

Description of STM32L4/L4+ HAL and low-layer drivers

Introduction

STMCube™ is STMicroelectronics's original initiative to ease developers' life by reducing development efforts, time and cost. STM32Cube covers the STM32 portfolio.

STM32Cube Version 1.x includes:

- The STM32CubeMX, a graphical software configuration tool that allows generating C initialization code using graphical wizards.
- A comprehensive embedded software platform, delivered per series (such as STM32CubeL4 for STM32L4 series and STM32L4+ series)
 - The STM32Cube Hardware Abstraction Layer (HAL), an STM32 abstraction layer embedded software ensuring maximized portability across the STM32 portfolio. The HAL is available for all peripherals.
 - The low-layer APIs (LL) offering a fast light-weight expert-oriented layer which is closer to the hardware than the HAL. The LL APIs are available only for a set of peripherals.
 - A consistent set of middleware components such as RTOS, USB, Graphics
 - All embedded software utilities coming with a full set of examples.

The HAL driver layer provides a generic multi-instance simple set of APIs (application programming interfaces) to interact with the upper layer (application, libraries and stacks).

The HAL driver APIs are split into two categories: generic APIs which provide common and generic functions for all the STM32 series and extension APIs which include specific and customized functions for a given line or part number. The HAL drivers include a complete set of ready-to-use APIs which simplify the user application implementation. As an example, the communication peripherals contain APIs to initialize and configure the peripheral, manage data transfers in polling mode, handle interrupts or DMA, and manage communication errors.

The HAL drivers are feature-oriented instead of IP-oriented. As an example, the timer APIs are split into several categories following the IP functions: basic timer, capture, pulse width modulation (PWM), etc..

The HAL driver layer implements run-time failure detection by checking the input values of all functions. Such dynamic checking contributes to enhance the firmware robustness. Run-time detection is also suitable for user application development and debugging.

The LL drivers offer hardware services based on the available features of the STM32 peripherals. These services reflect exactly the hardware capabilities and provide atomic operations that must be called following the programming model described in the product line reference manual. As a result, the LL services are not based on standalone processes and do not require any additional memory resources to save their states, counter or data pointers: all operations are performed by changing the associated peripheral registers content. Contrary to the HAL, the LL APIs are not provided for peripherals for which optimized access is not a key feature, or for those requiring heavy software configuration and/or complex upper level stack (such as FSMC, USB, SDMMC).

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

- The HAL offers high-level and feature-oriented APIs, with a high-portability level. They hide the MCU and peripheral complexity to end-user.
- The LL offers low-level APIs at registers level, with better optimization but less portability. They require deep knowledge of the MCU and peripherals specifications.

The source code of HAL and LL drivers is developed in Strict ANSI-C which makes it independent from the development tools. It is checked with CodeSonar™ static analysis tool. It is fully documented and is MISRA-C 2004 compliant.



Contents

1	Acronyms and definitions.....	34
2	Overview of HAL drivers.....	36
2.1	HAL and user-application files.....	36
2.1.1	HAL driver files	36
2.1.2	User-application files	37
2.2	HAL data structures	39
2.2.1	Peripheral handle structures	39
2.2.2	Initialization and configuration structure	40
2.2.3	Specific process structures	40
2.3	API classification	41
2.4	Devices supported by HAL drivers	42
2.5	HAL driver rules	47
2.5.1	HAL API naming rules	47
2.5.2	HAL general naming rules.....	48
2.5.3	HAL interrupt handler and callback functions.....	50
2.6	HAL generic APIs.....	50
2.7	HAL extension APIs	52
2.7.1	HAL extension model overview	52
2.7.2	HAL extension model cases	52
2.8	File inclusion model.....	54
2.9	HAL common resources.....	55
2.10	HAL configuration.....	55
2.11	HAL system peripheral handling	56
2.11.1	Clock.....	56
2.11.2	GPIOs.....	57
2.11.3	Cortex NVIC and SysTick timer.....	59
2.11.4	PWR	59
2.11.5	EXTI.....	59
2.11.6	DMA.....	61
2.12	How to use HAL drivers	62
2.12.1	HAL usage models	62
2.12.2	HAL initialization	63
2.12.3	HAL IO operation process	65
2.12.4	Timeout and error management.....	69

