF};
```

5.2.11 BSP_COM_SelectLogPort

This API selects a COM port for the printf output when the `USE_COM_LOG` definition is activated in the BSP configuration file.

```
#if (USE_COM_LOG > 0)
/**
  * @brief  Select the active COM port.
  * @param  COM COM port to be activated.
  *          This parameter can be COM1
  * @retval BSP status
  */
int32_t BSP_COM_SelectLogPort(COM_TypeDef COM)
{
  if(COM_ActiveLogPort != COM)
  {
    COM_ActiveLogPort = COM;
  }

  return BSP_ERROR_NONE;
}
#endif
```

Note:1 *If the first Port COM is used, the BSP_COM_SelectLogPort may not be called.*

Note:2 *If there is only one COM port (COMn = 1), the BSP_COM_SelectLogPort function may be omitted.*

To be able to use the COM port for the printf output the following `putc` and `putchar` functions must be implemented to override the default `putc` and `putchar` behavior.

```
#if (USE_COM_LOG > 0)
#ifdef __GNUC__
 int __io_putchar (int ch)
 #else
 int fputc (int ch, FILE *f)
 #endif /* __GNUC__ */
{
  (void)HAL_UART_Transmit (&hcom_uart [COM_ActiveLogPort], (uint8_t *)
&ch, 1, COM_POLL_TIMEOUT);
  return ch;
}
#endif /* USE_COM_LOG */
```

5.2.12 BSP_JOY_Init

This API configures the joystick GPIOs and EXTI Line if the button must be used in interrupt mode.

```
/* Store Joystick pins initialized */
uint32_t JoyPinsMask = 0;

int32_t BSP_JOY_Init (JOY_TypeDef JOY, JOYMode_TypeDef JoyMode,
JOYPin_TypeDef JoyPins)
{
  uint32_t joykey, key_pressed;
  GPIO_InitTypeDef gpio_init_structure;

static BSP_EXTI_LineCallback JoyCallback[JOY_KEY_NUMBER] = {
JOY1_SEL_EXTI_Callback,
JOY1_DOWN_EXTI_Callback,
JOY1_LEFT_EXTI_Callback,
JOY1_RIGHT_EXTI_Callback,
JOY1_UP_EXTI_Callback
};
  static const uint32_t JOY_EXTI_LINE[JOY_KEY_NUMBER]   = {
JOY1_SEL_EXTI_LINE,
JOY1_DOWN_EXTI_LINE,
JOY1_LEFT_EXTI_LINE,
JOY1_RIGHT_EXTI_LINE,
JOY1_UP_EXTI_LINE
};
  /* Store Joystick pins initialized */
  JoyPinsMask |= JoyPins;

  /* Initialized the Joystick. */
  for (joykey = 0; joykey < JOY_KEY_NUMBER; joykey++)
  {
    key_pressed = 1 << joykey;
    if (key_pressed & JoyPins)
    {
      if (JOY == JOY1)
      {
      /* Enable the JOY clock */
      JOY1_GPIO_CLK_ENABLE(key_pressed);

      gpio_init_structure.Pin = JOY1_PIN[joykey];
      gpio_init_structure.Pull = GPIO_PULLDOWN;
      gpio_init_structure.Speed = GPIO_SPEED_FREQ_VERY_HIGH;
```

```
 if (JoyMode == JOY_MODE_GPIO)
      {
        /* Configure Joy pin as input */
        gpio_init_structure.Mode = GPIO_MODE_INPUT;
        HAL_GPIO_Init(JOY1_PORT[joykey], &gpio_init_structure);
      }
      else if (JoyMode == JOY_MODE_EXTI)        {
        /* Configure Joy pin as input with External interrupt */
        gpio_init_structure.Mode = GPIO_MODE_IT_RISING;
        HAL_GPIO_Init(JOY1_PORT[joykey], &gpio_init_structure);

(void)HAL_EXTI_GetHandle(&hjoy_exti[joykey], JOY_EXTI_LINE[joykey]);
(void)HAL_EXTI_RegisterCallback(&hjoy_exti[joykey],
HAL_EXTI_COMMON_CB_ID, JoyCallback[joykey]);
        /* Enable and set Joy EXTI Interrupt to the lowest priority */
 HAL_NVIC_SetPriority((IRQn_Type)(JOY1_IRQn[joykey]),
BSP_JOY1_PRIO[joykey], 0x00);
        HAL_NVIC_EnableIRQ((IRQn_Type)(JOY1_IRQn[joykey]));
      }
      }
      else if (JOY == JOYx)
      {
      }
    }
  }

 return BSP_ERROR_NONE;
}
```

The joystick mode enumeration and keys must be defined in the common BSP driver header file as follows:

```
typedef enum
{
 JOY_NONE  = 0x00U,
 JOY_SEL   = 0x01U,
 JOY_DOWN  = 0x02U,
 JOY_LEFT  = 0x04U,
 JOY_RIGHT = 0x08U,
 JOY_UP    = 0x10U,
 JOY_ALL   = 0x1FU
} JOYPin_TypeDef;
#define JOY_KEY_NUMBER         5U

typedef enum {
  JOY_MODE_GPIO = 0,
  JOY_MODE_EXTI = 1
}JOYMode_TypeDef;
```

The JOY enumeration must be defined in the common BSP driver header file as follows:

```
typedef enum
 {
   JOY1 = 0, (…)
   JOYn
} JOY_TypeDef;
```

The different literals and defines must be defined in the common BSP driver header file as follows:

```
#define JOYx_NAME_PIN                          ((uint32_t) GPIO_PIN_M)
#define JOYx_NAME_GPIO_PORT                    ((GPIO_TypeDef*) GPIOy)
#define JOYx_NAME_GPIO_CLK_ENABLE ()           __HAL_RCC_GPIOj_CLK_ENABLE ()
#define JOYx_NAME_GPIO_CLK_DISABLE ()          __HAL_RCC_GPIOj_CLK_DISABLE ()
#define JOYx_NAME_EXTI_IRQn                    EXTIj_IRQn
JOY_NAME = JO_SEL ...JOY_UP
N: Number of Joystick instances
y = {A, B …} GPIO Port
x: Joystick instance
```

The Joystick EXTI Lines IRQ priorities are defined in the *boardname_conf.h* file.

```
#define BSP_JOYx_NAME_IT_PRIORITY           0x0FU
```

The priority, PIN and GPIO arrays are defined in the BSP common driver C file:

```
uint32_t BSP_JOYx_PRIO[JOY_KEY_NUMBER] =   {BSP_JOYx_SEL_IT_PRIORITY,
                                            BSP_JOYx_DOWN_IT_PRIORITY,
                                            BSP_JOYx_LEFT_IT_PRIORITY,
                                            BSP_JOYx_RIGHT_IT_PRIORITY,
                                            BSP_JOYx_UP_IT_PRIORITY};

static GPIO_TypeDef* JOYx_PORT[JOY_KEY_NUMBER] = {JOYx_SEL_GPIO_PORT,
                                                  JOYx_DOWN_GPIO_PORT,
                                                  JOYx_LEFT_GPIO_PORT,
                                                  JOYx_RIGHT_GPIO_PORT,
                                                  JOYx_UP_GPIO_PORT
                                                 };

static const uint16_t JOYx_PIN[JOY_KEY_NUMBER] = {JOYx_SEL_PIN,
                                                  JOYx_DOWN_PIN,
                                                  JOYx_LEFT_PIN,
                                                  JOYx_RIGHT_PIN,
                                                  JOYx_UP_PIN
                                                 };

static const IRQn_Type JOYx_IRQn[JOY_KEY_NUMBER] = {JOYx_SEL_EXTI_IRQn,
                                                    JOYx_DOWN_EXTI_IRQn,
                                                    JOYx_LEFT_EXTI_IRQn,
                                                    JOYx_RIGHT_EXTI_IRQn,
                                                    JOYx_UP_EXTI_IRQn
                                                   };
```

5.2.13 BSP_JOY_DeInit

This API de-initializes the GPIO peripheral registers to their default reset values.

```
int32_t BSP_JOY_DeInit(JOY_TypeDef JOY, JOYPin_TypeDef JoyPins)
{
  uint32_t joykey, key_pressed;

  /* Store Joystick pins initialized */
  JoyPinsMask &= JoyPins;

  /* Initialized the Joystick. */
  for(joykey = 0; joykey < JOY_KEY_NUMBER; joykey++)
  {
    key_pressed = 1 << joykey;
    if (key_pressed & JoyPins)
    {
      if (JOY == JOY1)
      {
        HAL_GPIO_DeInit(JOY1_PORT[joykey], JOY1_PIN[joykey]);
      }
      else if (JOY == JOYx)
      {
      }
    }
  }
 return BSP_ERROR_NONE;
}
```

Note: The BSP_JOY_DeInit must not disable the clock of the GPIO ports, because they may be used by other modules in the application.

5.2.14 BSP_JOY_GetState

This API returns the current joystick state and indicates the joystick key that was pressed.

```
int32_t BSP_JOY_GetState (JOY_TypeDef JOY)
{
  uint32_t joykey, key_pressed;

  for (joykey = 0; joykey < JOY_KEY_NUMBER; joykey++)
  {
    key_pressed = 1 << joykey;
    if (key_pressed & JoyPinsMask)
    {
      if (JOY == JOY1)
      {
        if (HAL_GPIO_ReadPin(JOY1_PORT[joykey], JOY1_PIN[joykey]) !=
GPIO_PIN_RESET)
        {
          /* Return Code Joystick key pressed */
          return key_pressed;
        }
      }
      else if (JOY == JOYx)
      {
      }
    }
  }

  /* No Joystick key pressed */
  return JOY_NONE;
}
```

5.2.15 BSP_POT_Init

This API configures the ADC channel associated with the potentiometer.

```
ADC_HandleTypeDef hpot_adc [POTn];
int32_t BSP_POT_Init(POT_TypeDef POT)
{
  int32_t ret = BSP_ERROR_NONE;

  if(POT >= POTn)
  {
    ret = BSP_ERROR_WRONG_PARAM;
  }
  else
  {
#if (USE_HAL_ADC_REGISTER_CALLBACKS == 0)
    /* Init the ADC Msp */
    ADCx_MspInit(&hpot_adc[POT]);
#else
    if(IsPotMspCbValid[POT] == 0U)
    {
      if(BSP_POT_RegisterDefaultMspCallbacks(POT) != BSP_ERROR_NONE)
      {
        return BSP_ERROR_MSP_FAILURE;
      }
    }
#endif

    if(MX_ADCx_Init(&hpot_adc[POT]) != HAL_OK)
    {
      ret = BSP_ERROR_PERIPH_FAILURE;
    }
  }

  return ret;
}
```

The POTs enumeration must be defined in the common BSP driver header file as follows:

```
typedef enum
{
  POT1 = 0, (…)
  POTn
} POT_TypeDef;
```

The different literals and defines must be defined in the common BSP driver header file as follows:

```
#define POTi_ADC                          ADCx
#define POTi_ADC_CHANNEL_GPIO_PORT        ((GPIO_TypeDef*) GPIOx)
#define POTi_CHANNEL_GPIO_CLK_ENABLE()    __HAL_RCC_GPIOj_CLK_ENABLE()
#define POTi_CHANNEL_GPIO_PIN             ((uint32_t) GPIO_PIN_M)
#define POTi_CLK_ENABLE()                 __HAL_RCC_GPIOk_CLK_ENABLE()

Where 0<= i < N and j, k = {A, B …}
N: Number of POTs
{A, B …} GPIO Port
```

The arrays definition are defined in the BSP common driver source file as follows:

```
ADC_TypeDef* POT_ADC [POTn] = {POTi_ADC};
GPIO_TypeDef* POT_CHANNEL_GPIO_PORT [POTn] = {POTi_TX_GPIO_PORT};
const uint16_t POT_CHANNEL_GPIO_PIN [POTn] = {POTi_TX_PIN};
```

5.2.16 BSP_POT_GetLevel

This API returns the level of voltage on potentiometer in percentage.

```
int32_t BSP_POT_GetLevel(POT_TypeDef POT)
{
  int32_t ret = BSP_ERROR_PERIPH_FAILURE;

  if(POT >= POTn)
  {
    ret = BSP_ERROR_WRONG_PARAM;
  }
  else
  {
   if(HAL_ADC_Start(&hpot_adc[POT]) == HAL_OK)
   {
    if(HAL_ADC_PollForConversion(&hpot_adc[POT], POT_ADC_POLL_TIMEOUT) ==
HAL_OK)
    {
      if(HAL_ADC_GetValue(&hpot_adc[POT]) <= (uint32_t)0xFFF)
     {
       ret
=(int32_t)POT_CONVERT2PERC((uint16_t)HAL_ADC_GetValue(&hpot_adc[POT]));
     }
   }
  }
}
return ret;
}
```

With POT_CONVERT2PERC defined BSP driver header file as follows:

```
#define POT_CONVERT2PERC(x) ((((int32_t)x) * 100)/(0xFFF))
```

5.2.17 BSP_POT_DeInit

This method de-initializes the ADC peripheral registers and related resources to their default reset values.

```
int32_t BSP_POT_DeInit(POT_TypeDef POT)
{
  int32_t ret = BSP_ERROR_NONE;

  if(POT >= POTn)
  {
    ret = BSP_ERROR_WRONG_PARAM;
```

```
 }
  else
  {
    /* ADC configuration */
    hpot_adc[POT].Instance = POT1_ADC;

#if (USE_HAL_ADC_REGISTER_CALLBACKS == 0)
    ADCx_MspDeInit(&hpot_adc[POT]);

#endif /* (USE_HAL_ADC_REGISTER_CALLBACKS == 0) */

    if(HAL_ADC_DeInit(&hpot_adc[POT]) != HAL_OK)
    {
      ret = BSP_ERROR_PERIPH_FAILURE;
    }
  }

  return ret;
}
```

Note: 1 All the BSP Potentiometers functions and resources must be delimited by the USE_BSP_POT_FEATURE in the BSP common driver.

Note: 2 By default, the USE_BSP_POT_FEATURE define is deactivated in the common driver header file.

```
#ifndef USE_BSP_POT_FEATURE
  #define USE_BSP_POT_FEATURE   0U
#endif
```

5.2.18 BSP_POT_RegisterDefaultMspCallbacks

This method registers the default POT Msp callbacks.

```
int32_t BSP_POT_RegisterDefaultMspCallbacks (POT_TypeDef POT)
{
  int32_t ret = BSP_ERROR_NONE;

  if(POT >= POTn)
  {
    ret = BSP_ERROR_WRONG_PARAM;
  }
  else
  {
    __HAL_ADC_RESET_HANDLE_STATE(&hpot_adc[POT]);
```

```
    /* Register default MspInit/MspDeInit Callback */
    if(HAL_ADC_RegisterCallback(&hpot_adc[POT], HAL_ADC_MSPINIT_CB_ID,
ADCx_MspInit) != HAL_OK)
    {
      ret = BSP_ERROR_PERIPH_FAILURE;
    }
    else if(HAL_ADC_RegisterCallback(&hpot_adc[POT],
HAL_ADC_MSPDEINIT_CB_ID, ADCx_MspDeInit) != HAL_OK)
    {
      ret = BSP_ERROR_PERIPH_FAILURE;
    }
    else
    {
      IsPotMspCbValid[POT] = 1U;
    }
  }

  /* BSP status */
  return ret;
}
```

5.2.19 BSP_POT_RegisterMspCallbacks

This method registers the specific user POT Msp callbacks.

```
int32_t BSP_POT_RegisterMspCallbacks (POT_TypeDef POT, BSP_POT_Cb_t
*Callback)
{
  int32_t ret = BSP_ERROR_NONE;

  if(POT >= POTn)
  {
    ret = BSP_ERROR_WRONG_PARAM;
  }
  else
  {
    __HAL_ADC_RESET_HANDLE_STATE(&hpot_adc[POT]);

    /* Register MspInit/MspDeInit Callbacks */
    if(HAL_ADC_RegisterCallback(&hpot_adc[POT], HAL_ADC_MSPINIT_CB_ID,
Callback->pMspInitCb) != HAL_OK)
    {
      ret = BSP_ERROR_PERIPH_FAILURE;
    }
```

```
    else if(HAL_ADC_RegisterCallback(&hpot_adc[POT],
HAL_ADC_MSPDEINIT_CB_ID, Callback->pMspDeInitCb) != HAL_OK)
    {
      ret = BSP_ERROR_PERIPH_FAILURE;
    }
    else
    {
      IsPotMspCbValid[POT] = 1U;
    }
  }

  /* BSP status */
  return ret;
}
```

With the `BSP_POT_Cb_t` structure defined as

```
typedef struct
{
  void (* pMspInitCb)(ADC_HandleTypeDef *);
  void (* pMspDeInitCb)(ADC_HandleTypeDef *);
}BSP_POT_Cb_t;
```

Note: The register callbacks statement and functions must be delimited by the USE_HAL_ADC_REGISTER_CALLBACKS conditional defined in the HAL configuration file.

```
#if (USE_HAL_ADC_REGISTER_CALLBACKS == 1)
(…)
#endif /*
```

5.2.20 Using IO expander

On some boards, LED, button or joystick GPIOs may be driven by external IO expander(s) (covered by the BSP IO class driver). The statement based on these GPIOs must be delimited by the use or not of the IO expander and the BSP IO configuration. The GPIOs configuration and use must be implemented as in this LED example:

```
int32_t BSP_LED_Init(Led_TypeDef Led)
{
  int32_t ret = BSP_ERROR_NONE;

#if (USE_BSP_IO_CLASS == 1)
```

```
  BSP_IO_Init_t io_init_structure;
#endif /* (USE_BSP_IO_CLASS == 1) */
  GPIO_InitTypeDef  gpio_init_structure;

  if(Led == LED1)
  {
   /* Enable the GPIO_LED clock */
   LED1_GPIO_CLK_ENABLE();

   /* Configure the GPIO_LED pin */
   gpio_init_structure.Mode = GPIO_MODE_OUTPUT_PP;
   gpio_init_structure.Pull = GPIO_PULLUP;
   gpio_init_structure.Speed = GPIO_SPEED_FREQ_HIGH;
   gpio_init_structure.Pin = LED_PIN [Led];
   HAL_GPIO_Init(LED_PORT[Led], &gpio_init_structure);
   HAL_GPIO_WritePin(LED_PORT [Led], (uint16_t)LED_PIN[Led],
GPIO_PIN_SET);
  }
  else
  {
#if (USE_BSP_IO_CLASS == 1)

   io_init_structure.Pin  = LED_PIN[Led];
   io_init_structure.Pull = IO_PULLUP;
   io_init_structure.Mode = IO_MODE_OUTPUT_PP;
   /* Initialize IO expander */
   if(BSP_IO_Init(0, &io_init_structure) != BSP_ERROR_NONE)
   {
    ret = BSP_ERROR_NO_INIT;
   }
   else
   {
     if(BSP_IO_WritePin(0, LED_PIN[Led], IO_PIN_SET) != BSP_ERROR_NONE)
     {
      ret = BSP_ERROR_NO_INIT;
     }
   }
#endif /* (USE_BSP_IO_CLASS == 1) */
  }

  return ret;
}
```

Note: 1 *`USE_BSP_IO_CLASS` is not defined by default to reduce the driver footprint of the common BSP driver, as the IO expander BSP driver and the used bus (generally I^2C) must be added*

in the project. To enable the IO expander, the `USE_BSP_IO_CLASS` define must be uncommented in the boardname_conf.h file.

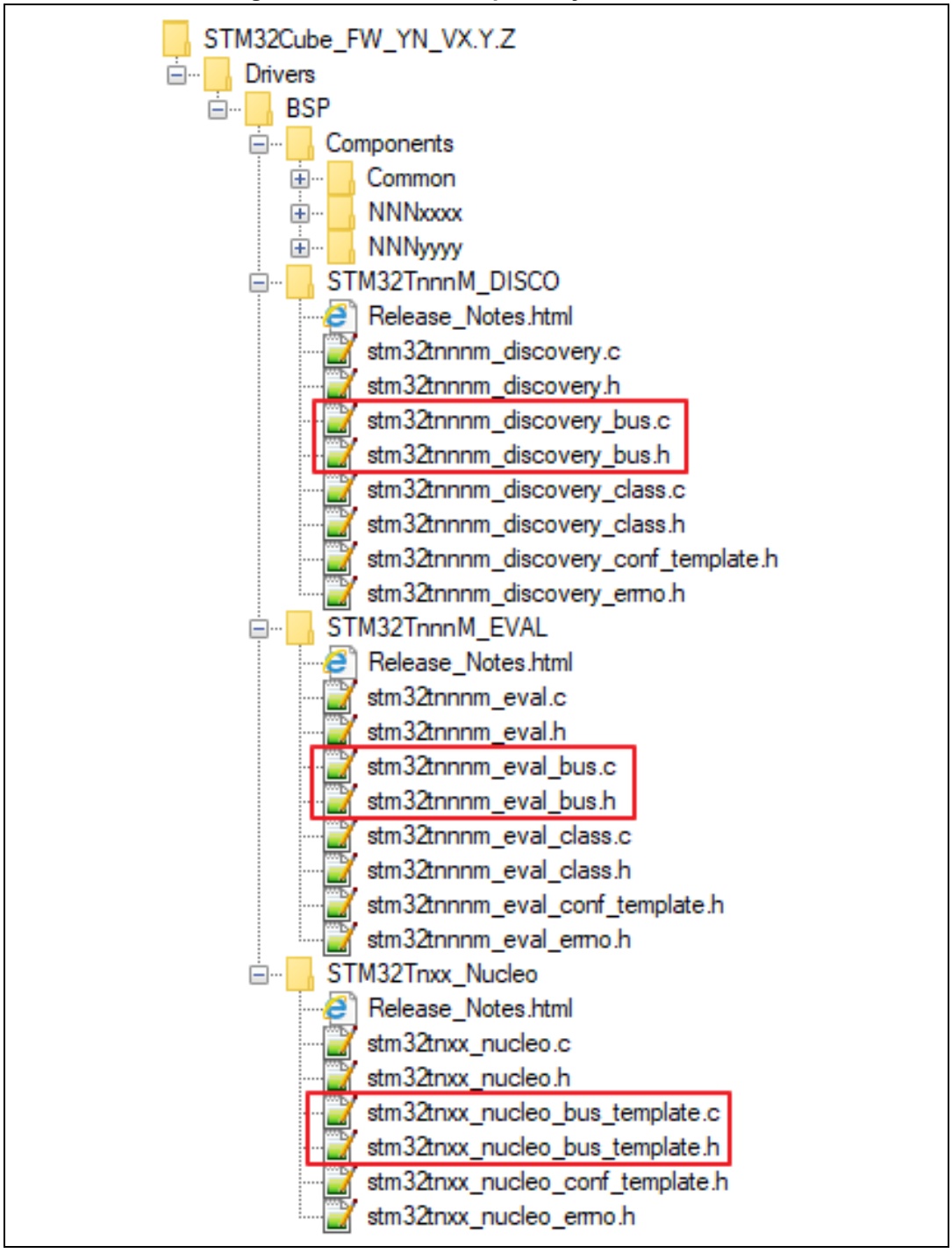
Note: 2 The BSP_IO_Init is built internally to prevent multiple initialization, if the BSP IOE is already initialized, calling the BSP_IO_Init has no effect.

Note: 3 If a BSP common function is entirely using the BSP IOE driver, it must be delimited by the `USE_BSP_IO_CLASS` define.

6 BSP bus driver

The BSP bus driver *boardname_bus.c* exports the transport functions used by the components IO operations. They can be shared by several IO operations of several external components.

Figure 11. BSP driver repository: bus drivers



6.1 BSP bus driver APIs

Each bus available on the board must provide the following APIs:

Table 5. BSP bus driver APIs

Service	Description	Peripheral
`BSP_PPPn_Init`	Initialize a bus	All
`BSP_PPPn_DeInit`	De-Initialize a bus	All
`BSP_PPPn_IsReady`	Return the status of the bus	I^2C
`BSP_PPPn_WriteReg`	Read registers through a bus (8 bits)	I^2C
`BSP_PPPn_ReadReg`	Write registers through bus (8 bits)	I^2C
`BSP_PPPn_WriteReg16`	Read registers through a bus (16 bits)	I^2C
`BSP_PPPn_ReadReg16`	Write registers through bus (16 bits)	I^2C
`BSP_PPPn_Recv`	Receive an amount of data through a bus	All
`BSP_PPPn_Send`	Send an amount width data through bus	All
`BSP_PPPn_SendRecv`	Send and receive an amount of data through bus (Full duplex)	Full duplex like SPI
`BSP_GetTick`	Return system tick in ms	Generic
`BSP_PPPn_RegisterMspCallbacks`	User register bus Msp callback	All
`BSP_PPPn_RegisterDefaultMspCallbacks`	Default register bus Msp callbacks	All

Note: 1 Throughout this section, the address argument is only available for I^2C bus.

Note: 2 In BSP bus drivers, only required services are mandatory, others can be omitted.

6.2 BSP bus APIs implementation

6.2.1 BSP_PPPn_Init

The `BSP_PPPn_Init ()` function initializes the bus MSP and HAL parts. To prevent multiple initializations when several components use the same bus, the initialization function must be protected to ensure that a bus initialization is called once. The protection mechanism must be based on the HAL state of the BUS peripheral.

```
static uint32_t PPPnInitCounter = 0;
(…)

int32_t BSP_PPPn_Init(__ARGS__)
{
  int32_t ret = BSP_ERROR_NONE;

  hbus_pppn.Instance  = BUS_PPPn;

  if(PPPnInitCounter++ == 0U)
  {
    if(HAL_PPP_GetState(&hbus_pppn) == HAL_PPP_STATE_RESET)
    {
#if (USE_HAL_PPP_REGISTER_CALLBACKS == 0)
      /* Init the BUS Msp */
      PPPn_MspInit(&hbus_pppn);
#else
      if(IsPppnMspCbValid == 0U)
      {
        if(BSP_PPPn_RegisterDefaultMspCallbacks() != BSP_ERROR_NONE)
        {
          ret = BSP_ERROR_MSP_FAILURE;
        }
      }
#endif

      if(ret == BSP_ERROR_NONE)
      {
        /* Init the BUS peripheral */
        if (MX_PPPn_Init(&hbus_pppn, __ARGS__)) != HAL_OK)
        {
          ret = BSP_ERROR_BUS_FAILURE;
        }
      }
    }
  }
```

```
  return ret;
}
__weak HAL_StatusTypeDef MX_PPPn_Init (PPP_HandleTypeDef *hppp , __ARGS__)
{
  /* Init the PPP MSP Part*/
  hppp->Init.param1 = PARAM#1;
  (…)
  hppp->Init.ParamN = PARAM#N;
  return HAL_PPP_Init (hppp); }
```

6.2.2 BSP bus MSP default functions

The default MSP functions (Init and DeInit) are defined as follows:

```
static void PPPn_MspInit(PPP_HandleTypeDef *hppp)
{
  GPIO_InitTypeDef gpio_init;

  /* Enable GPIO clock */
  BUS_PPPn_SIGNALA_GPIO_CLK_ENABLE();
  BUS_PPPn_SIGNALB_GPIO_CLK_ENABLE();

  /* Enable Main peripheral clock */
  BUS_PPPn_CLK_ENABLE();

  /* Configure PPP Signal A as alternate function */
  gpio_init.Pin = BUS_PPPn_SIGNALA_GPIO_PIN;
  gpio_init.Mode = GPIO_MODE_AF_PP;
  gpio_init.Speed = GPIO_SPEED_FREQ_HIGH;
  gpio_init.Pull = GPIO_PULLUP;
  gpio_init.Alternate = BUS_PPPn_SIGNALA_GPIO_AF;
  HAL_GPIO_Init(BUS_PPPn_SIGNALA_GPIO_PORT, &gpio_init);

  /* Configure PPPn Signal B as alternate function */
  gpio_init.Pin = BUS_PPPn_SIGNALB_PIN;
  gpio_init.Mode = GPIO_MODE_AF_PP;
  gpio_init.Alternate = BUS_PPPn_SIGNALB_GPIO_AF;
  HAL_GPIO_Init(BUS_PPPn_SIGNALB_GPIO_PORT, &gpio_init);
}

/**
* @brief Initialize PPP Msp part
* @param PPP handle
```

```
* @retval None
*/
static void PPPn_MspDeInit(PPP_HandleTypeDef *hppp)
{
  GPIO_InitTypeDef gpio_init;

  /* COM GPIO pin configuration */
  gpio_init.Pin = BUS_PPPn_SIGNALA_GPIO_PIN;
  HAL_GPIO_DeInit(BUS_PPPn_SIGNALA_GPIO_PORT, gpio_init.Pin);
  gpio_init.Pin = BUS_PPPn_SIGNALB_GPIO_PIN;
  HAL_GPIO_DeInit(BUS_PPPn_SIGNALB_GPIO_PORT, gpio_init.Pin);

  /* Disable PPP clock */
  BUS_PPPn_CLK_DISABLE ();
}
```

The different literals and defines must be defined in the common BSP driver header file as follows:

```
#define BUS_PPPn                                PPPn
#define BUS_PPPn_SIGNALA_GPIO_PORT              GPIOa
#define BUS_PPPn_SIGNALA_GPIO_CLK_ENABLE ()  __HAL_RCC_GPIOa_CLK_ENABLE ()
#define BUS_PPPn_SIGNALA_GPIO_PIN               GPIO_PIN_A
#define BUS_PPPn_SIGNALA_GPIO_AF                GPIO_AFa_PPPx
#define BUS_PPPn_SIGNALB_GPIO_PORT              GPIOb
#define BUS_PPPn_SIGNALB_GPIO_CLK_ENABLE ()  __HAL_RCC_GPIOb_CLK_ENABLE ()
#define BUS_PPPn_SIGNALB_GPIO_PIN               GPIO_PIN_B
#define BUS_PPPn_SIGNALB_GPIO_AF                GPIO_AFb_PPPy
#define BUS_PPPn_CLK_ENABLE ()                  __HAL_RCC_PPPn_CLK_ENABLE ()
#define BUS_PPPn_CLK_DISABLE ()                 __HAL_RCC_PPPn_CLK_DISABLE ()
```

Example:

```
#define BUS_SPI1                          SPI1
#define BUS_SPI1_CLK_ENABLE()             __HAL_RCC_SPI1_CLK_ENABLE()
#define BUS_SPI1_CLK_DISABLE()            __HAL_RCC_SPI1_CLK_DISABLE()
#define BUS_SPI1_SCK_GPIO_PIN             GPIO_PIN_5
#define BUS_SPI1_MISO_GPIO_PIN            GPIO_PIN_6
#define BUS_SPI1_MOSI_GPIO_PIN            GPIO_PIN_7
#define BUS_SPI1_SCK_GPIO_PORT            GPIOA
#define BUS_SPI1_MISO_GPIO_PORT           GPIOA
#define BUS_SPI1_MOSI_GPIO_PORT           GPIOA
#define BUS_SPI1_SCK_GPIO_AF              GPIO_AF5_SPI1
#define BUS_SPI1_MOSI_GPIO_AF             GPIO_AF5_SPI1
#define BUS_SPI1_MISO_GPIO_AF             GPIO_AF5_SPI1
```

```
#define BUS_SPI1_SCK_GPIO_CLK_ENABLE()  __HAL_RCC_GPIOA_CLK_ENABLE()
#define BUS_SPI1_MOSI_GPIO_CLK_ENABLE() __HAL_RCC_GPIOA_CLK_ENABLE()
#define BUS_SPI1_MISO_GPIO_CLK_ENABLE() __HAL_RCC_GPIOA_CLK_ENABL
```

Note: BUS_PPPn_FEATUREx_GPIO_CLK_DISABLE() macros are not defined as their usage is forbidden by the BSP as they can be shared with other firmware modules.

6.2.3 BSP_PPPn_DeInit

The BSP_PPPn_DeInit () function de-initializes the bus MSP and HAL parts.

```
 int32_t BSP_PPPn_DeInit(void)
{
  int32_t ret  = BSP_ERROR_NONE;

  if (PPPnInitCounter > 0U)
  {
    if (--PPPnInitCounter == 0U)
    {
#if (USE_HAL_PPP_REGISTER_CALLBACKS == 0)
      PPPn_MspDeInit(&hbus_pppn);
#endif /* (USE_HAL_PPP_REGISTER_CALLBACKS == 0) */

      /* Init the I2C */
      if (HAL_PPP_DeInit(&hbus_pppn) != HAL_OK)
      {
        ret = BSP_ERROR_BUS_FAILURE;
      }
    }
}
  return ret;
```

6.2.4 BSP_PPPn_ReadReg{16}

The `BSP_BUS_RegisterMspCallbacks` allows assigning the Msp callbacks to the bus handles.

```
int32_t BSP_PPPn_RegisterMspCallbacks(BSP_PPP_Cb_t *Callback)
{
  int32_t ret = BSP_ERROR_NONE;

  __HAL_PPP_RESET_HANDLE_STATE(&hbus_pppn);

  /* Register MspInit/MspDeInit Callbacks */
  if(HAL_PPP_RegisterCallback(&hbus_pppn, HAL_PPP_MSPINIT_CB_ID, Callback-
>pMspBusInitCb) != HAL_OK)
  {
    ret = BSP_ERROR_PERIPH_FAILURE;
  }
  else if(HAL_PPP_RegisterCallback(&hbus_pppn, HAL_PPP_MSPDEINIT_CB_ID,
Callback->pMspBusDeInitCb) != HAL_OK)
  {
    ret = BSP_ERROR_PERIPH_FAILURE;
  }
  else
  {
    IsPppnMspCbValid = 1U;
  }

  /* BSP status */
  return ret;
}
```

6.2.5 BSP_PPPn_RegisterDefaultMspCallbacks

The `BSP_BUS_RegisterDefaultMspCallbacks` allows assigning the default callbacks to the bus handles when the user has changed them.

```
int32_t BSP_PPPn_RegisterDefaultMspCallbacks(void)
{
  int32_t ret = BSP_ERROR_NONE;

  __HAL_PPP_RESET_HANDLE_STATE(&hbus_pppn);

  /* Register default MspInit/MspDeInit Callback */
  if(HAL_PPP_RegisterCallback(&hbus_pppn, HAL_PPP_MSPINIT_CB_ID,
PPPn_MspInit) != HAL_OK)
  {
    ret = BSP_ERROR_PERIPH_FAILURE;
  }
  else if(HAL_PPP_RegisterCallback(&hbus_pppn, HAL_PPP_MSPDEINIT_CB_ID,
PPPn_MspDeInit) != HAL_OK)
  {
    ret = BSP_ERROR_PERIPH_FAILURE;
  }
  else
  {
IsPppnMspCbValid = 1U;
  }

  /* BSP status */
  return ret;
}
```

Note: The register callbacks statement and functions must be delimited by the USE_HAL_PPP_REGISTER_CALLBACKS conditional defined in the HAL configuration file.

```
#if (USE_HAL_PPP_REGISTER_CALLBACKS == 1)
(…)
#endif /*
```

6.2.6 BSP_PPPn_WriteReg{16}

The `BSP_PPPn_WriteReg()` Function permits to Write a register of a specific slave (8/16 bits).

```
int32_t BSP_PPPn_WriteReg(uint16_t Addr, uint16_t Reg, uint8_t *pData,
uint16_t Length)
{
  int32_t ret = BSP_ERROR_NONE;
  uint32_t hal_error = HAL_OK;

  if(HAL_PPP_Transmit(&hbus_pppn, Addr, Reg, pData, Length, PPPn_TIMEOUT)
!= HAL_OK)
  {
    hal_error = HAL_PPP_GetError(&hbus_pppn);
    if( hal_error == HAL_PPP_ERROR_CODE#N )
    {
      ret = BSP_ERROR_BUS_ERROR_CODE#N;
    }
    if( hal_error == HAL_PPP_ERROR_CODE#M )
    {
      ret = BSP_ERROR_BUS_ERROR_CODE#M;
    }
    else
    {
      ret =  BSP_ERROR_PERIPH_FAILURE;
    }
  }

  return ret;
}
```

6.2.7 BSP_PPPn_ReadReg{16}

The `BSP_PPPn_ReadReg` functions permit to read a register of a specific slave (8/16 bits).

```
int32_t BSP_PPPn_ReadReg(uint16_t Addr, uint16_t Reg, uint8_t *pData,
uint16_t Length)
{
  int32_t ret = BSP_ERROR_NONE;
  uint32_t hal_error = HAL_OK;

  if(HAL_PPP_Receive(&hbus_pppn, Addr, Reg, pData, Length, PPPn_TIMEOUT)
!= HAL_OK)
  {
    hal_error = HAL_PPP_GetError(&hbus_pppn);
    if( hal_error == HAL_PPP_ERROR_CODE#N )
{
      ret = BSP_ERROR_BUS_ERROR_CODE#N;
    }
    if( hal_error == HAL_PPP_ERROR_CODE#M )
    {
      ret = BSP_ERROR_BUS_ERROR_CODE#M;
    }
    else
    {
      ret =  BSP_ERROR_PERIPH_FAILURE;
    }
  }

  return ret;
}
```

6.2.8 BSP_PPPn_Send

The `BSP_PPPn_Send` function permits to transmit a specific amount of data to a specific slave.

```
PPP_HandleTypeDef hbus ppp;
int32_t BSP_PPPn_Send (uint8_t *pdata, uint16_t Length)
{
  int32_t ret = BSP_ERROR_BUS_FAILURE;
  if (HAL_PPP_Transmit (&hbus_pppn, pData, Length, PPPn_TIMEOUT) ==
HAL_OK)
  {
    ret = Length;
  }
  return ret;
}
```

6.2.9 BSP_PPPn_Recv

The `BSP_PPPn_Recv` function permits to receive a specific amount of data from a slave.

```
PPP_HandleTypeDef hbus_pppn;
int32_t BSP_PPPn_Recv (uint8_t *pdata, uint16_t Length)
{
  int32_t ret = BSP_ERROR_BUS_FAILURE;
  if (HAL_PPP_Receive (&hbus_pppn, pData, Length, PPPn_TIMEOUT) == HAL_OK)
  {
    ret = BSP_ERROR_NONE;
  }
  return ret;
}
```

6.2.10 BSP_PPPn_SendRecv

The `BSP_PPPn_SendRecv` function permits to send and receive simultaneously a specific amount of data from and to a slave.

```
PPP_HandleTypeDef hbus_pppn; int32_t BSP_PPPn_SendRecv (uint8_t *pTxdata,
uint8_t *pRxdata, uint16_t Length)
{
  int32_t ret = BSP_ERROR_BUS_FAILURE;
  if (HAL_PPP_SendReceive (&hbus_pppn, pTxData, pRxData, Length,
PPPn_TIMEOUT) == HAL_OK)
  {
   ret =  BSP_ERROR_NONE;
  }
 return ret;
}
```

6.2.11 BSP_PPPn_IsReady

```
int32_t BSP_PPPn_IsReady(uint16_t DevAddr, uint32_t Trials)
{
  int32_t ret = BSP_ERROR_NONE;

  if(HAL_PPP_IsDeviceReady(&hbus_pppn, DevAddr, Trials,
BUS_PPPn_POLL_TIMEOUT) != HAL_OK)
  {
    ret = BSP_ERROR_BUSY;
  }

  return ret;
}
```

6.3 BSP bus services

The following table summarizes the most common communication services used by Evaluation, Discovery and Nucleo boards.

Table 6. Most common communication services

I²C	SPI	UART

`HAL_StatusTypeDef MX_I2Cn_Init(I2C_HandleTypeDef *phi2c, uint32_t timing);`
Or for old I²C peripheral version (V1)
`HAL_StatusTypeDef MX_I2Cn_Init(I2C_HandleTypeDef *phi2c, uint32_t Frequency);`
	`HAL_StatusTypeDef MX_SPIn_Init (SPI_HandleTypeDef *phspi, uint32_t baudrate_presc);`	`__weak HAL_StatusTypeDef MX_UARTn_Init(UART_HandleTypeDef *huart, MX_UART_InitTypeDef *UART_Init);`
		Note: the MX_UART_InitTypeDef is inherited from the common driver (see note below)

`int32_t BSP_I2Cn_Init(void);`	`int32_t BSP_SPIn_Init(void);`	`int32_t BSP_UARTn_Init (BUS_UART_InitTypeDef *Init);`
`BSP_I2Cn_DeInit(void);`	`int32_t BSP_SPIn_DeInit(void);`	`int32_t BSP_UARTn_DeInit(void);`
`int32_t BSP_I2Cn_IsReady(uint16_t DevAddr, uint32_t Trials);`	-	-
`int32_t BSP_I2Cn_WriteReg(uint16_t DevAddr, uint16_t Reg, uint8_t *pData, uint16_t Length);`	-	-
`int32_t BSP_I2Cn_ReadReg(uint16_t DevAddr, uint16_t Reg, uint8_t *pData, uint16_t Length);`	-	-
`int32_t BSP_I2Cn_WriteReg16(uint16_t DevAddr, uint16_t Reg, uint8_t *pData, uint16_t Length);`	-	-
`int32_t BSP_I2Cn_ReadReg16(uint16_t DevAddr, uint16_t Reg, uint8_t *pData, uint16_t Length);`	-	-
`int32_t BSP_I2Cn_Send(uint16_t DevAddr, uint8_t *pData, uint16_t Length);`	`int32_t BSP_SPIn_Send(uint8_t *pData, uint16_t Length);`	`int32_t BSP_UARTn_Send(uint8_t *pData, uint16_t Length);`
`int32_t BSP_I2Cn_Recv(uint16_t DevAddr, uint8_t *pData, uint16_t Length);`	`int32_t BSP_SPIn_Recv(uint8_t *pData, uint16_t Length);`	`int32_t BSP_UARTn_Recv(uint8_t *pData, uint16_t Length);`
-	`int32_t BSP_SPIn_SendRecv(uint8_t *pTxData, uint8_t *pRxData, uint16_t Length);`	-

Table 6. Most common communication services (continued)

I²C	SPI	UART
int32_t BSP_I2Cn_RegisterDefaultMspCallbacks (void);	int32_t BSP_SPIn_RegisterDefaultMspCallbacks (void);	int32_t BSP_UARTn_RegisterDefaultMspCallbacks (void);
int32_t BSP_I2Cn_RegisterMspCallbacks (BSP_I2C_Cb_t *Callbacks);	int32_t BSP_SPIn_RegisterMspCallbacks (BSP_SPI_Cb_t *Callbacks);	int32_t BSP_UARTn_RegisterMspCallbacks (BSP_UART_Cb_t *Callbacks);

Note: If the UART is used, the bus driver header file must add the following redefinition to inherit the different UART structures already defined in the common.

```
#include "stm32Tnxx_nucleo.h"
(…)
#define BUS_UART_InitTypeDef MX_UART_InitTypeDef
```

6.4 BSP SPI bus specific services

The sequence of the SPI initialization is provided as weak function that may be overridden by another configuration that can be generated by CubeMX.

The typical SPI initialization sequence is as follows:

```
phspi->Init.Mode = SPI_MODE_MASTER;
phspi->Init.Direction = SPI_DIRECTION_2LINES;
phspi->Init.DataSize = SPI_DATASIZE_8BIT;
phspi->Init.CLKPolarity = SPI_POLARITY_LOW;
phspi->Init.CLKPhase = SPI_PHASE_1EDGE;
phspi->Init.NSS = SPI_NSS_SOFT;
phspi->Init.BaudRatePrescaler = baudrate_presc;
phspi->Init.FirstBit = SPI_FIRSTBIT_MSB;
phspi->Init.TIMode = SPI_TIMODE_DISABLE;
phspi->Init.CRCCalculation = SPI_CRCCALCULATION_DISABLE;
phspi->Init.CRCPolynomial = 7;
if (HAL_SPI_Init(phspi)!= HAL_OK)
{
ret = HAL_ERROR;
}
```

The SPI baud-rate need to be isolated by the current system clock, a static service is added locally in the BSP BUS driver in order to calculate the prescaler from the wished independent baud rate expressed in Hz. This function is defined as follows:

```
static uint32_t SPI_GetPrescaler(uint32_t clock_src_hz, uint32_t
baudrate_mbps)
{
  uint32_t divisor = 0;
  uint32_t spi_clk = clock_src_hz;
  uint32_t presc = 0;
  static const uint32_t baudrate[]=
  {
SPI_BAUDRATEPRESCALER_2,
    SPI_BAUDRATEPRESCALER_4,
    SPI_BAUDRATEPRESCALER_8,
    SPI_BAUDRATEPRESCALER_16,
    SPI_BAUDRATEPRESCALER_32,
    SPI_BAUDRATEPRESCALER_64,
    SPI_BAUDRATEPRESCALER_128,
    SPI_BAUDRATEPRESCALER_256,
 };
  while (spi_clk > baudrate_mbps)
  {
    presc = baudrate[divisor];
    if (++divisor > 7)
      break;
    spi_clk= (spi_clk >> 1);
 }
  return presc;
}
```

The MX_SPIn_Init () uses this function to configure the prescaler using only the baud rate as follows:

```
MX_SPIn_Init (&hbus_spin, SPI_GetPrescaler (HAL_RCC_GetPCLK2Freq(),
BUS_SPIn_BAUDRATE))
```

While the `MX_SPIn_Init` is defined as follows:

```
__weak HAL_StatusTypeDef MX_SPIn_Init(SPI_HandleTypeDef* phspi, uint32_t
baudrate_presc)
{
  HAL_StatusTypeDef ret = HAL_OK;
  phspi->Init.Mode = SPI_MODE_MASTER;
  phspi->Init.Direction = SPI_DIRECTION_2LINES;
  phspi->Init.DataSize = SPI_DATASIZE_8BIT;
  phspi->Init.CLKPolarity = SPI_POLARITY_LOW;
  phspi->Init.CLKPhase = SPI_PHASE_1EDGE;
  phspi->Init.NSS = SPI_NSS_SOFT;
  phspi->Init.BaudRatePrescaler = baudrate_presc;
  phspi->Init.FirstBit = SPI_FIRSTBIT_MSB;
  phspi->Init.TIMode = SPI_TIMODE_DISABLE;
  phspi->Init.CRCCalculation = SPI_CRCCALCULATION_DISABLE;
  phspi->Init.CRCPolynomial = 7;
  if (HAL_SPI_Init(phspi)!= HAL_OK)
  {
    ret = HAL_ERROR;
 }
  return ret;
}
```

Note that the baud rate is defined in the BSP bus driver header file as follows:

```
#ifndef BUS_SPIn_BAUDRATE
#define BUS_SPIn_BAUDRATE 10000000U /* baud rate of SPIn = 10 Mbps*/
#endif
```

.The user can always override the default SPIn baud rate value in the common BSP configuration file.

```
#define BUS_SPIn_BAUDRATE 16000000U /* baud rate of SPIn = 16 Mbps */
```

6.5 BSP I²C bus specific services

The sequence of the I²C initialization is provided as weak function that may be overridden by another configuration that can be generated by CubeMX.

The I²C initialization sequence depends on the I²C peripheral version, it can be as follows for I²C V1 (STM32F1xx, STM32F2xx, STM32F4xx and STM32L1xx):

```
hi2c.Init.ClockSpeed = #I2Cn_FREQUENCY#;
hi2c.Init.DutyCycle = I2C_DUTYCYCLE_2;
hi2c.Init.OwnAddress1 = 0;
hi2c.Init.AddressingMode = I2C_ADDRESSINGMODE_7BIT;
hi2c.Init.DualAddressMode = I2C_DUALADDRESS_DISABLED;
hi2c.Init.OwnAddress2 = 0;
hi2c.Init.GeneralCallMode = I2C_GENERALCALL_DISABLED;
hi2c.Init.NoStretchMode = I2C_NOSTRETCH_DISABLED;
if (HAL_I2C_Init(hi2c)!= HAL_OK)
{
ret = HAL_ERROR;
}
```

And for the I²C V2 (TM32F0xx, STM32F3xx, STM32F7xx, STM32L0xx and STM32L4xx) and I²C V3 (STM32H7x):

```
hi2c->Init.Timing = #I2C_TIMING#;
hi2c->Init.OwnAddress1 = 0;
hi2c->Init.AddressingMode = I2C_ADDRESSINGMODE_7BIT;
hi2c->Init.DualAddressMode = I2C_DUALADDRESS_DISABLE;
hi2c->Init.OwnAddress2 = 0;
hi2c->Init.OwnAddress2Masks = I2C_OA2_NOMASK;
hi2c->Init.GeneralCallMode = I2C_GENERALCALL_DISABLE;
hi2c->Init.NoStretchMode = I2C_NOSTRETCH_DISABLE;
if (HAL_I2C_Init(hi2c)!= HAL_OK)
{
ret = HAL_ERROR;
}
else if (HAL_I2CEx_ConfigAnalogFilter(hi2c, I2C_ANALOGFILTER_DISABLE)!=
HAL_OK)
{
ret = HAL_ERROR;
}
else if (HAL_I2CEx_ConfigDigitalFilter(hi2c, 0x02)!= HAL_OK)
{
ret = HAL_ERROR;
}
```

In both I²C peripheral versions, the frequency must be defined in the BSP bus driver header file as follows:

```
#ifndef BUS_I2Cn_ FREQUENCY
#define BUS_I2Cn_FREQUENCY 100000U /* Frequency of I2Cn = 100 kHz*/
#endif
```

The user can always override the default I2Cn baud rate value in the common BSP configuration file.