3	Overview of low-layer drivers.....	73
3.1	Low-layer files	73
3.2	Overview of low-layer APIs and naming rules	75
3.2.1	Peripheral initialization functions	75
3.2.2	Peripheral register-level configuration functions	79
4	Cohabiting of HAL and LL	81
4.1	Low-layer driver used in standalone mode.....	81
4.2	Mixed use of low-layer APIs and HAL drivers	81
5	HAL System Driver	82
5.1	HAL Firmware driver API description	82
5.1.1	How to use this driver	82
5.1.2	Initialization and de-initialization functions	82
5.1.3	HAL Control functions.....	82
5.1.4	HAL Debug functions.....	83
5.1.5	HAL SYSCFG configuration functions.....	83
5.1.6	Detailed description of functions	84
5.2	HAL Firmware driver defines.....	90
5.2.1	HAL.....	90
6	HAL ADC Generic Driver.....	97
6.1	ADC Firmware driver registers structures	97
6.1.1	ADC_OversamplingTypeDef	97
6.1.2	ADC_InitTypeDef.....	97
6.1.3	ADC_ChannelConfTypeDef	100
6.1.4	ADC_AnalogWDGConfTypeDef.....	101
6.1.5	ADC_InjectionConfigTypeDef	102
6.1.6	ADC_HandleTypeDef	102
6.2	ADC Firmware driver API description.....	102
6.2.1	ADC peripheral features	102
6.2.2	How to use this driver	103
6.2.3	Peripheral Control functions	105
6.2.4	Peripheral state and errors functions	105
6.2.5	Detailed description of functions	105
6.3	ADC Firmware driver defines	114
6.3.1	ADC	114
7	HAL ADC Extension Driver	139
7.1	ADCEX Firmware driver registers structures	139

7.1.1	ADC_InjOversamplingTypeDef	139
7.1.2	ADC_InjectionConfTypeDef	139
7.2	ADCEX Firmware driver API description	141
7.2.1	IO operation functions	141
7.2.2	Peripheral Control functions	142
7.2.3	Detailed description of functions	142
7.3	ADCEX Firmware driver defines	149
7.3.1	ADCEX	149
8	HAL CAN Generic Driver	151
8.1	CAN Firmware driver registers structures	151
8.1.1	CAN_InitTypeDef	151
8.1.2	CAN_FilterConfTypeDef	152
8.1.3	CanTxMsgTypeDef	153
8.1.4	CanRxMsgTypeDef	153
8.1.5	CAN_HandleTypeDef	154
8.2	CAN Firmware driver API description	154
8.2.1	How to use this driver	154
8.2.2	Initialization and de-initialization functions	155
8.2.3	IO operation functions	156
8.2.4	Peripheral State and Error functions	156
8.2.5	Detailed description of functions	156
8.3	CAN Firmware driver defines	160
8.3.1	CAN	160
9	HAL CORTEX Generic Driver	168
9.1	CORTEX Firmware driver registers structures	168
9.1.1	MPU_Region_InitTypeDef	168
9.2	CORTEX Firmware driver API description	169
9.2.1	How to use this driver	169
9.2.2	Initialization and Configuration functions	169
9.2.3	Peripheral Control functions	170
9.2.4	Detailed description of functions	170
9.3	CORTEX Firmware driver defines	175
9.3.1	CORTEX	175
10	HAL CRC Generic Driver	178
10.1	CRC Firmware driver registers structures	178
10.1.1	CRC_InitTypeDef	178
10.1.2	CRC_HandleTypeDef	179

10.2	CRC Firmware driver API description	179
10.2.1	How to use this driver	179
10.2.2	Initialization and de-initialization functions	179
10.2.3	Peripheral Control functions	180
10.2.4	Peripheral State functions	180
10.2.5	Detailed description of functions	180
10.3	CRC Firmware driver defines	182
10.3.1	CRC	182
11	HAL CRC Extension Driver	185
11.1	CRCEX Firmware driver API description	185
11.1.1	How to use this driver	185
11.1.2	Extended configuration functions	185
11.1.3	Detailed description of functions	185
11.2	CRCEX Firmware driver defines	186
11.2.1	CRCEX	186
12	HAL CRYPT Generic Driver	188
12.1	CRYPT Firmware driver registers structures	188
12.1.1	CRYPT_InitTypeDef	188
12.1.2	CRYPT_HandleTypeDef	188
12.2	CRYPT Firmware driver API description	189
12.2.1	How to use this driver	189
12.2.2	Initialization and deinitialization functions	190
12.2.3	AES processing functions	191
12.2.4	Callback functions	192
12.2.5	AES IRQ handler management	192
12.2.6	Peripheral State functions	192
12.2.7	Detailed description of functions	193
12.3	CRYPT Firmware driver defines	202
12.3.1	CRYPT	202
13	HAL CRYPT Extension Driver	208
13.1	CRYPEx Firmware driver API description	208
13.1.1	Extended callback functions	208
13.1.2	AES extended processing functions	208
13.1.3	AES extended suspension and resumption functions	208
13.1.4	Detailed description of functions	209
14	HAL DAC Generic Driver	217