```
#define BUS_I2Cn_FREQUENCY 400000U /* Frequency of I2Cn = 400 kHz*/
```

For I²C peripheral version I²C v2 and I²C v3, the I²C frequency must be calculated using the system clock and the I²C control register based on the I²C timing. The BSP specification recommends insulating the bus frequency from the system configuration. Thus, a static service is added locally in the BSP BUS driver in order to calculate the timing field from I²C frequency expressed in Hz. This function is defined as follows:

```
#ifndef I2C_VALID_TIMING_NBR
  #define I2C_VALID_TIMING_NBR                  128U
#endif
#define I2C_SPEED_FREQ_STANDARD                 0U    /* 100 kHz */
#define I2C_SPEED_FREQ_FAST                     1U    /* 400 kHz */
#define I2C_SPEED_FREQ_FAST_PLUS                2U    /* 1 MHz */
#define I2C_ANALOG_FILTER_DELAY_MIN             50U   /* ns */
#define I2C_ANALOG_FILTER_DELAY_MAX             260U  /* ns */
#define I2C_USE_ANALOG_FILTER                   1U
#define I2C_DIGITAL_FILTER_COEF                 0U
#define I2C_PRESC_MAX                           16U
#define I2C_SCLDEL_MAX                          16U
#define I2C_SDADEL_MAX                          16U
#define I2C_SCLH_MAX                            256U
#define I2C_SCLL_MAX                            256U
#define SEC2NSEC                                1000000000UL

typedef struct
{
  uint32_t freq;        /* Frequency in Hz */
  uint32_t freq_min;    /* Minimum frequency in Hz */
  uint32_t freq_max;    /* Maximum frequency in Hz */
  uint32_t hddat_min;   /* Minimum data hold time in ns */
  uint32_t vddat_max;   /* Maximum data valid time in ns */
```

```
  uint32_t sudat_min; /* Minimum data setup time in ns */
  uint32_t lscl_min;   /* Minimum low period of the SCL clock in ns */
  uint32_t hscl_min;   /* Minimum high period of SCL clock in ns */
  uint32_t trise;      /* Rise time in ns */
  uint32_t tfall;      /* Fall time in ns */
  uint32_t dnf;        /* Digital noise filter coefficient */
} I2C_Charac_t;

typedef struct
{
  uint32_t presc;      /* Timing prescaler */
  uint32_t tscldel;    /* SCL delay */
  uint32_t tsdadel;    /* SDA delay */
  uint32_t sclh;       /* SCL high period */
  uint32_t scll;       /* SCL low period */
} I2C_Timings_t;

static const I2C_Charac_t I2C_Charac[] =
{
  [I2C_SPEED_FREQ_STANDARD] =
  {
    .freq = 100000,
    .freq_min = 80000,
    .freq_max = 120000,
    .hddat_min = 0, .vddat_max = 3450,
    .sudat_min = 250,
    .lscl_min = 4700,
    .hscl_min = 4000,
    .trise = 640,
    .tfall = 20,
    .dnf = I2C_DIGITAL_FILTER_COEF,
  },
  [I2C_SPEED_FREQ_FAST] =
  {
    .freq = 400000,
    .freq_min = 320000,
    .freq_max = 480000,
    .hddat_min = 0,
    .vddat_max = 900,
    .sudat_min = 100,
    .lscl_min = 1300,
    .hscl_min = 600,
    .trise = 250,
    .tfall = 100,
```

```
  .dnf = I2C_DIGITAL_FILTER_COEF,
   },
   [I2C_SPEED_FREQ_FAST_PLUS] =
   {
     .freq = 1000000,
     .freq_min = 800000,
     .freq_max = 1200000,
     .hddat_min = 0,
     .vddat_max = 450,
     .sudat_min = 50,
     .lscl_min = 500,
     .hscl_min = 260,
     .trise = 60,
     .tfall = 100,
     .dnf = I2C_DIGITAL_FILTER_COEF,
   },
};

static I2C_Timings_t I2c_valid_timing[I2C_VALID_TIMING_NBR];
static uint32_t      I2c_valid_timing_nbr = 0;
```

and:

```
/**
  * @brief  Compute I2C timing according current I2C clock source and
required I2C clock.
  * @param  clock_src_freq I2C clock source in Hz.
  * @param  i2c_freq Required I2C clock in Hz.
  * @retval I2C timing or 0 in case of error.
  */
static uint32_t I2C_GetTiming(uint32_t clock_src_freq, uint32_t i2c_freq)
{
  uint32_t ret = 0;
  uint32_t speed;
  uint32_t idx;

  if((clock_src_freq != 0U) && (i2c_freq != 0U))
  {
    for ( speed = 0 ; speed <=  (uint32_t)I2C_SPEED_FREQ_FAST_PLUS ;
speed++)
    { if ((i2c_freq >= I2C_Charac[speed].freq_min) &&
          (i2c_freq <= I2C_Charac[speed].freq_max))
      {
        I2C_Compute_PRESC_SCLDEL_SDADEL(clock_src_freq, speed);
        idx = I2C_Compute_SCLL_SCLH(clock_src_freq, speed);

        if (idx < I2C_VALID_TIMING_NBR)
        {
          ret = ((I2c_valid_timing[idx].presc  & 0x0FU) << 28) |\
                ((I2c_valid_timing[idx].tscldel & 0x0FU) << 20) |\
                ((I2c_valid_timing[idx].tsdadel & 0x0FU) << 16) |\
                ((I2c_valid_timing[idx].sclh & 0xFFU) << 8) |\
                ((I2c_valid_timing[idx].scll & 0xFFU) << 0);
        }
        break;
      }
    }
  }

  return ret;
}
```

```
/**
  * @brief  Compute PRESC, SCLDEL and SDADEL.
  * @param  clock_src_freq I2C source clock in HZ.
  * @param  I2C_speed I2C frequency (index).
  * @retval None.
  */
static void I2C_Compute_PRESC_SCLDEL_SDADEL(uint32_t clock_src_freq,
uint32_t I2C_speed)
{
  uint32_t prev_presc = I2C_PRESC_MAX;
  uint32_t ti2cclk;
  int32_t  tsdadel_min, tsdadel_max;
  int32_t  tscldel_min;
  uint32_t presc, scldel, sdadel;
  uint32_t tafdel_min, tafdel_max;

  ti2cclk   = (SEC2NSEC + (clock_src_freq / 2U))/ clock_src_freq;

  tafdel_min = (I2C_USE_ANALOG_FILTER == 1U) ? I2C_ANALOG_FILTER_DELAY_MIN
: 0U;
  tafdel_max = (I2C_USE_ANALOG_FILTER == 1U) ? I2C_ANALOG_FILTER_DELAY_MAX
: 0U;

  /* tDNF = DNF x tI2CCLK
     tPRESC = (PRESC+1) x tI2CCLK
     SDADEL >= {tf +tHD;DAT(min) - tAF(min) - tDNF - [3 x tI2CCLK]} /
{tPRESC}
     SDADEL <= {tVD;DAT(max) - tr - tAF(max) - tDNF- [4 x tI2CCLK]} /
{tPRESC} */

  tsdadel_min = (int32_t)I2C_Charac[I2C_speed].tfall +
(int32_t)I2C_Charac[I2C_speed].hddat_min -
    (int32_t)tafdel_min - (int32_t)(((int32_t)I2C_Charac[I2C_speed].dnf +
3) * (int32_t)ti2cclk);

  tsdadel_max = (int32_t)I2C_Charac[I2C_speed].vddat_max -
(int32_t)I2C_Charac[I2C_speed].trise -
    (int32_t)tafdel_max - (int32_t)(((int32_t)I2C_Charac[I2C_speed].dnf +
4) * (int32_t)ti2cclk);


  /* {[tr+ tSU;DAT(min)] / [tPRESC]} - 1 <= SCLDEL */
  tscldel_min = (int32_t)I2C_Charac[I2C_speed].trise +
(int32_t)I2C_Charac[I2C_speed].sudat_min;

  if (tsdadel_min <= 0)
```

```
 {
    tsdadel_min = 0;
}
if (tsdadel_max <= 0)
  {
    tsdadel_max = 0;
  }

  for (presc = 0; presc < I2C_PRESC_MAX; presc++)
  {
    for (scldel = 0; scldel < I2C_SCLDEL_MAX; scldel++)
    {
      /* TSCLDEL = (SCLDEL+1) * (PRESC+1) * TI2CCLK */
      uint32_t tscldel = (scldel + 1U) * (presc + 1U) * ti2cclk;

      if (tscldel >= (uint32_t)tscldel_min)
      {
        for (sdadel = 0; sdadel < I2C_SDADEL_MAX; sdadel++)
        {
          /* TSDADEL = SDADEL * (PRESC+1) * TI2CCLK */
          uint32_t tsdadel = (sdadel * (presc + 1U)) * ti2cclk;

          if ((tsdadel >= (uint32_t)tsdadel_min) && (tsdadel <=
(uint32_t)tsdadel_max))
          {
            if(presc != prev_presc)
            {
              I2c_valid_timing[I2c_valid_timing_nbr].presc = presc;
              I2c_valid_timing[I2c_valid_timing_nbr].tscldel = scldel;
              I2c_valid_timing[I2c_valid_timing_nbr].tsdadel = sdadel;
              prev_presc = presc;
              I2c_valid_timing_nbr ++;

              if(I2c_valid_timing_nbr >= I2C_VALID_TIMING_NBR)
              {
                return;
              }
            }
          }
        }
      }
    }
  }
}
```

```
/**
  * @brief  Calculate SCLL and SCLH and find best configuration.
  * @param  clock_src_freq I2C source clock in HZ.
  * @param  I2C_speed I2C frequency (index).
  * @retval config index (0 to I2C_VALID_TIMING_NBR], 0xFFFFFFFF for no
valid config.
  */
static uint32_t I2C_Compute_SCLL_SCLH (uint32_t clock_src_freq, uint32_t
I2C_speed)
{
 uint32_t ret = 0xFFFFFFFFU;
  uint32_t ti2cclk;
  uint32_t ti2cspeed;
  uint32_t prev_error;
  uint32_t dnf_delay;
  uint32_t clk_min, clk_max;
  uint32_t scll, sclh;
uint32_t tafdel_min;

  ti2cclk   = (SEC2NSEC + (clock_src_freq / 2U))/ clock_src_freq;
  ti2cspeed   = (SEC2NSEC + (I2C_Charac[I2C_speed].freq / 2U))/
I2C_Charac[I2C_speed].freq;

  tafdel_min = (I2C_USE_ANALOG_FILTER == 1U) ? I2C_ANALOG_FILTER_DELAY_MIN
: 0U;

  /* tDNF = DNF x tI2CCLK */
  dnf_delay = I2C_Charac[I2C_speed].dnf * ti2cclk;

  clk_max = SEC2NSEC / I2C_Charac[I2C_speed].freq_min;
  clk_min = SEC2NSEC / I2C_Charac[I2C_speed].freq_max;

  prev_error = ti2cspeed;

  for (uint32_t count = 0; count < I2c_valid_timing_nbr; count++)
  {
    /* tPRESC = (PRESC+1) x tI2CCLK*/
    uint32_t tpresc = (I2c_valid_timing[count].presc + 1U) * ti2cclk;

    for (scll = 0; scll < I2C_SCLL_MAX; scll++)
    {
      /* tLOW(min) <= tAF(min) + tDNF + 2 x tI2CCLK + [(SCLL+1) x tPRESC ]
*/
      uint32_t tscl_l = tafdel_min + dnf_delay + (2U * ti2cclk) + ((scll +
1U) * tpresc);
```

```
          /* tSCL = tf + tLOW + tr + tHIGH */
          uint32_t tscl = tscl_l + tscl_h + I2C_Charac[I2C_speed].trise +
I2C_Charac[I2C_speed].tfall;

          if ((tscl >= clk_min) && (tscl <= clk_max) && (tscl_h >=
I2C_Charac[I2C_speed].hscl_min) && (ti2cclk < tscl_h))
          {
            int32_t error = (int32_t)tscl - (int32_t)ti2cspeed;

            if (error < 0)
            {
              error = -error;
            }

            /* look for the timings with the lowest clock error */
            if ((uint32_t)error < prev_error)
            {
              prev_error = (uint32_t)error;
              I2c_valid_timing[count].scll = scll;
              I2c_valid_timing[count].sclh = sclh;
              ret = count;
            }
          }
        }
      }
    }
  }

  return ret;
}
```

The `MX_I2Cn_Init()` function configures the timing using only the I²C frequency as follows:

```
MX_I2Cn_Init (&hbus_i2cn, I2C_GetTiming ( SystemCoreClock ,
BUS_I2Cn_FREQUENCY ))
```

While the `MX_I2Cn_Init` is defined as follows:

```
__weak HAL_StatusTypeDef MX_I2Cn_Init (I2C_HandleTypeDef *hi2c, uint32_t
timing)
{
  HAL_StatusTypeDef ret = HAL_OK;
  hi2c->Init.Timing = timing;
  hi2c->Init.OwnAddress1 = 0;
  hi2c->Init.AddressingMode = I2C_ADDRESSINGMODE_7BIT;
  hi2c->Init.DualAddressMode = I2C_DUALADDRESS_DISABLE;
  hi2c->Init.OwnAddress2 = 0;
  hi2c->Init.OwnAddress2Masks = I2C_OA2_NOMASK;
  hi2c->Init.GeneralCallMode = I2C_GENERALCALL_DISABLE;
  hi2c->Init.NoStretchMode = I2C_NOSTRETCH_DISABLE;
  if (HAL_I2C_Init(hi2c) != HAL_OK)
  {
    ret = HAL_ERROR;
  }
else
  {
    uint32_t analog_filter;

    analog_filter = (I2C_USE_ANALOG_FILTER == 1U) ?
I2C_ANALOGFILTER_ENABLE : I2C_ANALOGFILTER_DISABLE;
    if (HAL_I2CEx_ConfigAnalogFilter(hI2c, analog_filter) != HAL_OK)
    {
      ret = HAL_ERROR;
    }
    else
    {
      if (HAL_I2CEx_ConfigDigitalFilter(hI2c, I2C_DIGITAL_FILTER_COEF) !=
HAL_OK)
      {
        ret = HAL_ERROR;
      }
    }
  }  return ret;
}
```

Note: If the user needs to use different I^2C timing configuration or system clock, he can simply override the weak MX_I2Cn_Init () function by fixing the timing parameters in the user files as follows:

```
HAL_StatusTypeDef MX_I2Cn_Init (I2C_HandleTypeDef *hi2c, uint32_t timing)
{
  UNUSED (timing);
  HAL_StatusTypeDef ret = HAL_OK;

  hi2c->Init.Timing = #TIMING#;
  hi2c->Init.OwnAddress1 = 0;
  hi2c->Init.AddressingMode = I2C_ADDRESSINGMODE_7BIT;
  hi2c->Init.DualAddressMode = I2C_DUALADDRESS_DISABLE;
  hi2c->Init.OwnAddress2 = 0;
  hi2c->Init.OwnAddress2Masks = I2C_OA2_NOMASK;
hi2c->Init.GeneralCallMode = I2C_GENERALCALL_DISABLE;
  hi2c->Init.NoStretchMode = I2C_NOSTRETCH_DISABLE;
  if (HAL_I2C_Init(hi2c) != HAL_OK)
  {
    ret = HAL_ERROR;
  }
  else if (HAL_I2CEx_ConfigAnalogFilter(hi2c, I2C_ANALOGFILTER_DISABLE) !=
HAL_OK)
  {
    ret = HAL_ERROR;
  }
  else
  {
    if (HAL_I2CEx_ConfigDigitalFilter(hi2c, 0x02) != HAL_OK)
    {
      ret = HAL_ERROR;
    }
  }

  return ret;
}
```

6.6 BSP bus multi configuration

The bus configuration can be different following the attached components requirements. In this case, different configurations are provided by several elementary bus initialization functions following the required configurations.

```
PPP_HandleTypeDef hbus_pppn;
Int32_t BSP_PPPn_InitA (void)
{
  int32_t ret = BSP_ERROR_BUS_FAILURE;
  hbus_pppn.Instance = BSP_PPPn;

if(PPPnInitCounter++ == 0U)
  {
  If (HAL_PPP_GetState (&hbus_pppn) == HAL_PPP_STATE_RESET)
  {
    if(ClassCtx[Instance].IsMspCallbacksValid == 0)
    {
      if(BSP_CLASS_RegisterDefaultMspCallbacks (Instance)!=
BSP_ERROR_NONE)
      {
        return BSP_ERROR_PERIPH_FAILURE;
      }
    }

    if (MX_PPPx_InitA (&hbus_pppn)! = HAL_OK)
    {
      ret = BSP_ERROR_BUS_FAILURE;
    }
   }
 }
 return ret;
}
__weak HAL_StatusTypeDef MX_PPPx_InitA (PPP_HandleTypeDef *hppp)
{
  hppp->Init.param1 = PARAM#1A;
  (…)
  hppp->Init.ParamN = PARAM#NA;
  return HAL_PPP_Init (hppp);
}

int32_t BSP_PPPn_InitB (void)
{
  int32_t ret = BSP_ERROR_BUS_FAILURE;
  hbus_pppn.Instance = BSP_PPPn;
```

```
if(PPPnInitCounter++ == 0U)
  {
  if (HAL_PPP_GetState (&hbus_pppn) == HAL_PPP_STATE_RESET)
  {
    if(ClassCtx[Instance].IsMspCallbacksValid == 0)
    {
      if(BSP_CLASS_RegisterDefaultMspCallbacks (Instance)!=
BSP_ERROR_NONE)
      {
        return BSP_ERROR_PERIPH_FAILURE;
      }
    }

    if (MX_PPPx_InitB (&hbus_pppn)! = HAL_OK)
    {
      ret = BSP_ERROR_BUS_FAILURE;
    }
  }
 }
  return ret;
}
__weak HAL_StatusTypeDef MX_PPPx_InitB (PPP_HandleTypeDef *hppp)
{
  hppp->Init.param1 = PARAM#1B;
  (…)
  hppp->Init.ParamN = PARAM#NB;
  return HAL_PPP_Init (hppp);
}
```

Note: 1 *The bus multi-configuration must be avoided, when a common configuration can be found without performance loss. Example if two component require two different I^2C frequencies, the lowest frequency can be used if it is accepted by the two components.*

Note: 2 *The different configuration must be used by the different Read/Write functions to select the right configuration for each component.*

6.7 BSP bus customization

The different bus services provided within the BSP drivers are based on polling model, IT and DMA are not supported natively. Moreover, these bus services do not embed internally concurrent access protection mechanism; in case of a busy bus, the BSP bus services return simply the BSP_ERROR_BUSY error that must be managed by the upper layers.

If the bus instance used by a component is not supported by the native mother board or specific bus services are required (interrupt model, DMA) or there is a need to manage concurrent access to bus through mutexes or semaphores, the user must create a bus driver extension file to add the required services in the application folders. Note that for

Nucleo bus driver, no additional bus extension file is required as it is always an application file.

Note: 1 When one of the BSP function needs to be overridden, the native BSP bus driver must be removed from the project setting and the extension file must be used instead without the _ex suffix.

Note: 2 The bus driver extension header file must be included in the board configuration file.

7 BSP component driver

The component driver is the interface for an external device on the board and it is not directly dependent of the HAL. The component driver provides specific unitary services and can be used by any board integrating this external component.

Figure 12. BSP driver repository: component drivers

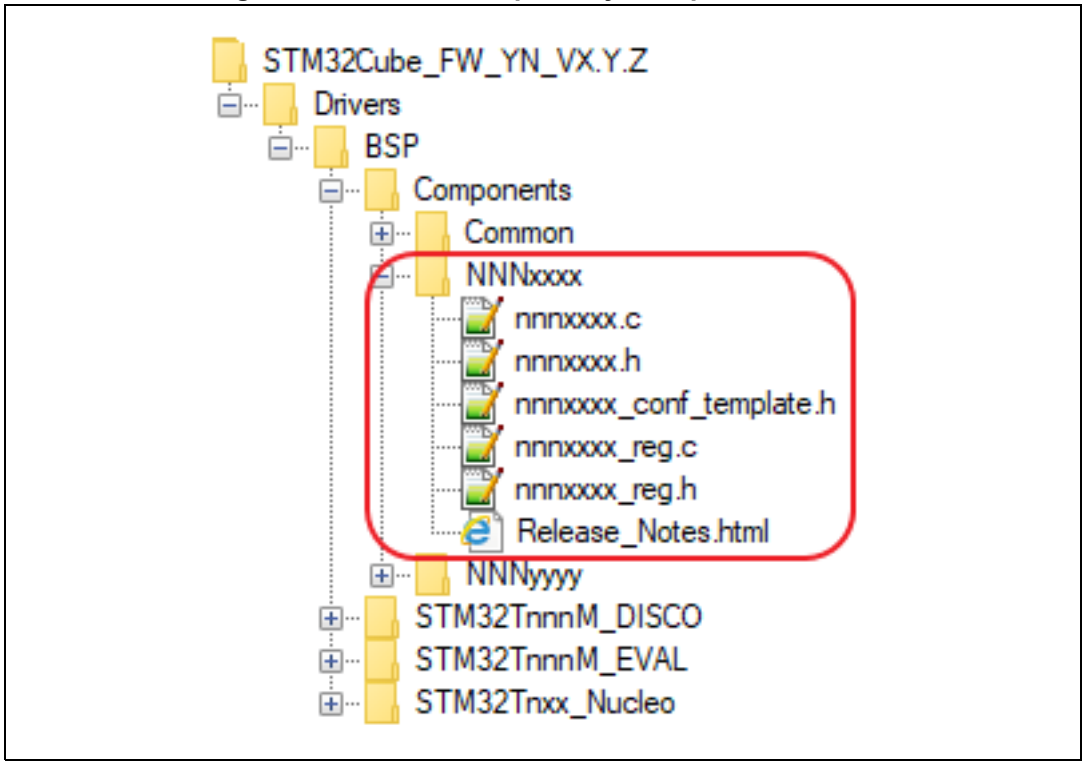
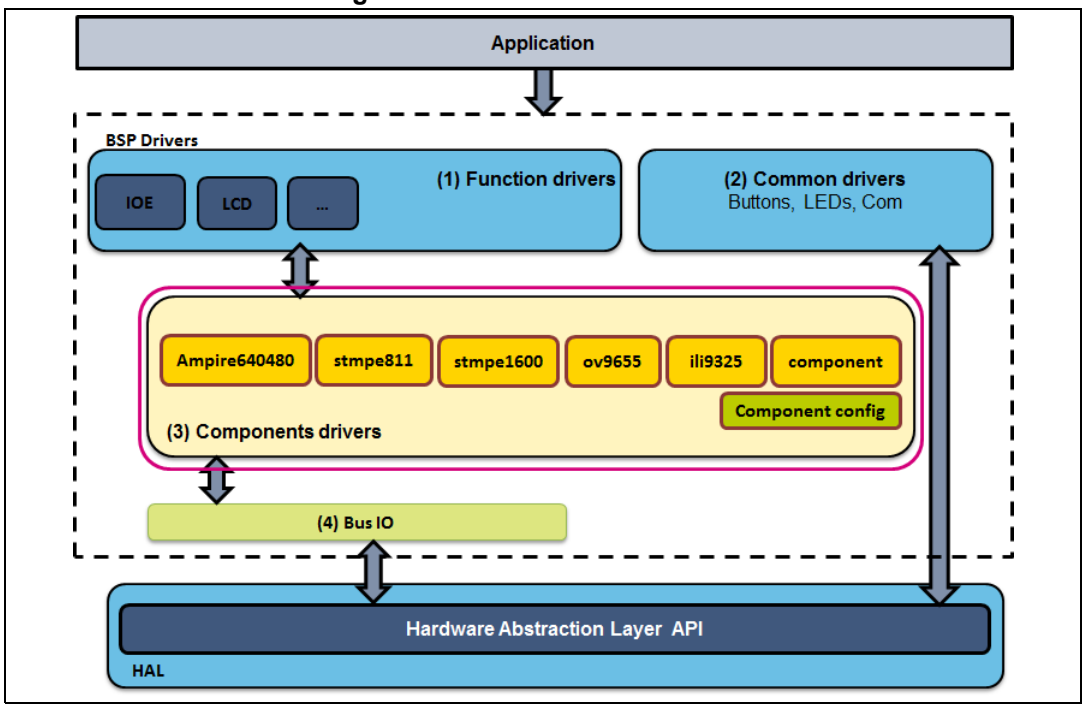


Figure 13. BSP driver architecture



The component drivers, except memory ones, are split into two drivers set:

- Component register drivers (*nnnxxxxx_reg.c/.h*): the register file contains the defined component registers, the low layer register Write and Read operations, the unitary functions and the context structure. This file is completely platform independent and may be used with any environment.
- Component core drivers (*nnnxxxxx.c/.h*): this file contains the component high-level functions based on the unitary register functions.

7.1 BSP component register file

The component register file is named always as follows: *nnnxxxx_reg.c* or *nnnxxxx_reg.h*, where nnxxxx is the component part number. This driver provides the set of unitary/elementary functions to set and get the different component registers.

In addition to the functional set/get elementary functions, the component register file must provide the generic read/write functions and the different component register and bit definition.

7.1.1 Component register header file

The component register header file (*nnnxxxx_reg.h*) looks like:

```
/* Component Registers definition */
#define NNNXXXX_REG_NAME1 0X06
#define NNNXXXX_REG_NAME2 0X07
#define NNNXXXX_REG_NAME3 0X08
#define NNNXXXX_REG_NAME4 0X09
#define NNNXXXX_REG_NAME5 0X0A
(…)

/* Component Context structure */
typedef int32_t (*NNNXXXX_Write_Func)(void *, uint8_t, uint8_t*,
uint16_t);
typedef int32_t (*NNNXXXX_Read_Func) (void *, uint8_t, uint8_t*,
uint16_t);
typedef struct
{
  NNNXXXX_Write_Func WriteReg;
  NNNXXXX_Read_Func  ReadReg;
  void *handle;
} nnnxxxx_ctx_t;

/* Prototypes of Read/Write component functions */
int32_t nnnxxxx_write_reg(nnnxxxx_ctx_t *ctx, uint16_t Reg, uint8_t *Data,
uint16_t len);
int32_t nnnxxxx_read_reg(nnnxxxx_ctx_t *ctx, uint16_t Reg, uint8_t *Data,
uint16_t len);

/* Component Register's bit definition */
#define NNNXXXX_ITEM_BIT_MASK MASK_VALUE
#define NNNXXXX_ITEM_BIT_POSITION POSITION_VALUE
(…)
/* Prototypes of component elementary functions */
int32_t nnnxxxx_set_item(nnnxxxx_ctx_t *ctx, NNNXXXX_FEATURE_t *value);
int32_t nnnxxxx_get_item(nnnxxxx_ctx_t *ctx, NNNXXXX_FEATURE_t *value);
(…)
```

*Note: The context structure is always the first argument of the register driver unitary functions. It is a data object required to link the register and core component drivers smoothly and it is always named nnnxxxx_ctx_t_t * and defined as above.*

7.1.2 Component register source file

In the component register source file (*nnnxxxx_reg.c*), the Read/Write and elementary functions are implemented as follows:

```
int32_t nnnxxxx_write_reg(nnnxxxx_ctx_t *ctx, uint16_t reg, uint8_t
*pdata, uint16_t len)
{
  return ctx->WriteReg(ctx->handle, Reg, pdata, len);
}
int32_t nnnxxxx_read_reg(nnnxxxx_ctx_t *ctx, uint16_t reg, uint8_t* pdata,
uint16_t len)
{
  return ctx->ReadReg(ctx->handle, reg, pdata, len);
}
int32_t nnnxxxx_set_item (nnnxxxx_ctx_t *ctx, uint8_t value)
{
  uint8_t tmp;
  if (nnnxxxx_read_reg(ctx, NNNXXXX_ITEM_REG, &tmp, 1))
    return NNNXXXX_ERROR;
  tmp &= ~ NNNXXXX_ITEM_BIT_MASK;
  tmp |= value << NNNXXXX_ITEM_BIT_POSITION;
  if (nnnxxxx_write_reg(ctx, NNNXXXX_ITEM_REG, &tmp, 1))
    return NNNXXXX_ERROR;
  return NNNXXXX_OK;
}
int32_t nnnxxxx_get_item (nnnxxxx_ctx_t *ctx, uint8_t *value)
{
  if (nnnxxxx_read_reg(ctx, NNNXXXX_ITEM_REG, (uint8_t *)value, 1))
    return NNNXXXX_ERROR;
  *value &= NNNXXXX_ITEM_BIT_MASK;
  *value = *value >> NNNXXXX_ITEM_BIT_POSITION;
  return NNNXXXX_OK;
}
```

Note: The exported types, the functions names and their parameters must be defined in lower case to be compatible with Linux environment.

7.2 BSP component core drivers

The component core drivers provide the high-level functions built on top of the component register drivers, it is the main application, middleware and BSP class interface. The component core drivers provide the main component object structure and the different literals associated with enumeration types, in addition to the high-level component function and the class structure.

7.2.1 Component core header file

The component header file provides the component object handle structure definition, the IO structure, the driver structure and the component functions.

```
#ifndef NNNXXXX_H
#define NNNXXXX_H
#ifdef __cplusplus
extern "C" {
#endif
(…)
typedef struct
{
 (…)
} NNNXXXX_Object_t;

typedef struct
{
 (…)
} NNNXXXX_IO_t;

typedef struct
{
 (…)
} NNNXXXX_Drv_t;

(…)
int32_t (NNNXXXX_Function (NNNXXXX_Object_t * pObj, DRV_ARGS)
(…)
#ifdef __cplusplus
}
#endif
#endif /* NNNXXXX_H */
```

Object structure

The component object structure is used to allow instantiating a component driver to handle several identical components on same hardware platform (board/extension board) and also as a global storage for internal parameters exchanged between the driver modules and functions.

The structure type definition is declared. A typical component object structure is defined as follows:

```
typedef struct
{
  NNNXXXX_IO_t IO;
  nnnxxxx_ctx_t Ctx;
  uint32_t IsInitialized;
  uint32_t Params#1;
  (…)
  uint32_t Params#N;
} NNNXXXX_Object_t;
```

Refer also to *Table 7*:

Table 7. BSP component drivers: object structure

Object field	Description
IO	IO bus structure
Ctx	The register driver context structure
IsInitialized	Set to 1 if the component has already initialized (Must be set during the first component Init call)
Params#N	Common class component private parameters

IO structure

The IO structure provides the main IO mechanism to link a component to the board that this later is attached to. The IO functions depend on each component and are defined as follows:

```
typedef int32_t (*NNNXXXX_Init_Func)    (void);
typedef int32_t (*NNNXXXX_DeInit_Func)  (void);
typedef int32_t (*NNNXXXX_GetTick_Func) (void);
typedef int32_t (*NNNXXXX_WriteReg_Func)(uint16_t, uint16_t, uint8_t*,
uint16_t);
typedef int32_t (*NNNXXXX_ReadReg_Func) (uint16_t, uint16_t, uint8_t*,
uint16_t);
typedef int32_t (*NNNXXXX_Recv_Func)    (uint8_t*, uint16_t);
typedef int32_t (*NNNXXXX_Send_Func)    (uint8_t*, uint16_t);
typedef int32_t (*NNNXXXX_SendRecv_Func)(uint8_t*, uint8_t*, uint16_t);
typedef struct
{
NNNXXXX_Init_Func        Init;
NNNXXXX_DeInit_Func      DeInit;
uint16_t                 Address;
NNNXXXX_WriteReg_Func    WriteReg;
NNNXXXX_ReadReg_Func     ReadReg;
NNNXXXX_Send_Func        Write;
NNNXXXX_Recv_Func        Read;
NNNXXXX_SendRecv_Func    SendRecv;
NNNXXXX_GetTick_Func     GetTick;
}NNNXXXX_IO_t;
```

The members of the NNNXXXX_IO_t structure are defined as described in *Table 8*:

Table 8. BSP component drivers: IO structure(1)

Object field	Description
Address	Device address (may be useless in SPI case)
Init	Initialize the bus
DeInit	De-initialize the bus (without modifying the MSP)
ReadReg	Read specific component registers
WriteReg	Write specific component registers
Recv	Receive multiple data
Send	Send multiple data
SendRecv	Send and receive multiple data simultaneously (Full duplex)
ITConfig	Configure the interruption or EXTI line associated with the component
GetTick	Return time stamp (may be used for delays and timeout computation)

1. The IO operation functions list highlighted in gray in *Table 8* depend on the component requirements.

Here are examples of components IO structure:

1. IO structure for LSM6SDL MEMS component

```
typedef int32_t (*LSM6DSL_Init_Func) (void);
typedef int32_t (*LSM6DSL_DeInit_Func) (void);
typedef int32_t (*LSM6DSL_GetTick_Func) (void);
typedef int32_t (*LSM6DSL_WriteReg_Func)(uint16_t, uint16_t, uint8_t*,
uint16_t);
typedef int32_t (*LSM6DSL_ReadReg_Func) (uint16_t, uint16_t, uint8_t*,
uint16_t);
typedef struct
{
 LSM6DSL_Init_Func Init;
 LSM6DSL_DeInit_Func DeInit;
 uint16_t Address;
 LSM6DSL_WriteReg_Func WriteReg;
 LSM6DSL_ReadReg_Func ReadReg;
 LSM6DSL_GetTick_Func GetTick;
} LSM6DSL_IO_t;
```

2. IO structure for ESP8266 WIFI component

```
typedef int8_t (*ESP8266_Init_Func)(void);
typedef int8_t (*ESP8266_DeInit_Func)(void);
typedef int16_t (*ESP8266_Send_Func)(uint8_t *, uint16_t len, uint32_t);
typedef int16_t (*ESP826 _Receive_Func)(uint8_t *, uint16_t len,
uint32_t);
typedef int32_t (*ESP8266_GetTick_Func)(void);

typedef struct
{
 ESP8266_Init_Func Init;
 ESP8266_DeInit_Func DeInit;
 ESP8266_GetTick_Func GetTick;
 ESP8266_Send_Func Send;
 ESP8266_Receive_Func Receive;
} ESP8266_IO_t;
```

Component driver structure

The component driver structure is an object that holds the most common services and elementary functions offered by the external component for a given class.

A typical component driver structure is defined as follows:

```
Typedef struct
{
 int32_t (* Init)   (NNNXXXX_Object_t *, DRV_ARGS);
 int32_t (* DeInit) (NNNXXXX_Object_t *, DRV_ARGS);
 int32_t (* Funct1) (NNNXXXX_Object_t *, DRV_ARGS);
 (…)
 int32_t (* FunctN) (NNNXXXX_Object_t *, DRV_ARGS);
} NNNXXXX_Drv_t;
```

Component driver APIs

The component drivers, contrary to the class driver structure, must provide all the elementary functions that cover all the functionalities and the features that the component may provide. In addition to the component specific functions:

```
uint32_t NNNXXXX_Init (NNNXXXX_Object_t* Obj, DRV_ARGS);
uint32_t NNNXXXX_DeInit (NNNXXXX_Object_t* Obj, DRV_ARGS);
uint32_t NNNXXXX_Function#x (NNNXXXX_Object_t* Obj, DRV_ARGS);
```

Each component must provide the following functions:

```
uint32_t NNNXXXX_RegisterBusIO (NNNXXXX_Object_t* Obj, NNNXXXX_IO_t *io);
uint32_t NNNXXXX_ReadID (NNNXXXX_Object_t* Obj, uint32_t* ID);
```

Note: The ReadID API may be omitted if the component does not support ID feature.

7.2.2 Component core source file

The component source file provides the component functions' implementation, the IO registration mechanism, the wrapper functions and the linking of core drivers to register ones.

Wrapper functions

The Write and Read APIs in the component register driver are defined by the component architecture itself and do not depend on the core driver context.

These operations are based on a global component context to fit any platform requirements:

```
typedef struct
{
  NNNXXX_Write_Func   WriteReg;
  NNNXXX_Read_Func    ReadReg;
  void                  *handle;
} nnnxxxx_ctx_t;
```

This context must be linked to the component core drivers to fit the board HW constraints.

This is insured during the registration phase by calling NNNXXXX_RegisterBusIO API.

The Read and Write functions in the core drivers are more linked to the bus driver IO operations prototypes and the class hardware configuration.

The core driver must use wrapper functions to adapt the IO operation with those from the component register driver.

A typical implementation of wrapper functions is as follows:

```
static int32_t ReadRegWrap(void *handle, uint8_t Reg, uint8_t* pData,
uint16_t len)
{
 NNNXXXX_Object_t *pObj = (NNNXXXX_Object_t *)handle;
 return pObj->IO.ReadReg(pObj->IO.Address, Reg, pData, len);
}

static int32_t WriteRegWrap(void *handle, uint8_t Reg, uint8_t* pData,
uint16_t len)
{
 NNNXXXX_Object_t *pObj = (NNNXXXX_Object_t *)handle;
 return pObj->IO.WriteReg(pObj->IO.Address, Reg, pData, len);
}
```

Register IO API

The `NNNXXXX_RegisterBusIO` in the component core driver permits to connect a component device to the IO bus defined by the board. It adds a mechanism to link the wrap Write and Read functions and is responsible of initializing the used bus.

Typical `NNNXXXX_RegisterBusIO` implementations is as follows:

```
uint32_t NNNXXXX_RegisterBusIO (NNNXXXX_Object_t *Obj, NNNXXXX_IO_t *io)
{
 if(Obj == NULL)
 {
  return NNNXXXX_ERROR;
 }

 Obj->IO.Address     = io->Address;
 Obj->IO.Init        = io->Init;
 Obj->IO.DeInit      = io->DeInit;
 Obj->IO.WriteReg    = io->WriteReg;        Connecting component to IO bus
 Obj->IO.ReadReg     = io->ReadReg;
 Obj->IO.GetTick     = io->GetTick;

 pObj->Ctx.ReadReg   = ReadRegWrap;
 pObj->Ctx.WriteReg  = WriteRegWrap;         Wrapping mechanism
 pObj->Ctx.handle    = pObj;

 if (pObj->IO.Init)
  return pObj->IO.Init();                    Bus Initialization
 else
  return NNNXXXX_ERROR;
}
```

7.3 Registering components

The IO operations functions are required by the components drivers to have access to transport services of the bus to which they are linked.

Most of the component IO operations are based on the generic bus services exported by the file boardname_bus.h. They can be directly assigned to the component IO structure as shown in the following listing:

```
NNNXXXX_Object_t ComponentObj;
void MyFunction (void)
{
(…)
NNNXXXX_IO_t IO;
/* Assign the component IO operations */
IO.DevAddr = DevAddr;
IO.Init = PPPn_Init;
IO.ReadReg = PPPn_ReadReg;
IO.WriteReg = PPPn_WriteReg;
IO.GetTick = BSP_GetTick;
NNNXXXX_RegisterBusIO (&ComponentObj, &IO);
(…)
NNNXXXX_Init (&ComponentObj, &Init);
}
```

However, some components may need more control signals or need a different IO operations function prototypes, in this case, the component IO operations need to be wrapped to the native BSP bus functions through dedicated functions that are implemented in the class driver or the middleware interface file in case the component services are directly called from middleware.

```
NNNXXXX_Object_t ComponentObj;
static int32_t CLASS_BUS_Init(void)
{
  GPIO_InitTypeDef GPIO_Init;
  int32_t ret = BSP_ERROR_UNKNOWN_FAILURE;

  if (BSP_PPPn_Init() < 0);
  {
    __HAL_RCC_GPIOy_CLK_ENABLE();
    GPIO_Init.Pin = GPIO_PIN_x;
    (…)
      HAL_GPIO_Init (GPIOy, &GPIO_Init);
     ret = BSP_ERROR_NONE;
  }
  return ret;
```

```
}

static int32_t CLASS_BUS_DeInit(void)
{
  int32_t ret = BSP_ERROR_UNKNOWN_FAILURE;
  if(BSP_ PPPn_DeInit() < 0);
  {
    HAL_GPIO_DeInit(GPIOy, GPIO_PIN_x);
    ret = BSP_ERROR_NONE;
  }
  return ret;
}
static int32_t CLASS_BUS_WriteReg(uint16_t Addr, uint16_t Reg, uint8_t
*pdata, uint16_t len )
{
  int32_t ret = BSP_ERROR_UNKNOWN_FAILURE;
  /* Control signal Enable */
  HAL_GPIO_WritePin(GPIOy, GPIO_PIN_x, GPIO_PIN_RESET);
  /* Format Write data */
  (…)
    if(BSP_PPPn_SendRecv(TxBuffer, RxBuffer, (len + 1)) < 0);
   {
     ret = BSP_ERROR_NONE;
   }
   /* Control signal Disable */
   HAL_GPIO_WritePin(GPIOy, GPIO_PIN_x, GPIO_PIN_SET);
   return ret;
}

static int32_t CLASS_BUS_ReadReg(uint16_t Addr, uint16_t Reg, uint8_t
*pdata, uint16_t len )
{
  int32_t ret = BSP_ERROR_UNKNOWN_FAILURE;
  /* Control signal Enable */
  HAL_GPIO_WritePin(GPIOy, GPIO_PIN_x, GPIO_PIN_RESET);
  /* Format Read data */
  (…)
    if(BSP_PPPn_SendRecv(TxBuffer, RxBuffer, (len + 1)) < 0);
   {
     ret = BSP_ERROR_NONE;
   }
   /* Control signal Disable */
   HAL_GPIO_WritePin(GPIOy, GPIO_PIN_x, GPIO_PIN_SET);
   return ret;
}
```

```
void MyFunction (void)
{
  (…)
   NNNXXXX_IO_t IO;
   /* Assign the component IO operations */
   IO.Init = CLASS_BUS_Init;
   IO.ReadReg = CLASS_BUS_ReadReg;
   IO.WriteReg = CLASS_BUS_WriteReg;
   IO.GetTick = BSP_GetTick;
   NNNXXXX_RegisterBusIO (&ComponentObj, &IO);
   (…)
   NNNXXXX_Init (&ComponentObj, &Init);
}
```

Note: When several components share the same buses, it is highly recommended that dedicated buses availability and control mechanisms must be used. For example, in case of I^2C, BSP_PPPn_IsReady function must be used before performing any IO operations. This is already handled in the HAL services internally and if the bus is used, a HAL error is returned. However, in case of LL, the check on the bus state must be added.

7.4 BSP component class drivers

The class header files are located under BSP/Components/Common folder (refer to *Figure 6*. BSP driver repository: common class headers)

The class header file provides the common class services in the class structure and must only include the ‘stdint.h’ file as the component does not depends on any STM32 platforms neither a specific board

```
#ifndef __CLASS_H
#define __CLASS_H
#ifdef __cplusplus
extern "C" {
#endif

#include <stdint.h>

typedef struct
{
 int32_t (*Init)        (void *, __ARGS_);
 int32_t (*DeInit)      (void *, __ARGS_);
 int32_t (*Funct1)      (void *, __ARGS_);
 (…)
 int32_t (*FunctN)      (void *, __ARGS_);
} CLASS_Drv_t;

#ifdef __cplusplus
}
#endif
#endif /* __CLASS_H */
```

7.5 BSP memory component drivers

The memory component driver is the interface for an external device on the board and it is, contrary to standard components, dependent of the HAL/LL. The component driver provides specific unitary services and may be used by any board integrating this external component.

Figure 14. BSP driver repository: memory component drivers

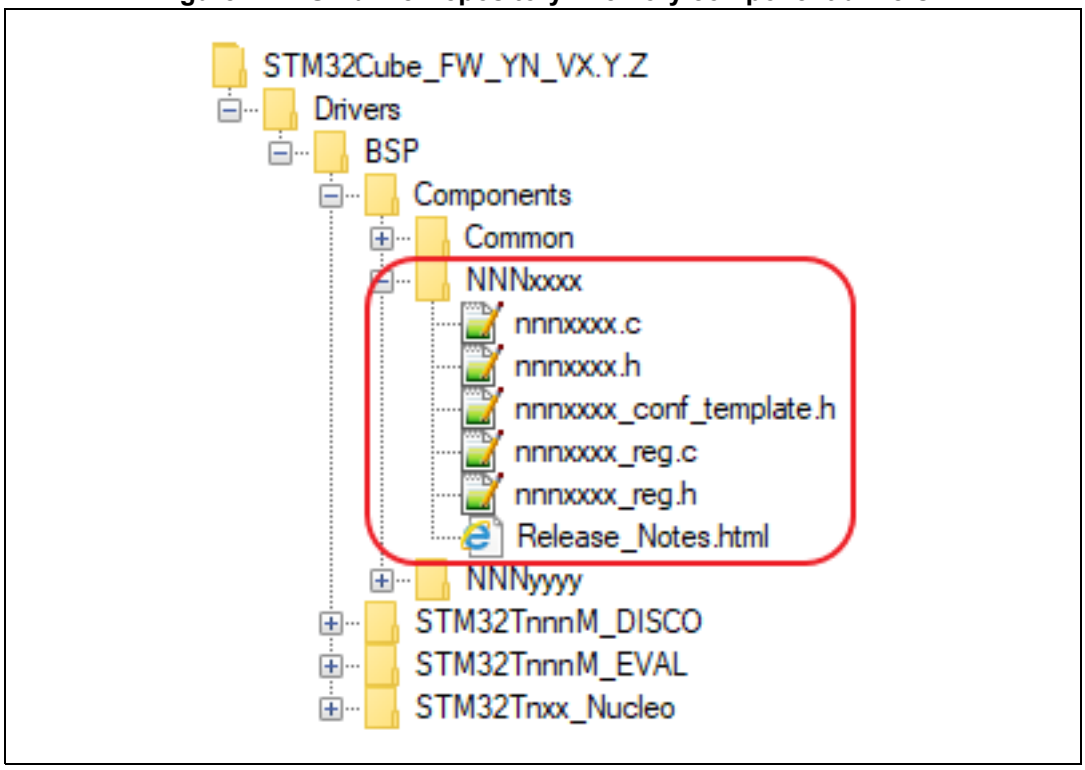
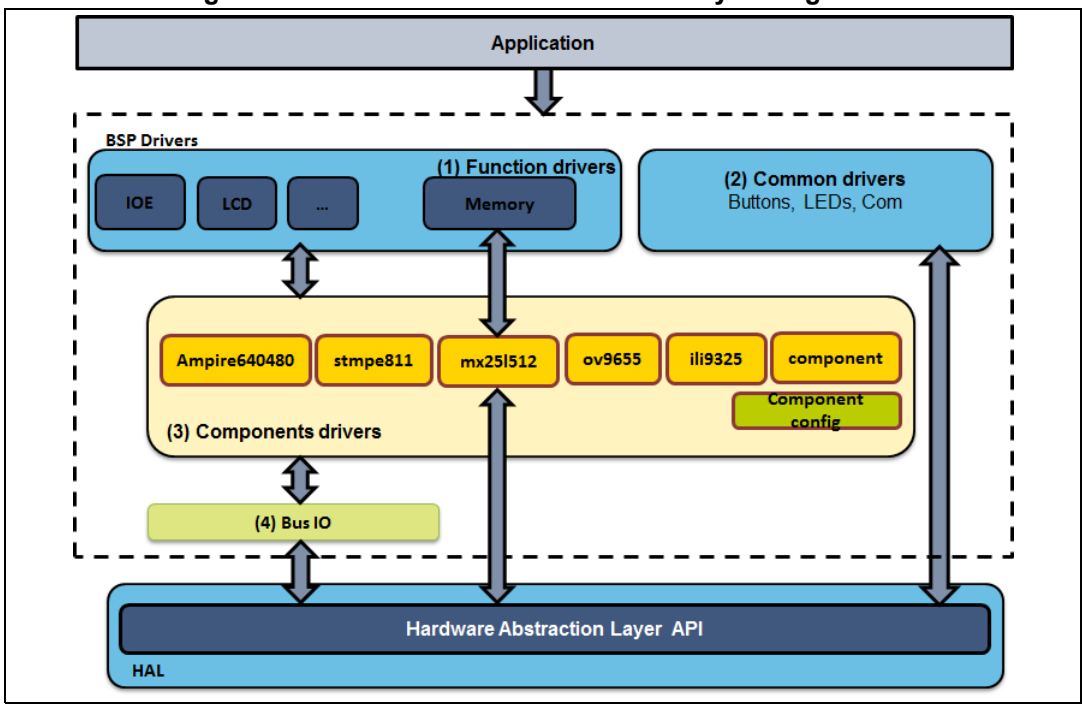


Figure 15. BSP driver architecture: memory calling model



The memory component driver is simply calling to the main STM32 peripheral, which is linked to, and is providing a set of unitary functions corresponding to the different commands processes required by the memory part number.

The memory component driver may call directly the HAL services (only the common services) and the platform must be selected in the component driver configuration file.

From BSP class driver, the services of the memory component are called in the same way as for the BSP Init model 2) single controller component P/N.

The component requires for the HAL the basic HAL services to build the command structure and execute it. The HAL services required from the component are basically to send command, send and receive data and optionally get status or poll for operation completion depending on the data stage required for each command (No Data, Data IN or Data OUT). As the memory component needs only to execute specific commands, no Init function is required, the memory initialization is done through the STM32 memory peripheral.

Typical example for this memory component architecture is the QSPI; examples of commands are as follows:

```
int32_t MX25L512_Read4BytesAddr (void *Ctx, uint8_t* pData, uint32_t
ReadAddr, uint32_t Size)
{
  QSPI_CommandTypeDef cmd;

  cmd.InstructionMode = QSPI_INSTRUCTION_4_LINES;
  cmd.Instruction = MX25L512_QPI_READ_4_BYTE_ADDR_CMD;
  cmd.AddressMode = QSPI_ADDRESS_4_LINES;
  cmd.AddressSize = QSPI_ADDRESS_32_BITS;
  cmd.Address = ReadAddr;
  (…)


  if (HAL_QSPI_Command(Ctx, &cmd, HAL_QPSI_TIMEOUT_DEFAULT_VALUE) !=
HAL_OK)
  {
    return MX25L512_ERROR;
  }

  if (pData && !Size)
  {
   /* Reception of the data */
    if (HAL_QSPI_Receive(Ctx, pData, HAL_QPSI_TIMEOUT_DEFAULT_VALUE) !=
HAL_OK)
    {
      return MX25L512_ERROR;
    }
  }
  return MX25L512_ERROR;
};
```

The memory command driver needs only to provide the unitary function and no common memory class structure is required.

If there is no command to return the characteristics of the memory device (capacity, block/sector size), the driver must provide a function called `NNNN_GetFlashInfo()`. Here is an example:

```
int32_t MX25L512_GetFlashInfo (MX25L512_Info_t *pInfo)
{
 /* Configure the structure with the memory configuration */
 pInfo->FlashSize          = MX25L512_FLASH_SIZE;
 pInfo->EraseSectorSize    = MX25L512_SUBSECTOR_SIZE;
 pInfo->EraseSectorsNumber =
(MX25L512_FLASH_SIZE/MX25L512_SUBSECTOR_SIZE);
 pInfo->ProgPageSize       = MX25L512_PAGE_SIZE;
 pInfo->ProgPagesNumber    = (MX25L512_FLASH_SIZE/MX25L512_PAGE_SIZE);
 return MX25L512_ERROR;
};
```

However the command set and response masks and values must be defined in the memory driver as defines.