14.1	DAC Firmware driver registers structures	217
14.1.1	DAC_HandleTypeDef	217
14.1.2	DAC_SampleAndHoldConfTypeDef	217
14.1.3	DAC_ChannelConfTypeDef	218
14.2	DAC Firmware driver API description	218
14.2.1	DAC Peripheral features	218
14.2.2	How to use this driver	221
14.2.3	Initialization and de-initialization functions	222
14.2.4	IO operation functions	222
14.2.5	Peripheral Control functions	222
14.2.6	Peripheral State and Errors functions	223
14.2.7	Detailed description of functions	223
14.3	DAC Firmware driver defines	227
14.3.1	DAC	227
15	HAL DAC Extension Driver	233
15.1	DACEx Firmware driver API description	233
15.1.1	How to use this driver	233
15.1.2	Extended features functions	233
15.1.3	Peripheral Control functions	233
15.1.4	Detailed description of functions	234
15.2	DACEx Firmware driver defines	238
15.2.1	DACEx	238
16	HAL DCMI Generic Driver	240
16.1	DCMI Firmware driver registers structures	240
16.1.1	DCMI_CodesInitTypeDef	240
16.1.2	DCMI_SyncUnmaskTypeDef	240
16.1.3	DCMI_InitTypeDef	240
16.1.4	DCMI_HandleTypeDef	241
16.2	DCMI Firmware driver API description	242
16.2.1	How to use this driver	242
16.2.2	Initialization and Configuration functions	243
16.2.3	IO operation functions	244
16.2.4	Peripheral Control functions	244
16.2.5	Peripheral State and Errors functions	244
16.2.6	Detailed description of functions	245
16.3	DCMI Firmware driver defines	249
16.3.1	DCMI	249

17	HAL DFSDM Generic Driver	256
17.1	DFSDM Firmware driver registers structures	256
17.1.1	DFSDM_Channel_OutputClockTypeDef.....	256
17.1.2	DFSDM_Channel_InputTypeDef.....	256
17.1.3	DFSDM_Channel_SerialInterfaceTypeDef	256
17.1.4	DFSDM_Channel_AwdTypeDef.....	257
17.1.5	DFSDM_Channel_InitTypeDef.....	257
17.1.6	DFSDM_Channel_HandleTypeDef	257
17.1.7	DFSDM_Filter_RegularParamTypeDef.....	258
17.1.8	DFSDM_Filter_InjectedParamTypeDef.....	258
17.1.9	DFSDM_Filter_FilterParamTypeDef	258
17.1.10	DFSDM_Filter_InitTypeDef	259
17.1.11	DFSDM_Filter_HandleTypeDef.....	259
17.1.12	DFSDM_Filter_AwdParamTypeDef	260
17.2	DFSDM Firmware driver API description	261
17.2.1	How to use this driver	261
17.2.2	Channel initialization and de-initialization functions	263
17.2.3	Channel operation functions.....	263
17.2.4	Channel state function.....	263
17.2.5	Filter initialization and de-initialization functions	264
17.2.6	Filter control functions	264
17.2.7	Filter operation functions	264
17.2.8	Filter state functions	265
17.2.9	Detailed description of functions	265
17.3	DFSDM Firmware driver defines	279
17.3.1	DFSDM.....	279
18	HAL DFSDM Extension Driver	283
18.1	DFSDMEx Firmware driver API description	283
18.1.1	Extended channel operation functions	283
18.1.2	Detailed description of functions	283
19	HAL DMA2D Generic Driver	284
19.1	DMA2D Firmware driver registers structures	284
19.1.1	DMA2D_ColorTypeDef.....	284
19.1.2	DMA2D_CLUTCfgTypeDef	284
19.1.3	DMA2D_InitTypeDef.....	284
19.1.4	DMA2D_LayerCfgTypeDef.....	285
19.1.5	__DMA2D_HandleTypeDef	286