```
/* Identification Operations */
#define MX25L512_ReadIDs_CMD                        0x9F
#define MX25L512_MULTIPLE_IO_READ_ID_CMD            0xAF
#define MX25L512_READ_SERIAL_FLASH_DISCO_PARAM_CMD  0x5A

/* Read Operations */
#define MX25L512_READ_CMD                           0x03
#define MX25L512_Read4BytesAddr_CMD                 0x13
#define MX25L512_FAST_READ_CMD                      0x0B
#define MX25L512_FAST_READ_DTR_CMD                  0x0D
#define MX25L512_FAST_READ_4_BYTE_ADDR_CMD          0x0C
#define MX25L512_FAST_READ_4_BYTE_ADDR_DTR_CMD      0x0E
```

The memory component driver header file must include the platform in use as follows:

```
#ifndef MX25L512_CONF_H
#define MX25L512_CONF_H
(…)
#include "stm32yyxx_hal.h"
(…)
#endif /* MX25L512_CONF_H */
```

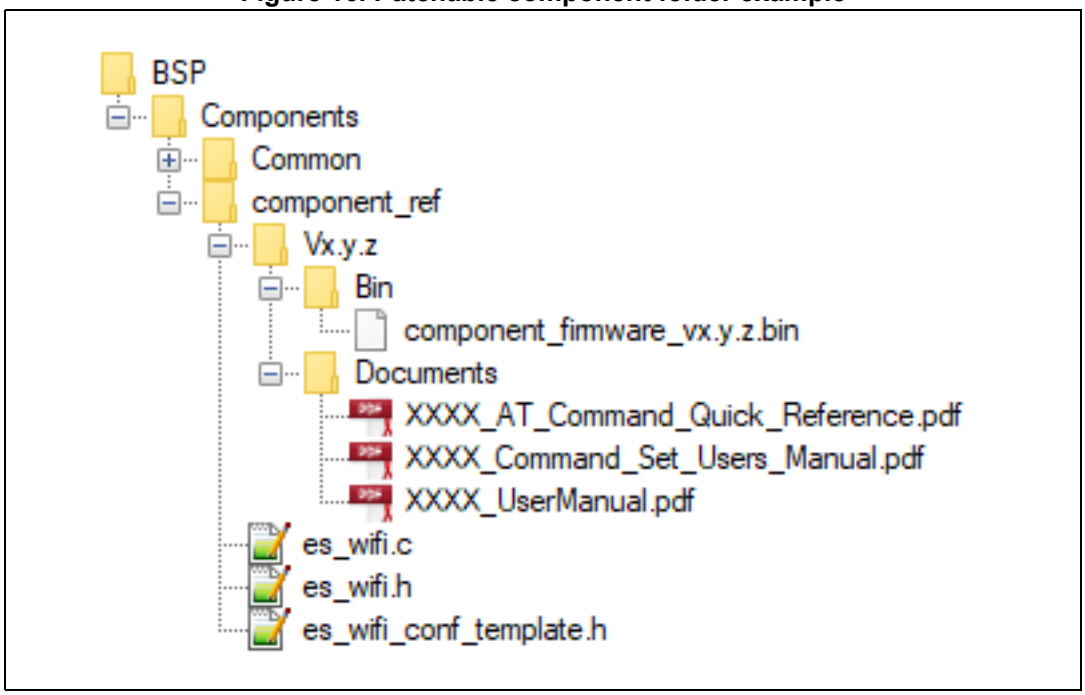
7.6 BSP component upgrade

This covers the case of active components built with internal MCU that are upgradeable. The drivers must consequently be associated to a specific firmware version and its container folder must integrate the binaries and the documentations relative to the procedure to perform the firmware upgrade.

The rationale behind this architecture approach is that the features offered by the component may be changed depending on the firmware version.

An example of patchable component folder is shown *Figure 16*.

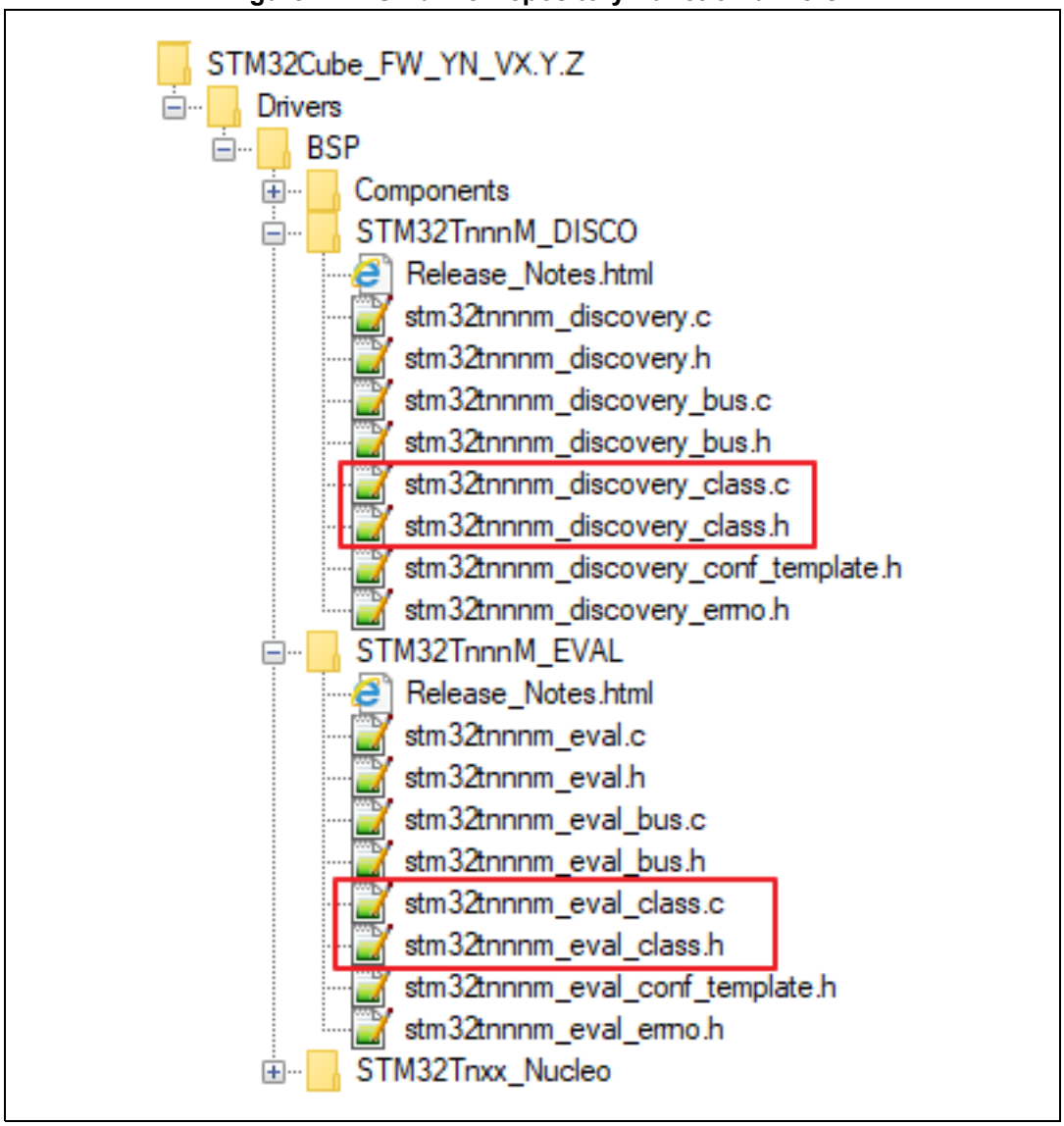
Figure 16. Patchable component folder example



8 BSP class driver

The class drivers provide a set of high-level functions for available classes, such as LCD, AUDIO or Touchscreen, for each board.

Figure 17. BSP driver repository: function drivers



Each class driver may provide three sets of functions:

1. Generic functions: common functions that must be provided by all the drivers for a given class for all the supported board
2. Specific functions: Advanced functions that are relative to specific features in the STM32 or the components available on the boards
3. Extended functions: Advanced services that may extend generic functions to provide more control on component drivers offering advanced features. The extended functions end always with the Ex suffix.

8.1 Generic common functions

The generic common functions provide most common services for a specific class and not depending on specific core or components features. These services are mandatory and must always be provided for all the BSP drivers of the same class.

Even the generic services are specific for each class driver, they must provide the following common services for all classes:

Table 9. BSP class drivers: generic common functions

Function	Description	Example
BSP_CLASS_RegisterMspCallbacks	Register Msp callbacks	int32_t BSP_AUDIO_OUT_RegisterMpsCallbacks (uint32_t Instance, BSP_AUDIO_OUT_Cb_t *CallBacks)
BSP_CLASS_RegisterDefaultMspCallbacks	Restore default Msp callbacks	int32_t BSP_AUDIO_OUT_RegisterDefaultMspCallbacks (uint32_t Instance)
BSP_CLASS_Init	Class driver generic Initialization	int32_t BSP_AUDIO_OUT_Init (uint32_t Instance, AUDIO_Init_TypeDef *AUDIO_Init)
BSP_CLASS_DeInit	Class driver generic De-initialization	int32_t BSP_AUDIO_OUT_DeInit (uint32_t Instance)
BSP_CLASS_Action	Common function for a given class	uint32_t BSP_AUDIO_OUT_Play (uint32_t Instance, uint8_t *pData, uint32_t len)

8.1.1 BSP_CLASS_Init

This function initializes the main peripheral of each class, for example I2S for audio and LTDC for Display, the associated Msp (GPIO, DMA, NVIC, Clock) and eventually the external components by linking the IO operation functions pointer and initializing the component itself by setting its internal registers to the right configuration.

Typically the BSP_CLASS_Init function, has several implementation scenarios depending on external component type (controller, memory), whether the board may use different components part number and if the external component (if used) may be identified dynamically by a single identifier through a ReadID like function.

Class with no external component

This is typically the use case when the STM32 is able to get internally the configuration from an external device and communicate with the device based on a standardized protocol. The BSP_CLASS_Init implementation is as follows:

```
uint32_t IsPppMspCbValid = 0;
int32_t BSP_CLASS_Init (uint32_t Instance, BSP_CLASS_Init_t *ClassInit)
{
  NNNXXXX_Init_t         Init;

  (…)
#if (USE_HAL_PPP_REGISTER_CALLBACKS == 0)
  /* Init the PPP Msp */
  PPP_MspInit(&hppp);
#else
  If (IsPppMspCbValid == 0)
      {
    BSP_CLASS_DefaultMspCallbacks (Instance);
   }
#endif

  if (MX_PPPx_Init(&hclass_ppp, &Init) != HAL_OK)
  {
    return BSP_ERROR_PERIPH_FAILURE;
  }


  if (MX_PPPx_ClockConfig(&hclass_ppp, Clock)!= HAL_OK)
  {
    return BSP_ERROR_CLOCK_FAILURE;
  }
  return BSP_ERROR_NONE;
}
```

Single controller component P/N

This is typically the use case when a STM32 peripheral requires an external controller that drives different signals and different data format; example: LCD controller, Audio CODEC but all board versions have always the same component reference.

```
NNNXXXX_Object_t ComponentObj;
NNNXXXX_Drv_t    ComponentDrv;

void BSP_CLASS_Init (uint32_t Instance, BSP_CLASS_Init_t  *ClassInit)
{
  NNNXXXX_IO_t           IO;
  NNNXXXX_Init_t         Init;
  (…)
  /* Configure the component */
  IO.DevAddr    = DevAddr;
  IO.Init       = BUS#N_Init;
  IO.ReadReg    = BUS#N_ReadReg;
  IO.WriteReg   = BUS#N_WriteReg;
  IO.GetTick    = BSP_GetTick;
  NNNXXXX_RegisterBusIO (&ComponentObj, &IO);
    (…)
  ComponentDrv.Init (&ComponentObj, Init);
    (…)
}
  return ret;
}
```

Multiple controller component P/N without ID:

This is typically the use case when a STM32 peripheral requires an external controller that drives different signals and different data formats; example: LCD controller, Audio CODEC. The board versions may have different references of components but the components don’t return IDs.

```
#ifdef BSP_CLASS_USE_NNNXXXXA
NNNXXXX_A_Object_t ComponentObj;
#else
#ifdef BSP_CLASS_USE_NNNXXXXB
NNNXXXX_B_Object_t ComponentObj;
#endif
#endif
CLASS_DrvTypeDef      *Class_Drv;
void                  *Class_CompObj;

uint32_t BSP_CLASS_Init (uint32_t Instance, BSP_CLASS_Init_t *ClassInit)
{
    (…)
#ifdef BSP_CLASS_USE_NNNXXXXA
    if (NNNXXXX_A_Probe((void *)&CodecAddress, ClassInit))
#else
#ifdef BSP_CLASS_USE_NNNXXXXB
    if (NNNXXXX_B_Probe((void *)&CodecAddress, ClassInit))
#endif
#endif
    {
       return BSP_ERROR_UNKNOWN_COMPONENT;
    }
    (…)
}
static int32_t NNNXXXX_A_Probe(void *Ctx, BSP_CLASS_Init_t *ClassInit)
{
  static NNNXXXX_A_Object_t   NNNXXXX_Obj;
  NNNXXXX_A_IO_t           IO;
  NNNXXXX_A_Init_t            Init;

  /* Configure the component */
  IO.DevAddr       = DevAddr;
  IO.DevAddr    = DevAddr;
  IO.Init       = BUS#N_Init;
  IO.ReadReg    = BUS#N_ReadReg;
  IO.WriteReg   = BUS#N_WriteReg;
  IO.GetTick    = BSP_GetTick;
  NNNXXXX_A_RegisterBusIO (&NNNXXXX_Obj, &IO);

   (…)
    If (NNNXXXX_A_Init (&NNNXXXX_Obj, &Init) ==0)
    {
      Class_Drv = (CLASS_Drv_t *) &NNNXXXX_drv;
```

```
 CompObj = &NNNXXXX_Obj;
      return BSP_ERROR_NONE;
    }
  return BSP_ERROR_UNKNOWN_COMPONENT;
}
```

Multiple controller component P/N with ID

This is typically the use case when a STM32 peripheral requires an external controller that drives different signals and different data format; example: LCD controller, Audio CODEC. The board versions may have different references of components but the components don’t return IDs.

```
NNNXXXX_A_Object_t ComponentObjA;
NNNXXXX_B_Object_t ComponentObjB;
CLASS_DrvTypeDef      *Class_Drv;
void                  *Class_CompObj;

uint32_t BSP_CLASS_Init (BSP_CLASS_Init_t *ClassInit)
{
    (…)
    if (NNNXXXX_A_Probe((void *)&CodecAddress, ClassInit))
    {
       return BSP_ERROR_UNKNOWN_COMPONENT;
    }
    else if (NNNXXXX_B_Probe((void *)&CodecAddress, ClassInit))
    {
       return BSP_ERROR_UNKNOWN_COMPONENT;
    }
    (…)
}

static int32_t NNNXXXX_A_Probe(void *Ctx, BSP_CLASS_Init_t *ClassInit)
{
  static NNNXXXX_A_Object_t   NNNXXXX_Obj;
  NNNXXXX_A_IO_t           IO;
  NNNXXXX_A_Init_t         Init;

  /* Configure the component */
  IO.DevAddr    = DevAddr;
  IO.Init       = BUS#N_Init;
  IO.ReadReg    = BUS#N_ReadReg;
  IO.WriteReg   = BUS#N_WriteReg;
  IO.GetTick    = BSP_GetTick;
  NNNXXXX_A_RegisterBusIO (&NNNXXXX_A_Obj, &IO);

   (…)
 If (NNNXXXX_A_Init (&NNNXXXX_A_Obj, &Init) ==0)
    {
      Class_Drv = (CLASS_Drv_t *) &NNNXXXX_A_drv;
      CompObj = &NNNXXXX_A_Obj;
      return BSP_ERROR_NONE;
    }
  return BSP_ERROR_UNKNOWN_COMPONENT;
}
```

Specific command based memories

This is typically the use case when the STM32 may interface a standard memory that requires specific part number commands and specific compile time configuration.

The BSP_CLASS_Init() follows the same architecture as the BSP class based on the second component driver model (Single controller component P/N) except there is no need to register IO operation and the component architecture is based on the memory component model.

Below an example of the BSP_QSPI_Init ():

```
int32_t BSP_QSPI_Init(uint32_t Interface)
{
  MX25L512_Info_t Info;
  hqspi.Instance = QUADSPI;

  /* Get QSPI information */
  MX25L512_GetFlashInfo (&Info);
  /* QSPI initialization */
  If (MX_QSPI_Init (&hqspi, Info.FlashSize) != MX25L512_ERROR)
  {
    return BSP_ERROR_PERIPH_FAILURE;
  }

   /* Reset the QSPI */
  if (QSPI_ResetMemory (Interface) != MX25L512_ERROR)
  {
    return BSP_ERROR_COMPONENT_FAILURE;
  }
  /* Put QSPI memory in QPI mode */
  if (MX25L512_EnterQuadMode (&hqspi) < MX25L512_ERROR)
  {
    return BSP_ERROR_COMPONENT_FAILURE;
  }
  /* Set the QSPI memory in 4-bytes address mode */
  if (QSPI_EnterFourBytesAddress(Interface) != MX25L512_ERROR)
  {
    return BSP_ERROR_COMPONENT_FAILURE;
  }
  /* Configuration of the dummy cycles on QSPI memory side */
  if (QSPI_DummyCyclesCfg(Interface) != MX25L512_ERROR)
  {
    return BSP_ERROR_COMPONENT_FAILURE;
  }
  /* Configuration of the Output driver strength on memory side */

if (QSPI_OutDrvStrengthCfg(Interface) != MX25L512_ERROR)
  {
  return BSP_ERROR_COMPONENT_FAILURE;
  }
  return BSP_ERROR_NONE;
}
```

8.1.2 BSP_CLASS_DeInit

This function de-initializes the class main peripheral, for example I2S for audio and LTDC for Display, and the associated Msp (GPIO, DMA, NVIC, Clock). Note that the GPIO port clocks are not disabled as they may be shared by other BSP drivers or the user code. The BSP_CLASS_DeInit implementation, is as follows:

```
uint32_t BSP_CLASS_DeInit (uint32_t Instance)
{
  int32_t ret  = BSP_ERROR_NONE;

#if (USE_HAL_PPP_REGISTER_CALLBACKS == 0)
  /* Init the PPP Msp */
  PPP_MspDeInit(&hclass_ppp);
#else
  IsPppMspCbValid = 0;
#endif

 /* De-initialize the PPP peripheral */
  if (HAL_PPP_DeInit (&hclass_ppp)!= HAL_OK)
  {
    ret = BSP_ERROR_BUS_FAILURE;
  }
  return ret;
}
```

8.1.3 BSP_CLASS_RegisterDefaultMspCallbacks

The BSP_CLASS_RegisterDefaultMspCallbacks allows to assign the default callbacks to the main peripherals handles when they are changed by the user.

```
int32_t BSP_CLASS_RegisterDefaultMspCallbacks (uint32_t Instance)
{
   HAL_PPP_RegisterCallback (&hclass_ppp, HAL_PPP_MSPINIT_CB_ID,
PPPx_MspInit);
   HAL_PPP_RegisterCallback (&hclass_ppp, HAL_PPP_MSPDEINIT_CB_ID,
PPPx_MspDeInit);
   IsPppMspCbValid = 1;
   return BSP_ERROR_NONE;
}
```

8.1.4 BSP_CLASS_RegisterMspCallbacks

The BSP_CLASS_RegisterMspCallbacks allows to assign the Msp callbacks to the main peripherals handles.

```
int32_t BSP_CLASS_RegisterMspCallbacks (uint32_t Instance, BSP_CLASS_Cb_t
*CallBacks)
{
  HAL_PPP_RegisterCallback (&hclass_ppp, HAL_PPP_MSPINIT_CB_ID, CallBacks-
>pMspInitCb);
  HAL_PPP_RegisterCallback (&hclass_ppp, HAL_PPP_MSPDEINIT_CB_ID,
CallBacks->pMspDeInitCb);
  IsPppMspCbValid = 1;
  return BSP_ERROR_NONE;
}
```

8.1.5 RegisterMspCallbacks example

This section gives an example of register Msp callbacks implementation for BUS BSP driver based on I^2C peripheral.

```
uint32_t IsI2c1MspCbValid = 0;
#if (USE_HAL_I2C_REGISTER_CALLBACKS == 1)
/**
  * @brief Default BSP I2C1 Bus Msp Callbacks
  * @param Instance    I2C instance
  * @retval BSP status
  */
int32_t BSP_I2C1_RegisterDefaultMspCallbacks (uint32_t Instance)
{
  /* Prevent unused argument(s) compilation warning */
  UNUSED(Instance);

  __HAL_I2C_RESET_HANDLE_STATE(&hbus_i2c1);

  /* Register MspInit Callback */
  HAL_I2C_RegisterCallback(&hbus_i2c1, HAL_I2C_MSPINIT_CB_ID,
I2C1_MspInit);

  /* Register MspDeInit Callback */
  HAL_I2C_RegisterCallback(&hbus_i2c1, HAL_I2C_MSPDEINIT_CB_ID,
I2C1_MspDeInit);

  IsI2c1MspCbValid = 1;
  return BSP_ERROR_NONE;
}

/**
  * @brief BSP I2C1 Bus Msp Callback registering
  * @param Instance      I2C instance
  * @param pMspInitCb    pointer to MspInit callback function
  * @param pMspDeInitCb  pointer to MspDeInit callback function
```

```
 * @retval BSP status
  */
int32_t BSP_I2C1_RegisterMspCallbacks (uint32_t Instance, BSP_I2C_Cb_t
*Callbacks)
{
  /* Prevent unused argument(s) compilation warning */
  UNUSED(Instance);

  __HAL_I2C_RESET_HANDLE_STATE(&hbus_i2c1);

  /* Register MspInit Callback */
  HAL_I2C_RegisterCallback(&hbus_i2c1, HAL_I2C_MSPINIT_CB_ID, Callbacks ->
pMspInitCb);

  /* Register MspDeInit Callback */
  HAL_I2C_RegisterCallback(&hbus_i2c1, HAL_I2C_MSPDEINIT_CB_ID, Callbacks-
> pMspDeInitCb);

  IsI2c1MspCbValid = 1;

  return BSP_ERROR_NONE;
}
#endif /* USE_HAL_I2C_REGISTER_CALLBACKS */
```

In the Init bus function, a check on the Msp registration callback usage must be done:

```
static uint32_t I2C1InitCounter = 0;
int32_t BSP_I2C1_Init(void)
{
  int32_t ret = BSP_ERROR_NONE;

  hbus_i2c1.Instance  = BUS_I2C1;

  if(I2C1InitCounter++ == 0U)
  {
    if(HAL_I2C_GetState(&hbus_i2c1) == HAL_I2C_STATE_RESET)
    {
#if (USE_HAL_I2C_REGISTER_CALLBACKS == 0)
      /* Init the I2C Msp */
      I2C1_MspInit(&hbus_i2c1);
#else
      if(IsFmpi2c1MspCbValid == 0U)
      {
        if(BSP_I2C1_RegisterDefaultMspCallbacks() != BSP_ERROR_NONE)
```

```
        {
          ret = BSP_ERROR_MSP_FAILURE;
        }
      }
#endif
      if(ret == BSP_ERROR_NONE)
      {
        /* Init the I2C */
       if (MX_I2C1_Init(&hbus_i2c1, I2C_GetTiming (HAL_RCC_GetPCLK1Freq(),
BUS_I2C1_FREQUENCY)) != HAL_OK)
        {
          ret = BSP_ERROR_BUS_FAILURE;
        }
        else
        {
          if( HAL_I2CEx_ConfigAnalogFilter(&hbus_i2c1,
I2C_ANALOGFILTER_ENABLE) != HAL_OK)
          {
            ret = BSP_ERROR_BUS_FAILURE;
          }
        }
      }
    }
  }

  return ret;
}
```

Note: A given bus among BSP bus drivers maybe used by several classes. To insure that Class_A DeInit doesn't affect Class_B, a global variable `PPPnInitCounter` is used to check whether we must proceed or not for Init or DeInit peripheral operations.

The De-Init bus function must reset the Msp registration callback usage:

```
int32_t BSP_I2C1_DeInit(void)
{
  int32_t ret  = BSP_ERROR_NONE;

  if (I2C1InitCounter > 0U)
  {
    if (--I2C1InitCounter == 0U)
    {
#if (USE_HAL_I2C_REGISTER_CALLBACKS == 0)
      I2C1_MspDeInit(&hbus_i2c1);
#endif /* (USE_HAL_I2C_REGISTER_CALLBACKS == 0) */

      /* Init the I2C */
      if (HAL_I2C_DeInit(&hbus_i2c1) != HAL_OK)
      {
        ret = BSP_ERROR_BUS_FAILURE;
      }
    }
  }

  return ret;
}
```

8.1.6 BSP_CLASS_Action

This is the generic name of the class specific functions. Each class must provide frequently used functions to bring the high-level services to the end user, by hiding the main peripheral and the Msp configuration complexity. The prototype for such function is as follows:

```
int32_t BSP_CLASS_Action (uint32_t Instance, __ARGS__);
```

Note: __ARGS__: the list of arguments of the BSP_CLASS_Action function.

An example of BSP_CLASS_Action is:

```
int32_t BSP_SD_ReadBlocks (uint32_t Instance, uint32_t *pData, uint32_t
BlockIdx, uint32_t BlocksNbr)
```

8.2 Specific functions

The specific drivers services are set of advanced functions that are relative to specific features in a STM32 serie or a component:

Table 10. BSP class drivers: Specific functions

Class	Function
AUDIO_IN	int32_t BSP_AUDIO_IN_RecordChannels(uint32_t Instance, uint8_t **pBuf, uint32_t NbrOfBytes);
LCD	int32_t BSP_LCD_SetLayerAddress(uint32_t Instance, uint32_t LayerIndex, uint32_t Address) int32_t BSP_LCD_SetLayerWindow(uint32_t Instance, uint16_t LayerIndex, uint16_t Xpos, uint16_t Ypos, uint16_t Width, uint16_t Height) int32_t BSP_LCD_SetLayerVisible(uint32_t Instance, uint32_t LayerIndex, FunctionalState State) int32_t BSP_LCD_ConfigLayer (uint32_t Instance, uint32_t LayerIndex, BSP_LCD_LayerConfig_t *Config)

8.3 Extended functions

The extended functions are implementing a set of advanced services that extend existing generic functions to provide more control on components drivers offering advanced features. Contrary to the specific functions, the extended ones are not mandatory as equivalent services are already covered by the common functions. The extended functions extend the common functions by providing a special services set specific for a function driver.

An example of such services is described in the *Table 11* below for LCD driver:

Table 11. BSP class drivers: Extended functions

Driver	Services
Common	int32_t BSP_LCD_Init(uint32_t Instance, LCD_OrientationTypeDef Orientation)
Extended	int32_t BSP_LCD_InitEx(uint32_t Instance, uint32_t Orientation, uint32_t PixelFormat, uint32_t Width, uint32_t Height)

8.4 BSP classes

This section lists the APIs, for each class, that must be implemented. MX_PPP_Init and MX_PPP_ClockConfig APIs are specially prefixed with "MX_" as they can be generated later by STM32CubeMX tool.

8.4.1 BSP AUDIO OUT class APIs

Table 12. BSP AUDIO OUT class APIs

Function	Arguments	description
int32_t BSP_AUDIO_OUT_Init	uint32_t Instance BSP_AUDIO_Init_t *AUDIO_Init	AUDIO OUT Initialization
int32_t BSP_AUDIO_OUT_DeInit	uint32_t Instance	AUDIO OUT De-Initialization
__weak HAL_StatusTypeDef MX_PPP_Init	PPP_HandleTypeDef *hppp MX_PPP_TypeDef *Init	AUDIO OUT periph configuration
static HAL_StatusTypeDef PPP_MspInit	PPP_HandleTypeDef *hppp	Initializes AUDIO OUT MSP part. To be used with RegisterCallbacks
static HAL_StatusTypeDef PPP_MspDeInit	PPP_HandleTypeDef *hppp	De-Initializes AUDIO OUT MSP part To be used with RegisterCallbacks
__weak HAL_StatusTypeDef MX_PPP_ClockConfig	PPP_HandleTypeDef *hppp uint32_t SampleRate	Set AUDIO OUT clock configuration
int32_t BSP_AUDIO_OUT_Play	uint32_t Instance uint32_t* pData uint32_t NbrOfBytes	Start audio play
int32_t BSP_AUDIO_OUT_Pause	uint32_t Instance	Pause audio play
int32_t BSP_AUDIO_OUT_Resume	uint32_t Instance	Resume audio play
int32_t BSP_AUDIO_OUT_Stop	uint32_t Instance	Stop audio play
int32_t BSP_AUDIO_OUT_Mute	uint32_t Instance	AUDIO OUT mute
int32_t BSP_AUDIO_OUT_UnMute	uint32_t Instance	AUDIO OUT unmute
int32_t BSP_AUDIO_OUT_IsMute	uint32_t Instance	Returns 1(mute), 0(unmute)
int32_t BSP_AUDIO_OUT_SetVolume	uint32_t Instance uint32_t Volume Volume level to be set in percentage from 0% to 100% (0 for Mute and 100 for Max volume level).	Set AUDIO OUT volume
int32_t BSP_AUDIO_OUT_GetVolume	uint32_t Instance uint32_t *Volume	Get AUDIO OUT volume
int32_t BSP_AUDIO_OUT_SetSampleRate	uint32_t Instance uint32_t SampleRate	Set AUDIO OUT frequency
int32_t BSP_AUDIO_OUT_GetSampleRate	uint32_t Instance uint32_t *SampleRate	Get AUDIO OUT frequency

Table 12. BSP AUDIO OUT class APIs (continued)

Function	Arguments	description
int32_t BSP_AUDIO_OUT_SetDevice	uint32_t Instance uint32_t Device	Set AUDIO OUT device
int32_t BSP_AUDIO_OUT_GetDevice	uint32_t Instance uint32_t *Device	Get AUDIO OUT device
int32_tBSP_AUDIO_OUT_SetBitsPerSample	uint32_t Instance uint32_t BitsPerSample	Set AUDIO OUT resolution
int32_t SP_AUDIO_OUT_GetBitsPerSample	uint32_t Instance uint32_t *BitsPerSample	Get AUDIO OUT resolution
int32_t BSP_AUDIO_OUT_SetChannelsNbr	uint32_t Instance uint32_t ChannelNbr	Set AUDIO OUT channels number
int32_t BSP_AUDIO_OUT_GetChannelsNbr	uint32_t Instance uint32_t *ChannelNbr	Get AUDIO OUT channels number
int32_t BSP_AUDIO_OUT_GetState	uint32_t Instance uint32_t *State	Get AUDIO OUT state : AUDIO_OUT_STATE_RESET AUDIO_OUT_STATE_PLAYING AUDIO_OUT_STATE_STOP AUDIO_OUT_STATE_PAUSE
void BSP_AUDIO_OUT_DMA_IRQHandler	uint32_t Instance uint32_t Device	Handles audio OUT DMA transfer request depending on used instance and device
void BSP_AUDIO_OUT_TransferComplete_CallBack	uint32_t Instance	Transfer Complete callback
void BSP_AUDIO_OUT_HalfTransfer_CallBack	uint32_t Instance	Half Transfer callback
void BSP_AUDIO_OUT_Error_CallBack	uint32_t Instance	Error callback
int32_t BSP_AUDIO_OUT_RegisterMspCallbacks	uint32_t Instance BSP_AUDIO_OUT_Cb_t *CallBacks	Register AUDIO OUT Msp callbacks
int32_t BSP_AUDIO_OUT_RegisterDefaultMspCallbacks	uint32_t Instance	Register AUDIO OUT default Msp callbacks

With:

```
typedef struct
{
  uint32_t                    Device;
  uint32_t                    SampleRate;
  uint32_t                    BitsPerSample;
  uint32_t                    ChannelsNbr;
  uint32_t                    Volume;
}BSP_AUDIO_Init_t;
```

`MX_PPP_TypeDef` can be:

```
typedef struct
{
  uint32_t AudioFrequency;
  uint32_t AudioMode;
  uint32_t DataSize;
  uint32_t MonoStereoMode;
  uint32_t ClockStrobing;
  uint32_t Synchro;
  uint32_t OutputDrive;
  uint32_t SynchroExt;
  uint32_t FrameLength;
  uint32_t ActiveFrameLength;
  uint32_t SlotActive;
}MX_SAI_TypeDef;

typedef struct
{
  uint32_t AudioFrequency;
  uint32_t AudioMode;
  uint32_t FullDuplexMode;
}MX_I2S_TypeDef;
```

```
#if(USE_PPP_REGISTER_CALLBACKS == 1)
typedef struct
{
void (* pMspInitCb)(PPP_HandleTypeDef *);
```

```
void (* pMspDeInitCb)(PPP_HandleTypeDef *);
}BSP_AUDIO_OUT_Cb_t;
#endif /* (USE_HAL_PPP_REGISTER_CALLBACKS == 1) */
```

8.4.2 BSP AUDIO IN class APIs

Table 13. BSP AUDIO IN class APIs

Function	Arguments	description
int32_t BSP_AUDIO_IN_Init	uint32_t Instance BSP_AUDIO_Init_t *AUDIO_Init	AUDIO IN Initialization
int32_t BSP_AUDIO_IN_DeInit	uint32_t Instance	AUDIO IN De-Initialization
__weak HAL_StatusTypeDef MX_PPP_Init	PPP_HandleTypeDef *hppp MX_PPP_TypeDef *Init	AUDIO IN periph configuration
static HAL_StatusTypeDef PPP_MspInit	PPP_HandleTypeDef *hppp	Initializes default AUDIO OUT MSP part
static HAL_StatusTypeDef PPP_MspDeInit	PPP_HandleTypeDef *hppp	De-Initializes AUDIO IN MSP part
__weak HAL_StatusTypeDef MX_PPP_ClockConfig	PPP_HandleTypeDef *hppp uint32_t SampleRate	Set AUDIO IN clock configuration
int32_t BSP_AUDIO_IN_Record	uint32_t Instance uint8_t* pBuf uint32_t NbrOfBytes	Start audio Recording in default mode
int32_t BSP_AUDIO_IN_Pause	uint32_t Instance	Pause audio recording
int32_t BSP_AUDIO_IN_Resume	uint32_t Instance	Resume audio recording
int32_t BSP_AUDIO_IN_Stop	uint32_t Instance	Stop audio recording
int32_t BSP_AUDIO_IN_SetDevice	uint32_t Instance uint32_t Device	Set AUDIO IN input device
int32_t BSP_AUDIO_IN_GetDevice	uint32_t Instance uint32_t *Device	Get AUDIO IN input device
int32_t BSP_AUDIO_IN_SetSampleRate	uint32_t Instance uint32_t SampleRate	Set AUDIO IN frequency
int32_t BSP_AUDIO_IN_GetSampleRate	uint32_t Instance uint32_t * SampleRate	Get AUDIO IN frequency
int32_t BSP_AUDIO_IN_SetBitsPerSample	uint32_t Instance uint32_t BitsPerSample	Set AUDIO IN sample resolution
int32_t BSP_AUDIO_IN_GetBitsPerSample	uint32_t Instance uint32_t *BitsPerSample	Get AUDIO IN sample resolution
int32_t BSP_AUDIO_IN_SetVolume	uint32_t Instance uint32_t Volume	Set AUDIO IN volume
int32_t BSP_AUDIO_IN_GetVolume	uint32_t Instance uint32_t *Volume	Get AUDIO IN volume
int32_t BSP_AUDIO_IN_SetChannelsNbr	uint32_t Instance uint32_t ChannelNbr	Set AUDIO IN channels number
int32_t BSP_AUDIO_IN_GetChannelsNbr	uint32_t Instance uint32_t *ChannelNbr	Get AUDIO IN channels number

Table 13. BSP AUDIO IN class APIs (continued)

Function	Arguments	description
int32_t BSP_AUDIO_IN_GetState	uint32_t Instance uint32_t *State	Get AUDIO IN state: AUDIO_IN_STATE_RESET AUDIO_IN_STATE_RECORDING AUDIO_IN_STATE_STOP AUDIO_IN_STATE_PAUSE
void BSP_AUDIO_IN_DMA_IRQHandler	uint32_t Instance uint32_t Device	Handles audio IN DMA transfer request depending on used instance and device
void BSP_AUDIO_IN_TransferComplete_CallBack	uint32_t Instance	Transfer Complete callback
void BSP_AUDIO_IN_HalfTransfer_CallBack	uint32_t Instance	Half Transfer callback
void BSP_AUDIO_IN_Error_CallBack	uint32_t Instance	Error callback
int32_t BSP_AUDIO_IN_RegisterMspCallbacks	uint32_t Instance BSP_AUDIO_IN_Cb_t *CallBacks	Register AUDIO IN Msp callbacks
int32_t BSP_AUDIO_IN_RegisterDefaultMspCallbacks	uint32_t Instance	Register AUDIO IN default Msp callbacks

Where MX_PPP_TypeDef can be:

```
typedef struct
{
  /* Filter parameters */
  DFSDM_Filter_TypeDef   *FilterInstance;
  uint32_t               RegularTrigger;
  uint32_t               SincOrder;
  uint32_t               Oversampling;
  /* Channel parameters */
  DFSDM_Channel_TypeDef *ChannelInstance;
  uint32_t              DigitalMicPins;
  uint32_t              DigitalMicType;
  uint32_t              Channel4Filter;
  uint32_t              ClockDivider;
  uint32_t              RightBitShift;
}MX_DFSDM_TypeDef;
```

```
#if(USE_PPP_REGISTER_CALLBACKS == 1)
typedef struct
{
void (* pMspInitCb)(PPP_HandleTypeDef *);
void (* pMspDeInitCb)(PPP_HandleTypeDef *);
}BSP_AUDIO_IN_Cb_t;
#endif /* (USE_HAL_PPP_REGISTER_CALLBACKS == 1) */
```

Note: ***When using DFSDM IP for recording, the following specific APIs must be implemented:***

Table 14. Specific APIs for recording

Function	Arguments	description
BSP_AUDIO_IN_RecordChannels	uint32_t Instance, uint8_t **pBuf, uint32_t NbrOfBytes	Start recording using digital microphones already initialized. The data is to be processed at application level.
BSP_AUDIO_IN_StopChannels	uint32_t Instance, uint32_t Device	Stop recording on a given digital microphone
BSP_AUDIO_IN_PauseChannels	uint32_t Instance, uint32_t Device	Pause recording on a given digital microphone
BSP_AUDIO_IN_ResumeChannels	uint32_t Instance, uint32_t Device	Resume recording on a given digital microphone

Be aware of DFSDM filters synchronization while developing these APIs.

8.4.3 BSP TS class APIs

Table 15. BSP TS class APIs

Function	Arguments	description
int32_t BSP_TS_Init	uint32_t Instance TS_Init_t *TS_Init	Touch Screen Initialization
int32_t BSP_TS_DeInit	uint32_t Instance	Touch Screen De-Initialization
int32_t BSP_TS_GetState	uint32_t Instance TS_State_t *TS_State	Returns status and positions of the touch screen
int32_t BSP_TS_GetMultiTouchState	uint32_t Instance TS_MultiTouch_State_t *TS_MultiTouchState	Returns status and positions of the touch screen for multi-points
int32_t BSP_TS_GestureConfig	uint32_t Instance TS_Gesture_Config_t *GestureConfig	Configure Touch Screen gesture

Table 15. BSP TS class APIs (continued)

Function	Arguments	description
int32_t BSP_TS_Get_Gesture	uint32_t Instance	Returns the gesture: GESTURE_ID_NO_GESTURE GESTURE_ID_MOVE_UP GESTURE_ID_MOVE_RIGHT GESTURE_ID_MOVE_DOWN GESTURE_ID_MOVE_LEFT GESTURE_ID_ZOOM_IN GESTURE_ID_ZOOM_OUT
int32_t BSP_TS_Set_Orientation	uint32_t Instance uint32_t Orientation	Set touch screen orientation
int32_t BSP_TS_Get_Orientation	uint32_t Instance uint32_t *Orientation	Get touch screen orientation
int32_t BSP_TS_EnableIT	uint32_t Instance	Enable IT mode
int32_t BSP_TS_DisableIT	uint32_t Instance	Disable IT mode
int32_t BSP_TS_GetCapabilities	uint32_t Instance TS_Capabilities_t *Capabilities	Get Touch Screen capabilities
void BSP_TS_IRQHandler	uint32_t Instance	Handles TS interrupt request

Where:

```
typedef struct
{
  uint32_t   Width;                  /* Screen Width */
  uint32_t   Height;                 /* Screen Height */
  uint32_t   Orientation;            /* Touch Screen orientation from the
upper left position  */
  uint32_t   Accuracy;              /* Expressed in pixel and means the x
or y difference vs old
                                       position to consider the new values
valid */
}TS_Init_t;
```

```
typedef struct
{
  uint32_t  TouchDetected;
  uint32_t  TouchX;
  uint32_t  TouchY;
} TS_State_t;
```

```
typedef struct
{
  uint32_t  TouchDetected;
  uint32_t  TouchX[TS_TOUCH_NBR];
  uint32_t  TouchY[TS_TOUCH_NBR];
  uint32_t  TouchWeight[TS_TOUCH_NBR];
  uint32_t  TouchEvent[TS_TOUCH_NBR];
  uint32_t  TouchArea[TS_TOUCH_NBR];
} TS_MultiTouch_State_t;
```

```
typedef struct
{
  uint32_t  Radian;
  uint32_t  OffsetLeftRight;
  uint32_t  OffsetUpDown;
  uint32_t  DistanceLeftRight;
  uint32_t  DistanceUpDown;
  uint32_t  DistanceZoom;
}TS_Gesture_Config_t;
```

```
typedef struct
{
  uint8_t   MultiTouch;
  uint8_t   Gesture;
  uint8_t   MaxTouch;
  uint32_t  MaxXl;
  uint32_t  MaxYl;
} TS_Capabilities_t;
```

8.4.4 BSP SD class APIs

Table 16. BSP SD class APIs

Function	Arguments	description
int32_t BSP_SD_Init	uint32_t Instance	SD Instance Initialization: Instance can be SD_INTERFACE_SDMMC or SD_INTERFACE_SDMMC2
int32_t BSP_SD_DeInit	uint32_t Instance	SD Instance De-Initialization: Instance can be SD_INTERFACE_SDMMC or SD_INTERFACE_SDMMC2
HAL_StatusTypeDef MX_SDMMCx_Init	uint32_t Instance	SDMMCx MX initialization X can be 1 or 2 depends on the number of uSD on the board
static HAL_StatusTypeDef SD_MspInit	SD_HandleTypeDef *hsd	SD Msp initialization

Table 16. BSP SD class APIs (continued)

Function	Arguments	description
static HAL_StatusTypeDef SD_MspDeInit	SD_HandleTypeDef *hsd	SD Msp de-initialization
int32_t BSP_SD_DetectITConfig	uint32_t Instance	Configure detect pin in interrupt mode
int32_t BSP_SD_ReadBlocks	uint32_t Instance uint32_t *pData uint32_t BlockIdx uint32_t BlocksNbr	Read card blocks in polling mode
int32_t BSP_SD_WriteBlocks	uint32_t Instance uint32_t *pData uint32_t BlockIdx uint32_t BlocksNbr	Write card blocks in polling mode
int32_t BSP_SD_ReadBlocks_DMA	uint32_t Instance uint32_t *pData uint32_t BlockIdx uint32_t BlocksNbr	Read card blocks in DMA mode
int32_t BSP_SD_WriteBlocks_DMA	uint32_t Instance uint32_t *pData uint32_t BlockIdx uint32_t BlocksNbr	Write card blocks in DMA mode
int32_t BSP_SD_ReadBlocks_IT	uint32_t Instance uint32_t *pData uint32_t BlockIdx uint32_t BlocksNbr	Read card blocks in IT mode
int32_t BSP_SD_WriteBlocks_IT	uint32_t Instance uint32_t *pData uint32_t BlockIdx uint32_t BlocksNbr	Write card blocks in IT mode
int32_t BSP_SD_Erase	uint32_t Instance uint32_t BlockIdx uint32_t BlocksNbr	Erase card from start address to end address
int32_t BSP_SD_GetCardState	uint32_t Instance	-
int32_t BSP_SD_GetCardInfo	uint32_t Instance BSP_SD_CardInfo *CardInfo	Get card info
int32_t BSP_SD_IsDetected	uint32_t Instance	Detect card presence
void BSP_SD_DMA_IRQHandler	uint32_t Instance	Handles SD DMA transfer interrupt request
void BSP_SD_IRQHandler	uint32_t Instance	Handles SD interrupt request
void BSP_SD_DETECT_IRQHandler	uint32_t Instance	Handles SD detect pin interrupt request
void BSP_SD_AbortCallback	uint32_t Instance	Abort callback
void BSP_SD_WriteCpltCallback	uint32_t Instance	Write complete callback

Table 16. BSP SD class APIs (continued)

Function	Arguments	description
void BSP_SD_ReadCpltCallback	uint32_t Instance	Read complete callback
int32_t BSP_SD_RegisterMspCallbacks	uint32_t Instance BSP_SD_Cb_t *CallBacks	Register SD Msp callbacks
int32_t BSP_SD_RegisterDefaultMspCallbacks	uint32_t Instance	Register SD default Msp callbacks

Note: BSP_SD_DMA_IRQHandler can be omitted if SD IP has its own DMA.

With:

```
#define BSP_SD_CardInfo HAL_SD_CardInfoTypeDef
#if (USE_HAL_SD_REGISTER_CALLBACKS == 1)
typedef struct
{
  void (* pMspInitCb)(SD_HandleTypeDef *);
  void (* pMspDeInitCb)(SD_HandleTypeDef *);
}BSP_SD_Cb_t;
#endif /* (USE_HAL_SD_REGISTER_CALLBACKS == 1) */
```

8.4.5 BSP SDRAM class APIs

Table 17. BSP SDRAM class APIs

Function	Arguments	description
int32_t BSP_SDRAM_Init	uint32_t Instance	SDRAM class Initialization
int32_t BSP_SDRAM_DeInit	uint32_t Instance	SDRAM class De-Initialization
HAL_StatusTypeDef MX_SDRAM_Init	SDRAM_HandleTypeDef *hSdram	SDRAM MX initialization
static HAL_StatusTypeDef SDRAM_MspInit	SDRAM_HandleTypeDef *hSdram	SDRAM Msp initialization
static HAL_StatusTypeDef SDRAM_MspDeInit	SDRAM_HandleTypeDef *hSdram	SDRAM Msp de-initialization
int32_t BSP_SDRAM_SendCmd	uint32_t Instance FMC_SDRAM_CommandTypeDef *SdramCmd	Send Command to SDRAM
void BSP_SDRAM_DMA_IRQHandler	uint32_t Instance	Handles SDRAM DMA transfer interrupt request
int32_t BSP_SDRAM_RegisterMspCallbacks	uint32_t Instance BSP_SDRAM_Cb_t *CallBacks	Register SDRAM MSP callbacks
int32_t BSP_SDRAM_RegisterDefaultMspCallbacks	uint32_t Instance	Default SDRAM MSP callbacks

With:

```
#if (USE_HAL_SDRAM_REGISTER_CALLBACKS == 1)
typedef struct
{
  void (* pMspInitCb)(SDRAM_HandleTypeDef *);
  void (* pMspDeInitCb)(SDRAM_HandleTypeDef *);
}BSP_SDRAM_Cb_t;
#endif /* (USE_HAL_SDRAM_REGISTER_CALLBACKS == 1) */
```

8.4.6 BSP SRAM class APIs

Table 18. BSP SRAM class APIs

Function	Arguments	description
int32_t BSP_SRAM_Init	uint32_t Instance	SRAM class Initialization
int32_t BSP_SRAM_DeInit	uint32_t Instance	SRAM class De-Initialization
HAL_StatusTypeDef MX_SRAM_Init	SRAM_HandleTypeDef *hSram	SRAM MX initialization
static HAL_StatusTypeDef SRAM_MspInit	SRAM_HandleTypeDef *hSram	SRAM Msp initialization
static HAL_StatusTypeDef SRAM_MspDeInit	SRAM_HandleTypeDef *hSram	SRAM Msp de-initialization
void BSP_SRAM_DMA_IRQHandler	uint32_t Instance	Handles SRAM DMA transfer interrupt request
int32_t BSP_SRAM_RegisterMspCallbacks	uint32_t Instance BSP_SRAM_Cb_t *CallBacks	Register DRAM MSP callbacks
int32_t BSP_SRAM_RegisterDefaultMspCallbacks	uint32_t Instance	Register Default SDRAM MSP callbacks

With:

```
#if (USE_HAL_SRAM_REGISTER_CALLBACKS == 1)
typedef struct
{
  void (* pMspInitCb)(SRAM_HandleTypeDef *);
  void (* pMspDeInitCb)(SRAM_HandleTypeDef *);
}BSP_SRAM_Cb_t;
#endif /* (USE_HAL_SRAM_REGISTER_CALLBACKS == 1) */
```

8.4.7 BSP NOR class APIs

Table 19. BSP NOR class APIs

Function	Argument	description
int32_t BSP_NOR_Init	uint32_t Instance	NOR class Initialization
int32_t BSP_NOR_DeInit	uint32_t Instance	NOR class De-Initialization
HAL_StatusTypeDef MX_NOR_Init	NOR_HandleTypeDef *hNor	NOR peripheral initialization
static HAL_StatusTypeDef NOR_MspInit	NOR_HandleTypeDef *hNor	NOR Msp initialization
static HAL_StatusTypeDef NOR_MspDeInit	NOR_HandleTypeDef *hNor	NOR Msp de-initialization
int32_t BSP_NOR_ReadData	uint32_t Instance uint32_t StartAddress uint16_t *pData uint32_t Size	Read data from NOR flash
int32_t BSP_NOR_WriteData	uint32_t Instance uint32_t StartAddress uint16_t *pData uint32_t Size	Write data to NOR flash
int32_t BSP_NOR_ProgramData	uint32_t Instance uint32_t StartAddress uint16_t *pData uint32_t Size	Program NOR flash
int32_t BSP_NOR_EraseBlock	uint32_t Instance uint32_t BlockAddress	Erase a NOR flash block
int32_t BSP_NOR_EraseChip	uint32_t Instance	Erase whole NOR flash
int32_t BSP_NOR_ReadID	uint32_t Instance NOR_IDTypeDef *pNOR_ID	Read NOR flash ID
int32_t BSP_NOR_ReturnToReadMode	uint32_t Instance	Back to read mode
int32_t BSP_NOR_RegisterMspCallbacks	uint32_t Instance BSP_NOR_Cb_t *Callbacks	Register NOR MSP callbacks
int32_t BSP_NOR_RegisterDefaultMspCallbacks	uint32_t Instance	Register Default NOR MSP callbacks

With:

```
#if (USE_HAL_NOR_REGISTER_CALLBACKS == 1)
typedef struct
{
  void (* pMspInitCb)(NOR_HandleTypeDef *);
  void (* pMspDeInitCb)(NOR_HandleTypeDef *);
}BSP_NOR_Cb_t;
#endif /* (USE_HAL_NOR_REGISTER_CALLBACKS == 1) */
```

8.4.8 BSP QSPI class APIs

Table 20. BSP QSPI class APIs

Function	Argument	description
int32_t BSP_QSPI_Init	uint32_t Instance BSP_QSPI_Init_t *Init	QSPI class Initialization
int32_t BSP_QSPI_DeInit	uint32_t Instance	QSPI class De-Initialization
HAL_StatusTypeDef MX_QSPI_Init	QSPI_HandleTypeDef *hQspi MX_QSPI_Init_t *Init	Configure the QSPI peripheral
static HAL_StatusTypeDef QSPI_MspInit	QSPI_HandleTypeDef *hQspi	QSPI Msp initialization
static HAL_StatusTypeDef QSPI_MspDeInit	QSPI_HandleTypeDef *hQspi	QSPI Msp de-initialization
static int32_t QSPI_ResetMemory	uint32_t Instance	Reset the QSPI memory
static int32_t QSPI_DummyCyclesCfg	uint32_t Instance	Configure the dummy cycles on memory side
static int32_t QSPI_SetODS	uint32_t Instance	Configure the Output driver strength on memory side
int32_t BSP_QSPI_Read	uint32_t Instance uint8_t *pData uint32_t ReadAddr uint32_t Size	Reads an amount of data from the QSPI memory
int32_t BSP_QSPI_Write	uint32_t Instance uint8_t *pData uint32_t WriteAddr, uint32_t Size	Writes an amount of data to the QSPI memory
int32_t BSP_QSPI_EraseBlock	uint32_t Instance uint32_t BlockAddress BSP_QSPI_Erase_t BlockSize	Erases the specified block of the QSPI memory
int32_t BSP_QSPI_EraseChip	uint32_t Instance	Erases the entire QSPI memory
int32_t BSP_QSPI_GetStatus	uint32_t Instance	Get QSPI status

Table 20. BSP QSPI class APIs (continued)

Function	Argument	description
int32_t BSP_QSPI_ConfigFlash	uint32_t Instance BSP_QSPI_Interface_t Mode BSP_QSPI_Transfer_t Rate	Configure QSPI Interface mode and transfer rate
int32_t BSP_QSPI_SelectFlashID	uint32_t Instance uint32_t FlashId	Select the flash ID to be used for memories supporting dual flash mode
int32_t BSP_QSPI_ReadID	uint32_t Instance uint32_t *Id	Get flash ID
int32_t BSP_QSPI_GetInfo	uint32_t Instance BSP_QSPI_Info_t *pInfo	Return the configuration of the QSPI memory
int32_t BSP_QSPI_EnableMemoryMappedMode	uint32_t Instance	Configure the QSPI in memory-mapped mode
int32_t BSP_QSPI_DisableMemoryMappedMode	uint32_t Instance	Exit the QSPI from memory-mapped mode
int32_t BSP_QSPI_RegisterMspCallbacks	uint32_t Instance BSP_QSPI_Cb_t *CallBacks	Register QSPI MSP callbacks
int32_t BSP_QSPI_RegisterDefaultMspCallbacks	uint32_t Instance	Register Default QSPI MSP callbacks

With:

```
typedef struct
{
  BSP_QSPI_Interface_t        InterfaceMode;    /*!<  Current Flash
Interface mode */
  BSP_QSPI_Transfer_t         TransferRate;     /*!<  Current Flash Transfer
mode  */
  BSP_QSPI_DualFlash_t       DualFlashMode;   /*!<  Dual Flash mode
*/
} BSP_QSPI_Init_t;
```

```
typedef struct
{
  uint32_t FlashSize;            /*!< Size of the flash */
  uint32_t EraseSectorSize;      /*!< Size of sectors for the erase
operation */
  uint32_t EraseSectorsNumber;   /*!< Number of sectors for the erase
operation */
  uint32_t ProgPageSize;         /*!< Size of pages for the program
operation */
  uint32_t ProgPagesNumber;      /*!< Number of pages for the program
operation */
} QSPI_Info_t;

typedef enum
{
  QSPI_ACCESS_NONE = 0,          /*!< Instance not initialized,
*/
  QSPI_ACCESS_INDIRECT,          /*!< Instance use indirect mode access
*/
  QSPI_ACCESS_MMP                /*!< Instance use Memory Mapped Mode read
*/
} QSPI_Access_t;
```

```
#define BSP_QSPI_Info_t                NNNXXXX_Info_t
#define BSP_QSPI_Interface_t         NNNXXXX_Interface_t
#define BSP_QSPI_Transfer_t          NNNXXXX_Transfer_t
#define BSP_QSPI_DualFlash_t       NNNXXXX_DualFlash_t
#define BSP_QSPI_Erase_t             NNNXXXX_Erase_t

NNNXXXX: QSPI component
```

```
typedef struct
{
  uint32_t FlashSize;
  uint32_t ClockPrescaler;
  uint32_t SampleShifting;
  uint32_t DualFlashMode;
}MX_QSPI_Init_t;
```

```
#if (USE_HAL_QSPI_REGISTER_CALLBACKS == 1)
typedef struct
{
  void (* pMspInitCb)(QSPI_HandleTypeDef *);
  void (* pMspDeInitCb)(QSPI_HandleTypeDef *);
}BSP_QSPI_Cb_t;
```

```
#endif /* (USE_HAL_QSPI_REGISTER_CALLBACKS == 1) */
```

8.4.9 BSP IO expander class APIs

Table 21. BSP IOExpander class APIs

Function	Argument	description
int32_t BSP_IOEXPANDER_Init	uint32_t Instance uint32_t Functions	Initializes and start the IOExpander component where Functions can be any combination of: IOEXPANDER_IO_MODE IOEXPANDER_IDD_MODE IOEXPANDER_TS_MODE
int32_t BSP_IOEXPANDER_DeInit	uint32_t Instance	De-Initializes the IOExpander component
int32_t BSP_IO_Init	uint32_t Instance BSP_IO_Init_t *Init	IO class Initialization
int32_t BSP_IO_DeInit	uint32_t Instance	IO class De-Initialization
int32_t BSP_IO_WritePin	uint32_t Instance uint32_t Pin	Sets the selected pins state
int32_t BSP_IO_ReadPin	uint32_t Instance uint32_t Pin	Gets the selected pins current state
int32_t BSP_IO_TogglePin	uint32_t Instance uint32_t Pin	Toggles the selected pins state
int32_t BSP_IO_GetIT	uint32_t Instance uint32_t Pins	Gets the selected pins IT status
int32_t BSP_IO_ClearIT	uint32_t Instance uint32_t Pins	Clears all the IO IT pending bits

With:

```
typedef struct
{
  uint32_t Pin;       /*!< Specifies the IO pins to be configured */
  uint32_t Mode;      /*!< Specifies the operating mode for the selected
pins */
  uint32_t Pull;      /*!< Specifies the Pull-up or Pull-Down activation
for the selected pins */
}BSP_IO_Init_t;
```

8.4.10 BSP LCD class APIs

Table 22. BSP LCD class APIs

function	Arguments	description
int32_t BSP_LCD_Init	uint32_t Instance uint32_t Orientation	Initializes the LCD in a given orientation: LCD_ORIENTATION_LANDSCAPE LCD_ORIENTATION_PORTRAIT LCD_ORIENTATION_LANDSCAPE_ROT180 LCD_ORIENTATION_PORTRAIT_ROT180 Note: A non-supported orientation can be omitted
int32_t BSP_LCD_DeInit	uint32_t Instance	De-Initializes the LCD
HAL_StatusTypeDef MX_PPP_Init	PPP_HandleTypeDef *hppp	Peripherals initialization. If more than an IP is used, a MX_PPP_Init function is to be defined for each IP
HAL_StatusTypeDef MX_PPP_ClockConfig	PPP_HandleTypeDef *hppp	LCD clock source configuration
int32_t BSP_LCD_DisplayOn	uint32_t Instance	Set the display On
int32_t BSP_LCD_DisplayOff	uint32_t Instance	Set the display Off
int32_t BSP_LCD_SetBrightness	uint32_t Instance uint32_t Brightness	Set the display brightness
int32_t BSP_LCD_GetBrightness	uint32_t Instance uint32_t *Brightness	Get the display brightness
int32_t BSP_LCD_GetXSize	uint32_t Instance uint32_t *XSize	Get the display width
int32_t BSP_LCD_GetYSize	uint32_t Instance uint32_t *YSize	Get the display height
int32_t BSP_LCD_SetActiveLayer	uint32_t Instance uint32_t LayerIndex	Set the display Active layer. This API is required for the common basic_gui file. If the LCD doesn't support multi-layer, this function must simply set "ActiveLayer" variable stored in LCD context, to 0.
int32_t BSP_LCD_DrawBitmap	uint32_t Instance uint32_t Xpos uint32_t Ypos uint8_t *pBmp	Draw a bitmap image. This API is required for the common basic_gui file.
int32_t BSP_LCD_DrawHLine	uint32_t Instance uint32_t Xpos uint32_t Ypos uint32_t Length uint32_t Color	Draw an horizontal line. This API is required for the common basic_gui file.

Table 22. BSP LCD class APIs (continued)

function	Arguments	description
int32_t BSP_LCD_DrawVLine	uint32_t Instance uint32_t Xpos uint32_t Ypos uint32_t Length uint32_t Color	Draw a vertical line. This API is required for the common basic_gui file.
int32_t BSP_LCD_FillRect	uint32_t Instance uint32_t Xpos uint32_t Ypos uint32_t Width uint32_t Height uint32_t Color	Draw a rectangle. This API is required for the common basic_gui file.
int32_t BSP_LCD_FillRGBRect	uint32_t Instance uint32_t Xpos uint32_t Ypos uint8_t* pData uint32_t Width uint32_t Height	Draw an RGB rectangle. This API is required for the common basic_gui file.
int32_t BSP_LCD_ReadPixel	uint32_t Instance uint32_t Xpos uint32_t Ypos uint32_t *Color	Read a pixel color. This API is required for the common basic_gui file.
int32_t BSP_LCD_WritePixel	uint32_t Instance uint32_t Xpos uint32_t Ypos uint32_t Color	Write a color to a pixel. This API is required for the common basic_gui file.
int32_t BSP_LCD_RegisterDefaultMspCallbacks	uint32_t Instance BSP_LCD_Cb_t *CallBacks	Register default Msp Callbacks
int32_t BSP_LCD_RegisterMspCallbacks	uint32_t Instance	Register user Msp Callbacks

With:

```
#if (USE_HAL_PPP_REGISTER_CALLBACKS == 1)
typedef struct
{
  void (* pMspInitCb)(PPP_HandleTypeDef *);
  void (* pMspDeInitCb)(PPP_HandleTypeDef *);
}BSP_LCD_Cb_t;
#endif /* (USE_HAL_PPP_REGISTER_CALLBACKS == 1) */
```

8.4.11 BSP camera class APIs

Table 23. BSP camera class APIs

Function	Arguments	description
int32_t BSP_CAMERA_Init	uint32_t Instance uint32_t Resolution uint32_t PixelFormat	CAMERA class Initialization with selected resolution and pixel format
int32_t BSP_CAMERA_DeInit	uint32_t Instance	CAMERA class De-Initialization
HAL_StatusTypeDef MX_DCMI_Init	DCMI_HandleTypeDef *hdcmi	CAMERA MX initialization
static HAL_StatusTypeDef CAMERA_MspInit	DCMI_HandleTypeDef *hdcmi	CAMERA Msp initialization
static HAL_StatusTypeDef CAMERA_MspDeInit	DCMI_HandleTypeDef *hdcmi	CAMERA Msp de-initialization
int32_t BSP_CAMERA_Start	uint32_t Instance	Starts the camera capture in continuous or snapshot mode
int32_t BSP_CAMERA_Suspend	uint32_t Instance	Suspend the camera capture
int32_t BSP_CAMERA_Resume	uint32_t Instance	Resume the camera capture
int32_t BSP_CAMERA_Stop	uint32_t Instance	Stop the camera capture
int32_t BSP_CAMERA_GetCapabilities	uint32_t Instance CAMERA_Capabilities_t *Capabilities	Get the Camera capabilities
int32_t BSP_CAMERA_SetResolution	uint32_t Instance uint32_t Resolution	Set the camera resolution: CAMERA_R160x120 CAMERA_R320x240 CAMERA_R480x272 CAMERA_R640x480 CAMERA_R800x480
int32_t BSP_CAMERA_GetResolution	uint32_t Instance uint32_t *Resolution	Get the camera resolution
int32_t BSP_CAMERA_SetPixelFormat	uint32_t Instance uint32_t PixelFormat	Set the camera pixel format: CAMERA_PF_RGB565 CAMERA_PF_RGB888 CAMERA_PF_YUV422
int32_t BSP_CAMERA_GetPixelFormat	uint32_t Instance uint32_t *PixelFormat	Get the camera pixel format
int32_t BSP_CAMERA_SetLightMode	uint32_t Instance uint32_t LightMode	Set the camera Light Mode: CAMERA_LIGHT_AUTO CAMERA_LIGHT_SUNNY CAMERA_LIGHT_OFFICE CAMERA_LIGHT_HOME CAMERA_LIGHT_CLOUDY

Table 23. BSP camera class APIs (continued)

Function	Arguments	description
int32_t BSP_CAMERA_GetLightMode	uint32_t Instance uint32_t *LightMode	Get the camera Light Mode
int32_t BSP_CAMERA_SetColorEffect	uint32_t Instance uint32_t ColorEffect	Set the camera color Effect: CAMERA_COLOR_EFFECT_NONE CAMERA_COLOR_EFFECT_BLUE CAMERA_COLOR_EFFECT_RED CAMERA_COLOR_EFFECT_GREEN CAMERA_COLOR_EFFECT_BW CAMERA_COLOR_EFFECT_SEPIA CAMERA_COLOR_EFFECT_NEGATIVE
int32_t BSP_CAMERA_GetColorEffect	uint32_t Instance uint32_t *ColorEffect	Get the camera color Effect
int32_t BSP_CAMERA_SetBrightness	uint32_t Instance int32_t Brightness	Set the camera brightness: Can be a value between -4 and 4
int32_t BSP_CAMERA_GetBrightness	uint32_t Instance int32_t *Brightness	Get the camera brightness
int32_t BSP_CAMERA_SetSaturation	uint32_t Instance int32_t Saturation	Set the camera saturation: Can be a value between -4 and 4
int32_t BSP_CAMERA_GetSaturation	uint32_t Instance int32_t *Saturation	Get the camera saturation
int32_t BSP_CAMERA_SetContrast	uint32_t Instance int32_t Contrast	Set the camera contrast: Can be a value between -4 and 4
int32_t BSP_CAMERA_GetContrast	uint32_t Instance int32_t *Contrast	Get the camera contrast
int32_t BSP_CAMERA_SetHueDegree	uint32_t Instance int32_t HueDegree	Set the camera hue degree: Can be a value between -6 and 5
int32_t BSP_CAMERA_GetHueDegree	uint32_t Instance int32_t *HueDegree	Get the camera hue degree
int32_t BSP_CAMERA_SetMirrorFlip	uint32_t Instance uint32_t MirrorFlip	Set the camera mirror/flip: CAMERA_MIRRORFLIP_NONE CAMERA_MIRRORFLIP_FLIP CAMERA_MIRRORFLIP_MIRROR
int32_t BSP_CAMERA_GetMirrorFlip	uint32_t Instance uint32_t *MirrorFlip	Get the camera mirror/flip
int32_t BSP_CAMERA_SetZoom	uint32_t Instance uint32_t Zoom	Set the camera zoom: CAMERA_ZOOM_x8 CAMERA_ZOOM_x4 CAMERA_ZOOM_x2 CAMERA_ZOOM_x1
int32_t BSP_CAMERA_GetZoom	uint32_t Instance uint32_t *Zoom	Get the camera zoom

Table 23. BSP camera class APIs (continued)

Function	Arguments	description
int32_t BSP_CAMERA_EnableNightMode	uint32_t Instance	Enable the camera night mode
int32_t BSP_CAMERA_DisableNightMode	uint32_t Instance	Disable the camera night mode
void BSP_CAMERA_DMA_IRQHandler	uint32_t Instance	Handles Camera DMA transfer interrupt request
void BSP_CAMERA_IRQHandler	uint32_t Instance	Handles DCMI interrupt request
void BSP_CAMERA_LineEventCallback	uint32_t Instance	Camera line event callback
void BSP_CAMERA_VsyncEventCallback	uint32_t Instance	Camera vsync event callback
void BSP_CAMERA_FrameEventCallback	uint32_t Instance	Camera frame event callback
int32_t BSP_CAMERA_RegisterMspCallbacks	uint32_t Instance	Register Camera MSP callbacks
int32_t BSP_CAMERA_RegisterDefaultMspCallbacks	uint32_t Instance	Default Camera MSP callbacks

With:

```
typedef struct
{
  uint32_t Resolution;
  uint32_t LightMode;
  uint32_t ColorEffect;
  uint32_t Brightness;
  uint32_t Saturation;
  uint32_t Contrast;
  uint32_t HueDegree;
  uint32_t MirrorFlip;
  uint32_t Zoom;
  uint32_t NightMode;
} CAMERA_Capabilities_t;
```

8.4.12 BSP EEPROM class APIs

Table 24. BSP EEPROM class APIs

Function	Argument	description
int32_t BSP_EEPROM_Init	uint32_t Instance	Initializes the EEPROM
int32_t BSP_EEPROM_DeInit	uint32_t Instance	De-Initializes the EEPROM
int32_t BSP_EEPROM_WritePage	uint32_t Instance uint8_t *pBuffer uint32_t PageNbr	EEPROM write page
int32_t BSP_EEPROM_ReadPage	uint32_t Instance uint8_t *pBuffer uint32_t PageNbr	EEPROM read page
int32_t BSP_EEPROM_WriteBuffer	uint32_t Instance uint8_t *pBuffer uint32_t WriteAddr uint32_t NbrOfBytes	EEPROM write buffer
int32_t BSP_EEPROM_ReadBuffer	uint32_t Instance uint8_t *pBuffer uint32_t ReadAddr uint32_t NbrOfBytes	EEPROM read buffer
int32_t BSP_EEPROM_IsDeviceReady	uint32_t Instance	Check if the EEPROM is ready for write operations

8.4.13 BSP motion class APIs

Table 25. BSP motion class APIs

Function	Argument	description
int32_t BSP_MOTION_SENSOR_Init	uint32_t Instance uint32_t Functions	Motion sensor Initialization. Functions can be any combination of: MOTION_GYRO MOTION_ACCELERO MOTION_MAGNETO
int32_t BSP_MOTION_SENSOR_DeInit	uint32_t Instance	Motion sensor De-Initialization
int32_t BSP_MOTION_SENSOR_GetCapabilities	uint32_t Instance MOTION_SENSOR_Capabilities_t *Capabilities	Get Motion sensor capabilities
int32_t BSP_MOTION_SENSOR_ReadID	uint32_t Instance uint8_t *Id	Read ID of Motion sensor component
int32_t BSP_MOTION_SENSOR_Enable	uint32_t Instance uint32_t Function	Enable the motion sensor function(MOTION_GYRO or MOTION_ACCELERO or MOTION_MAGNETO)

Table 25. BSP motion class APIs (continued)

Function	Argument	description
int32_t BSP_MOTION_SENSOR_Disable	uint32_t Instance uint32_t Function	Disable the motion sensor function(MOTION_GYRO or MOTION_ACCELERO or MOTION_MAGNETO)
int32_t BSP_MOTION_SENSOR_GetAxes	uint32_t Instance uint32_t Function BSP_MOTION_SENSOR_Axes_t *Axes	Get XYZ values from the motion sensor function(MOTION_GYRO or MOTION_ACCELERO or MOTION_MAGNETO)
int32_t BSP_MOTION_SENSOR_GetAxesRaw	uint32_t Instance uint32_t Function BSP_MOTION_SENSOR_AxesRaw_t *Axes	Get XYZ raw values from the motion sensor function(MOTION_GYRO or MOTION_ACCELERO or MOTION_MAGNETO)
int32_t BSP_MOTION_SENSOR_GetSensitivity	uint32_t Instance uint32_t Function float_t *Sensitivity	Get the sensitivity value from the motion sensor function(MOTION_GYRO or MOTION_ACCELERO or MOTION_MAGNETO)
int32_t BSP_MOTION_SENSOR_GetOutputDataRate	uint32_t Instance uint32_t Function float_t *Odr	Get the Output Data Rate value from the motion sensor function(MOTION_GYRO or MOTION_ACCELERO or MOTION_MAGNETO)
int32_t BSP_MOTION_SENSOR_SetOutputDataRate	uint32_t Instance uint32_t Function float_t Odr	Set the Output Data Rate value of the motion sensor function(MOTION_GYRO or MOTION_ACCELERO or MOTION_MAGNETO)
int32_t BSP_MOTION_SENSOR_GetFullScale	uint32_t Instance uint32_t Function int32_t *Fullscale	Get Full Scale value from the motion sensor function(MOTION_GYRO or MOTION_ACCELERO or MOTION_MAGNETO)
int32_t BSP_MOTION_SENSOR_SetFullScale	uint32_t Instance uint32_t Function int32_t Fullscale	Set Full Scale value of the motion sensor function(MOTION_GYRO or MOTION_ACCELERO or MOTION_MAGNETO)

With:

```
typedef struct
{
  int32_t x;
  int32_t y;
  int32_t z;
} BSP_MOTION_SENSOR_Axes_t;

typedef struct
{
  int16_t x;
  int16_t y;
  int16_t z;
} BSP_MOTION_SENSOR_AxesRaw_t;

/* Motion Sensor instance Info */
typedef struct
{
  uint8_t  Acc;
  uint8_t  Gyro;
  uint8_t  Magneto;
  uint8_t  LowPower;
  uint32_t GyroMaxFS;
  uint32_t AccMaxFS;
  uint32_t MagMaxFS;
  float_t  GyroMaxOdr;
  float_t  AccMaxOdr;
  float_t  MagMaxOdr;
} MOTION_SENSOR_Capabilities_t;
```

8.4.14 BSP environmental class APIs

Table 26. BSP environmental class APIs

Function	Argument	description
int32_t BSP_ENV_SENSOR_Init	uint32_t Instance uint32_t Functions	Environmental sensor Initialization. Functions can be any combination of: ENV_TEMPERATURE ENV_PRESSURE ENV_HUMIDITY
int32_t BSP_ENV_SENSOR_DeInit	uint32_t Instance	Environmental sensor De-Initialization
int32_t BSP_ENV_SENSOR_GetCapabilities	uint32_t Instance ENV_SENSOR_Capabilities_t *Capabilities	Get Environmental sensor capabilities

Table 26. BSP environmental class APIs (continued)

Function	Argument	description
int32_t BSP_ENV_SENSOR_ReadID	uint32_t Instance uint8_t *Id	Read ID of Environmental sensor component
int32_t BSP_ENV_SENSOR_Enable	uint32_t Instance uint32_t Function	Enable the Environmental sensor function(ENV_TEMPERATURE or ENV_PRESSURE or ENV_HUMIDITY)
int32_t BSP_ENV_SENSOR_Disable	uint32_t Instance uint32_t Function	Disable the Environmental sensor function(ENV_TEMPERATURE or ENV_PRESSURE or ENV_HUMIDITY)
int32_t BSP_ENV_SENSOR_GetValue	uint32_t Instance uint32_t Function float_t *Value	Get XYZ values from the Environmental sensor function(ENV_TEMPERATURE or ENV_PRESSURE or ENV_HUMIDITY)
int32_t BSP_ENV_SENSOR_GetOutputDataRate	uint32_t Instance uint32_t Function float_t *Odr	Get the Output Data Rate value from the Environmental sensor function(ENV_TEMPERATURE or ENV_PRESSURE or ENV_HUMIDITY)
int32_t BSP_ENV_SENSOR_SetOutputDataRate	uint32_t Instance uint32_t Function float_t Odr	Set the Output Data Rate value of the Environmental sensor function(ENV_TEMPERATURE or ENV_PRESSURE or ENV_HUMIDITY)

With:

```
typedef struct
{
  uint8_t Temperature;
  uint8_t Pressure;
  uint8_t Humidity;
  uint8_t LowPower;
  float_t HumMaxOdr;
  float_t TempMaxOdr;
  float_t PressMaxOdr;
} ENV_SENSOR_Capabilities_t;
```

8.4.15 BSP OSPI class APIs

BSP OSPI NOR APIs

Table 27. BSP OSPI NOR class APIs

Function	Argument	description
int32_t BSP_OSPI_NOR_Init	uint32_t Instance BSP_OSPI_NOR_Init_t *Init	OSPI NOR class Initialization
int32_t BSP_OSPI_NOR_DeInit	uint32_t Instance	OSPI NOR class De-Initialization
HAL_StatusTypeDef MX_OSPI_NOR_Init	OSPI_HandleTypeDef *hOspi MX_OSPI_Init_t *Init	Configure the OSPI peripheral
static HAL_StatusTypeDef OSPI_NOR_MspInit	OSPI_HandleTypeDef *hOspi	OSPI NOR Msp initialization
static HAL_StatusTypeDef OSPI_NOR_MspDeInit	OSPI_HandleTypeDef *hOspi	OSPI NOR Msp de-initialization
int32_t BSP_OSPI_NOR_Read	uint32_t Instance uint8_t *pData uint32_t ReadAddr uint32_t Size	Reads an amount of data from the OSPI NOR memory
int32_t BSP_OSPI_NOR_Write	uint32_t Instance uint8_t *pData uint32_t WriteAddr, uint32_t Size	Writes an amount of data to the OSPI NOR memory
int32_t BSP_OSPI_NOR_Erase_Block	uint32_t Instance uint32_t BlockAddress BSP_OSPI_NOR_Erase_t BlockSize	Erases the specified block of the OSPI NOR memory
int32_t BSP_OSPI_NOR_Erase_Chip	uint32_t Instance	Erases the entire OSPI NOR memory
int32_t BSP_OSPI_NOR_GetStatus	uint32_t Instance	Get OSPI NOR status
int32_t BSP_OSPI_NOR_ConfigFlash	uint32_t Instance BSP_OSPI_NOR_Interface_t Mode BSP_OSPI_NOR_Transfer_t Rate	Configure OSPI NOR Interface mode and transfer rate
int32_t BSP_QSPI_ReadID	uint32_t Instance uint32_t *Id	Get flash ID
int32_t BSP_QSPI_GetInfo	uint32_t Instance BSP_OSPI_NOR_Info_t *pInfo	Return the configuration of the OSPI NOR memory
int32_t BSP_OSPI_NOR_EnableMemoryMappedMode	uint32_t Instance	Configure the OSPI NOR in memory-mapped mode

Table 27. BSP OSPI NOR class APIs (continued)

Function	Argument	description
int32_t BSP_OSPI_NOR_DisableMemoryMappedMode	uint32_t Instance	Exit the OSPI NOR from memory-mapped mode
int32_t BSP_OSPI_NOR_RegisterMspCallbacks	uint32_t Instance BSP_OSPI_Cb_t *CallBacks	Register OSPI NOR MSP callbacks
int32_t BSP_OSPI_NOR_RegisterDefaultMspCallbacks	uint32_t Instance	Register Default OSPI NOR MSP callbacks

With:

```
typedef struct
{
  BSP_OSPI_NOR_Interface_t        InterfaceMode;
  BSP_OSPI_NOR_Transfer_t         TransferRate;
} BSP_OSPI_NOR_Init_t;
```

```
typedef struct
{
  uint32_t MemorySize;
  uint32_t ClockPrescaler;
  uint32_t SampleShifting;
  uint32_t TransferRate;
} MX_OSPI_InitTypeDef;
```

```
#if (USE_HAL_OSPI_REGISTER_CALLBACKS == 1)
typedef struct
{
  void (* pMspInitCb)(OSPI_HandleTypeDef *);
  void (* pMspDeInitCb)(OSPI_HandleTypeDef *);
}BSP_QSPI_Cb_t;
#endif /* (USE_HAL_OSPI_REGISTER_CALLBACKS == 1) */
```

```
#define BSP_OSPI_NOR_Info_t               NNNXXXX_Info_t
#define BSP_OSPI_NOR_Interface_t         NNNXXXX_Interface_t
#define BSP_OSPI_NOR_Transfer_t          NNNXXXX_Transfer_t
#define BSP_OSPI_NOR_Erase_t               NNNXXXX_Erase_t

NNNXXXX: OSPI component
```

Note: The following advanced APIs are to be implemented if the OSPI support the erase suspend/resume or the power down enter/leave features:

Table 28. Advanced APIS for OSPI support additional features

int32_t BSP_OSPI_NOR_SuspendErase	uint32_t Instance	Suspend the OSPI NOR memory
int32_t BSP_OSPI_NOR_ResumeErase	uint32_t Instance	Resume the erase of OSPI NOR memory
int32_t BSP_OSPI_NOR_EnterDeepPowerDown	uint32_t Instance	Enter OSPI NOR memory in Deep Power Down
int32_t BSP_OSPI_NOR_LeaveDeepPowerDown	uint32_t Instance	Exit OSPI NOR memory from Deep Power Down

BSP OSPI RAM class APIs

Table 29. BSP OSPI RAM class APIs

Function	Argument	description
int32_t BSP_OSPI_RAM_Init	uint32_t Instance BSP_OSPI_RAM_Init_t *Init	OSPI RAM class Initialization:
int32_t BSP_OSPI_RAM_DeInit	uint32_t Instance	OSPI RAM class De-Initialization
HAL_StatusTypeDef MX_OSPI_RAM_Init	OSPI_HandleTypeDef *hOspi MX_OSPI_Init_t *Init	Configure the OSPI peripheral
static HAL_StatusTypeDef OSPI_RAM_MspInit	OSPI_HandleTypeDef *hOspi	OSPI RAM Msp initialization
static HAL_StatusTypeDef OSPI_RAM_MspDeInit	OSPI_HandleTypeDef *hOspi	OSPI RAM Msp de-initialization
int32_t BSP_OSPI_RAM_Read	uint32_t Instance uint8_t *pData uint32_t ReadAddr uint32_t Size	Reads an amount of data from the OSPI RAM memory in polling mode
int32_t BSP_OSPI_RAM_Read_DMA	uint32_t Instance uint8_t *pData uint32_t ReadAddr uint32_t Size	Reads an amount of data from the OSPI RAM memory in DMA mode
int32_t BSP_OSPI_RAM_Write	uint32_t Instance uint8_t *pData uint32_t WriteAddr, uint32_t Size	Writes an amount of data to the OSPI RAM memory in polling mode
int32_t BSP_OSPI_RAM_Write_DMA	uint32_t Instance uint8_t *pData uint32_t WriteAddr, uint32_t Size	Writes an amount of data to the OSPI RAM memory in DMA mode
int32_t BSP_OSPI_RAM_ConfigHyperRAM	uint32_t Instance BSP_OSPI_RAM_Latency_t Latency BSP_OSPI_RAM_BurstType_t BurstType BSP_OSPI_RAM_BurstLength_t BurstLength	Set HyperRAM to desired configuration
int32_t BSP_OSPI_RAM_EnableMemoryMappedMode	uint32_t Instance	Configure the OSPI RAM in memory-mapped mode
int32_t BSP_OSPI_RAM_DisableMemoryMappedMode	uint32_t Instance	Exit the OSPI RAM from memory-mapped mode
void BSP_OSPI_RAM_DMA_IRQHandler	uint32_t Instance	Handles OctoSPI HyperRAM DMA transfer interrupt request
void BSP_OSPI_RAM_IRQHandler	uint32_t Instance	Handles OctoSPI HyperRAM interrupt request

Table 29. BSP OSPI RAM class APIs (continued)

Function	Argument	description
int32_t BSP_OSPI_RAM_RegisterMspCallbacks	uint32_t Instance BSP_OSPI_Cb_t *CallBacks	Register OSPI RAM MSP callbacks
int32_t BSP_OSPI_RAM_RegisterDefaultMspCallbacks	uint32_t Instance	Register Default OSPI RAM MSP callbacks

With:

```
typedef struct
{
  BSP_OSPI_RAM_Latency_t      LatencyType;    /*!< Current HyperRAM Latency
Type */
  BSP_OSPI_RAM_BurstType_t    BurstType;      /*!< Current HyperRAM Burst
Type   */
  BSP_OSPI_RAM_BurstLength_t  BurstLength;    /*!< Current HyperRAM Burst
Length */
} BSP_OSPI_RAM_Init_t;
```

```
typedef enum
{
  BSP_OSPI_RAM_VARIABLE_LATENCY = HAL_OSPI_VARIABLE_LATENCY,
  BSP_OSPI_RAM_FIXED_LATENCY    = HAL_OSPI_FIXED_LATENCY
} BSP_OSPI_RAM_Latency_t;

typedef enum
{
  BSP_OSPI_RAM_HYBRID_BURST = 0,
  BSP_OSPI_RAM_LINEAR_BURST
} BSP_OSPI_RAM_BurstType_t;

#define BSP_OSPI_RAM_BurstLength_t NNNXXXX_BurstLength_t
NNNXXXX: OSPI hyperRAM component
```

8.5 BSP class driver context

BSP driver resources (process variables) must always be stored in a global BSP driver context structure. Note that the class_ppp handle of the BSP class driver main peripheral must always be defined outside the context to export them when needed by the x*xxx_it.c* file to handle IRQ or by the application if needed to change peripheral parameters.

An example of context structure is defined as below (AUDIO IN):

```
typedef struct
{
  uint32_t Instance;      /*Audio IN instance*/
  uint32_t Device;        /*Audio IN device to be used*/
  uint32_t SampleRate;    /*Audio IN Sample rate*/
  uint32_t BitsPerSample; /*Audio IN Sample resolution*/
  uint32_t ChannelsNbr;   /*Audio IN number of channel*/
  uint16_t *pBuff;        /*Audio IN record buffer*/
  uint8_t  **pMultiBuff;  /*Audio IN multi-buffer*/
  uint32_t Size;          /*Audio IN record buffer size*/
  uint32_t Volume;        /*Audio IN volume*/
  uint32_t State;         /*Audio IN State*/
  uint32_t IsMultiBuff;         /*Audio IN multi-buffer usage*/
  uint32_t IsMspCallbacksValid; /*Is Msp Callbacks registered*/
} AUDIO_IN_Ctx_t;
```

8.6 BSP class driver inter dependency

In some cases, one or more BSP drivers may be used by several other BSP class drivers, this typically the case of the IO expander, the BSP IO may be used by the BSP TS, IDD and IO itself.

The BSP class drivers may need to call the BSP_IO_Init () several times internally. To prevent multi initialization the BSP_IO_Init () must manage such multi call by using its internal Is_Initialized variable to check whether or not the IO expander is already called.

The BSP_IO_Init must be defined as below:

```
int32_t BSP_IO_Init (uint32_t Instance)
{
  int32_t ret = BSP_ERROR_UNKNOWN_FAILURE;
  static uint8_t IO_IsInitialized = 0;

  if (IO_IsInitialized == 0)
  {
       IO_IsInitialized = 1;
       (…)
       ret = BSP_ERROR_NONE;
      }
    }
  }
  else
  {
     ret = BSP_ALREADY_INITIALIZED;
  }

  return ret;
}
```

8.7 Using driver structure

Typically the class driver structure, defined under /common, is used when we need to use only the generic functions in a component driver in standalone mode or when a BSP class driver may support several component references offering the same services. Typical use of the class driver structure is as follows:

```
NNNXXXX_Object_t ComponentObj;
NNNXXXX_Drv_t    ComponentDrv;

void BSP_CLASS_Init (BSP_CLASS_Init_t *ClassInit)
{
  NNNXXXX_IO_t            IO;
  NNNXXXX_Init_t             Init;
  (…)
  /* Configure the component */
  IO.DevAddr    = DevAddr;
  IO.Init       = BUS#N_Init;
  IO.ReadReg    = BUS#N_ReadReg;
  IO.WriteReg   = BUS#N_WriteReg;
  IO.GetTick    = BSP_GetTick;
  NNNXXXX_RegisterBusIO (&ComponentObj, &IO);
    (…)
  ComponentDrv.Init (&ComponentObj, Init);
    (…)
}
```

Or with dynamic class driver structure assignment using the ReadID function or the component use define in the global BSP configuration files.

```
NNNXXXX_A_Object_t ComponentObjA;
CLASS_DrvTypeDef      *Class_Drv;
void                  *Class_CompObj;
int32_t BSP_CLASS_Init (BSP_CLASS_Init_t *ClassInit)
{
    (…)
    if (NNNXXXX_A_Probe((void *)&CodecAddress, ClassInit))
    {
       return BSP_ERROR_UNKNOWN_COMPONENT;
    }
    (…)
}
static int32_t NNNXXXX_Probe(void *Ctx, BSP_CLASS_Init_t *ClassInit)
{
  static NNNXXXX_Object_t   NNNXXXX_Obj;
  NNNXXXX_IO_t           IO;
  NNNXXXX_Init_t            Init;

  /* Configure the component */
  IO.DevAddr    = DevAddr;
  IO.Init       = BUS#N_Init;
  IO.ReadReg    = BUS#N_ReadReg;
  IO.WriteReg   = BUS#N_WriteReg;
  IO.GetTick    = BSP_GetTick;
  NNNXXXX_RegisterBusIO (&NNNXXXX_Obj, &IO);

  if ((NNNXXXX_ReadID (&NNNXXXX_Obj) & NNNXXXX_ID_MASK) == NNNXXXX_ID)
  {
    (…)
    NNNXXXX_Init (&NNNXXXX_Obj, &Init);

    Class_Drv = (CLASS_Drv_t *) &NNNXXXX_drv;
    Class_CompObj = &NNNXXXX_Obj;
  return BSP_ERROR_NONE;
  }
  return BSP_ERROR_UNKNOWN_COMPONENT;
}
```

9 BSP IRQ handlers

9.1 Generic rules

The IRQ handlers associated with the BSP drivers must be defined internally in the BSP driver, it is not recommended to use the BSP peripherals handles in the *stm32yyxxxx_it.c* using HAL APIs except for raw memory access (SDRAM, SRAM…), that write and read operation require to call HAL services thus need for handles. The BSP IRQ handler naming is following this rule:

```
void {BOARD_PREFIX}_{CATEGORY/CLASS}_{Mode}_IRQHandler (#INSTANCE#,
#SUB_INSTANCE)
```

And then called in the *stm32yyxxxx_it.c* file, as follows:

```
void #IRQLINE#_IRQHandler(void)
{
{BOARD_PREFIX}_{CATEGORY/CLASS}_{Mode}_IRQHandler (#INSTANCE#,
#SUB_INSTANCE#);
}
```

The table below provides some examples of the above symbols:

Table 30. Examples of symbols

BOARD_PREFIX	CLASS	CATEGORY (Common Class Driver)	MODE	INSTANCE	SUBINSTANCE
ST native board: BSP	CLASSNAME, ex: AUDIO_OUT AUDIO_IN LCD Etc…	PB, JOY, POT, COM, Etc…	None: (Empty)	0….N COM1…COMn JOY1….JOYn	XXX_MIC1 XXX_MIC2 XXX_MIC3 XXX_MIC4
Shield: SHIELDNAME					
User board: user board name			DMA Model: DMA IT model: IT		

Some examples are listed below.

- Standard board common BSP driver

```
void EXTI9_5_IRQHandler(void)
{
  BSP_PB_IRQHandler (BUTTON_USER);
}
void EXTI9_5_IRQHandler(void)
{
  BSP_JOY_IRQHandler (JOY1, JOY_ALL);
}
```

- Standard board AUDIO BSP driver

```
void DMA2_Stream0_IRQHandler(void)
{
BSP_AUDIO_IN_IRQHandler (0, AUDIO_IN_ANALOG_MIC);
}

void DMA2_Stream1_IRQHandler(void)
{
BSP_AUDIO_IN_IRQHandler (0,  AUDIO_IN_DIGITAL_MIC1);
}
```

- Shield common BSP driver

```
void EXTI9_5_IRQHandler(void)
{
SxRadioBoard_IRQHandler (RADIO_DIO0);
}
```

9.2 BSP IRQ Handlers implementation

As described in the previous section, the BSP IRQ handlers are defined as follows:

```
void {BOARD_PREFIX}_{CATEGORY/CLASS}_IRQHandler (#INSTANCE#,
#SUB_INSTANCE)
```

The BSP IRQ handler function is platform and features agnostic, however the internal implementation depends on the hardware configuration and the BSP IRQ lines configuration.

Typically, the BSP IRQ handler can be managed in two ways following the hardware configuration:

The category/Class sub-instances use each an IRQ line. Example, the different buttons uses different IRQ lines.

```
void BSP_PB_IRQHandler (Button_TypeDef Button)
{
HAL_EXTI_IRQHandler (&hpb_exti [Button]);
}
```

And in the BSP Button IRQ Handler is called in the *stm32yyxxx_it.c* file as follows:

```
void EXTIa_IRQHandler(void)
{
BSP_PB_IRQHandler (BUTTON_NAME#1);
}
void EXTIb_IRQHandler(void)
{
BSP_PB_IRQHandler (BUTTON_NAME#2);
}
```

The same IRQ line manages a category/Class sub-instances. Example, the different Joystick keys share the same IRQ line.

```
void BSP_JOY_IRQHandler (JOY_TypeDef JOY, uint32_t JoyPins)
{
UNUSED(JoyPins);
HAL_EXTI_IRQHandler (&hjoy_Exti[JOY]);
}
```

And in the BSP Button IRQ Handler is called in the *stm32yyxxx_it.c* file as follows:

```
void EXTIa_IRQHandler(void)
{
BSP_JOY_IRQHandler (JOYi, JOY_ALL);
}
```

10 BSP error management

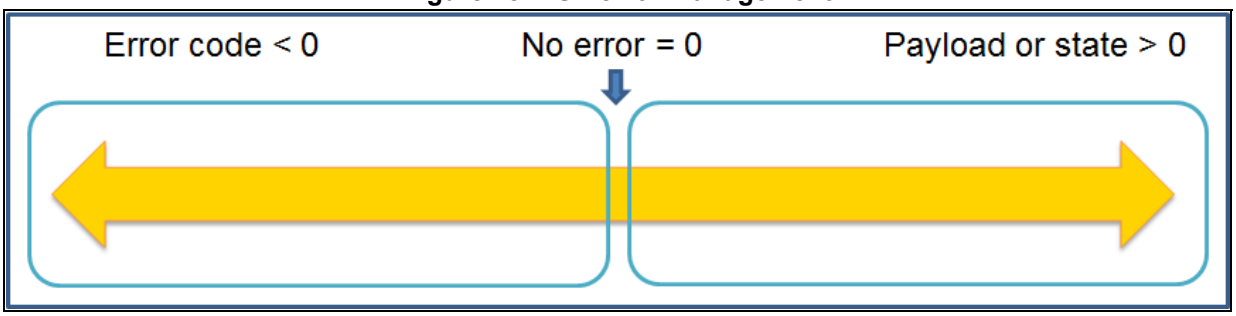
The common BSP drivers' module must return specific error to their functions. Each caller to one or more services from this module including the application must make a check on the returned values before using the associated services.

The class drivers are based on dynamic component driver assignment, at user file level, the application must check on the return value of the BSP_CLASS_Init before using the class driver services and inside the BSP services, the functions pointers validity must be always checked and return an error when the function pointer is not valid.

The main rule for the error returned values is as follows:

- Negative values: error codes
- Zero: no error
- Positive values: state, information

Figure 18. BSP error management



Note: In general, returned value by BSP Driver and component functions must always be a pointer, for example:

```
int32_t BSP_CLASS_Function (uint32_t Instance, uint32_t * value);
or
int32_t BSP_CLASS_Function (uint32_t Instance, BSP_ITEM_t * sValue);
```

Exception is made for status check function and Get function type with positive values, for example:

```
int32_t BSP_I2C1_IsReady (uint16_t DevAddr, uint32_t Trials)
{
int32_t ret = BSP_ERROR_NONE;
if(HAL_I2C_IsDeviceReady(&hbus_i2c1, DevAddr, Trials,
BUS_I2C1_POLL_TIMEOUT) != HAL_OK)
{
ret = BSP_ERROR_BUSY;
}
return ret;
}
```

10.1 Component drivers

All the functions of the components drivers must return a component specific error code (int32_t) defined following the scheme above.

For components drivers returned values such as data transfer length or data value must be returned as positive values.

10.2 BSP common drivers

The class drivers, bus drivers and common drivers use common error code defined in the *boardname_errno.h* file and defined as follows:

```
/* Common Error codes */
#define BSP_ERROR_NONE 0
#define BSP_ERROR_NO_INIT -1
#define BSP_ERROR_WRONG_PARAM -2
#define BSP_ERROR_BUSY -3
#define BSP_ERROR_PERIPH_FAILURE -4
#define BSP_ERROR_COMPONENT_FAILURE -5
#define BSP_ERROR_UNKNOWN_FAILURE -6
#define BSP_ERROR_UNKNOWN_COMPONENT -7
#define BSP_ERROR_BUS_FAILURE -8
#define BSP_ERROR_CLOCK_FAILURE -9
#define BSP_ERROR_MSP_FAILURE -10
#define BSP_ERROR_FEATURE_NOT_SUPPORTED -11
```

10.3 BSP class drivers

The specific class driver errors are also defined in the *boardname_errno.h* file as follows:

```
/* BSP QSPI error codes */
#define BSP_ERROR_QSPI_ASSIGN_FAILURE -20
#define BSP_ERROR_QSPI_SETUP_FAILURE -21
#define BSP_ERROR_QSPI_MMP_LOCK_FAILURE -22
#define BSP_ERROR_QSPI_MMP_UNLOCK_FAILURE -23
/* BSP TS error code */
#define BSP_ERROR_TS_TOUCH_NOT_DETECTED -30
```

10.4 BSP bus drivers

The specific bus drivers' errors are also defined in the boardname_errno.h file as follows:

```
/* BSP BUS error code */
#define BSP_ERROR_BUS_TRANSACTION_FAILURE -100
#define BSP_ERROR_BUS_ARBITRATION_LOSS -101
#define BSP_ERROR_BUS_ACKNOWLEDGE_FAILURE -102
#define BSP_ERROR_BUS_PROTOCOL_FAILURE -103
#define BSP_ERROR_BUS_MODE_FAULT -104
#define BSP_ERROR_BUS_FRAME_ERROR -105
#define BSP_ERROR_BUS_CRC_ERROR -106
#define BSP_ERROR_BUS_DMA_FAILURE -107
```

Revision history

Table 31. Document revision history

Date	Document revision	Changes
4-Jun-2018	1	Initial version
19-Jun-2019	2	Updated: – error management section: adding error codes for bus and classes – BSP IRQ handlers section: adding generic rules and usage implementation – code examples according to BSP naming rules Added: – chapters for STM32Cube and BSP naming rules – OSPI class APIs list – details on APIs parameters: Enumerations and structures – Bus specific services and customization sections

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.

© 2019 STMicroelectronics – All rights reserved