19.2	DMA2D Firmware driver API description.....	286
19.2.1	How to use this driver	286
19.2.2	Initialization and Configuration functions.....	287
19.2.3	IO operation functions	288
19.2.4	Peripheral Control functions	288
19.2.5	Peripheral State and Errors functions	289
19.2.6	Detailed description of functions	289
19.3	DMA2D Firmware driver defines	296
19.3.1	DMA2D	296
20	HAL DMA Generic Driver	303
20.1	DMA Firmware driver registers structures.....	303
20.1.1	DMA_InitTypeDef	303
20.1.2	__DMA_HandleTypeDef.....	303
20.2	DMA Firmware driver API description	305
20.2.1	How to use this driver	305
20.2.2	Initialization and de-initialization functions	306
20.2.3	IO operation functions	306
20.2.4	Peripheral State and Errors functions	306
20.2.5	Detailed description of functions	306
20.3	DMA Firmware driver defines.....	309
20.3.1	DMA.....	309
21	HAL DMA Extension Driver.....	319
21.1	DMAEx Firmware driver registers structures.....	319
21.1.1	HAL_DMA_MuxSyncConfigTypeDef.....	319
21.1.2	HAL_DMA_MuxRequestGeneratorConfigTypeDef.....	319
21.2	DMAEx Firmware driver API description	320
21.2.1	How to use this driver	320
21.2.2	Extended features functions	320
21.2.3	Detailed description of functions	320
21.3	DMAEx Firmware driver defines.....	322
21.3.1	DMAEx.....	322
22	HAL DSI Generic Driver	325
22.1	DSI Firmware driver registers structures.....	325
22.1.1	DSI_InitTypeDef	325
22.1.2	DSI_PLLInitTypeDef.....	325
22.1.3	DSI_VidCfgTypeDef	325
22.1.4	DSI_CmdCfgTypeDef.....	327

22.1.5	DSI_LPCmdTypeDef	328
22.1.6	DSI_PHY_TimerTypeDef	329
22.1.7	DSI_HOST_TimeoutTypeDef	330
22.1.8	DSI_HandleTypeDef	330
22.2	DSI Firmware driver API description	331
22.2.1	Initialization and Configuration functions	331
22.2.2	IO operation functions	331
22.2.3	Peripheral Control functions	331
22.2.4	Peripheral State and Errors functions	332
22.2.5	Detailed description of functions	333
22.3	DSI Firmware driver defines	343
22.3.1	DSI	343
23	HAL FIREWALL Generic Driver	351
23.1	FIREWALL Firmware driver registers structures	351
23.1.1	FIREWALL_InitTypeDef	351
23.2	FIREWALL Firmware driver API description	351
23.2.1	How to use this driver	351
23.2.2	Initialization and Configuration functions	352
23.2.3	Detailed description of functions	352
23.3	FIREWALL Firmware driver defines	354
23.3.1	FIREWALL	354
24	HAL FLASH Generic Driver	359
24.1	FLASH Firmware driver registers structures	359
24.1.1	FLASH_EraseInitTypeDef	359
24.1.2	FLASH_OBProgramInitTypeDef	359
24.1.3	FLASH_ProcessTypeDef	360
24.2	FLASH Firmware driver API description	361
24.2.1	FLASH peripheral features	361
24.2.2	How to use this driver	361
24.2.3	Programming operation functions	362
24.2.4	Peripheral Control functions	362
24.2.5	Peripheral Errors functions	362
24.2.6	Detailed description of functions	362
24.3	FLASH Firmware driver defines	365
24.3.1	FLASH	365
25	HAL FLASH Extension Driver	375

25.1	FLASHEX Firmware driver API description.....	375
25.1.1	Flash Extended features.....	375
25.1.2	How to use this driver	375
25.1.3	Extended programming operation functions	375
25.1.4	Extended specific configuration functions	375
25.1.5	Detailed description of functions	376
25.2	FLASHEX Firmware driver defines	377
25.2.1	FLASHEX	377
26	HAL FLASH__RAMFUNC Generic Driver	378
26.1	FLASH__RAMFUNC Firmware driver API description	378
26.1.1	Flash RAM functions	378
26.1.2	ramfunc functions	378
26.1.3	Detailed description of functions	378
27	HAL GFXMMU Generic Driver	380
27.1	GFXMMU Firmware driver registers structures	380
27.1.1	GFXMMU_BuffersTypeDef	380
27.1.2	GFXMMU_InterruptsTypeDef.....	380
27.1.3	GFXMMU_InitTypeDef	380
27.1.4	GFXMMU_HandleTypeDef	381
27.1.5	GFXMMU_LutLineTypeDef	381
27.2	GFXMMU Firmware driver API description	382
27.2.1	How to use this driver	382
27.2.2	Initialization and de-initialization functions	382
27.2.3	Operation functions	382
27.2.4	State functions.....	383
27.2.5	Detailed description of functions	383
27.3	GFXMMU Firmware driver defines	386
27.3.1	GFXMMU.....	386
28	HAL GPIO Generic Driver	387
28.1	GPIO Firmware driver registers structures	387
28.1.1	GPIO_InitTypeDef	387
28.2	GPIO Firmware driver API description	387
28.2.1	GPIO Peripheral features	387
28.2.2	How to use this driver	388
28.2.3	Initialization and de-initialization functions	388
28.2.4	IO operation functions	388
28.2.5	Detailed description of functions	389

28.3	GPIO Firmware driver defines.....	391
28.3.1	GPIO.....	391
29	HAL GPIO Extension Driver.....	394
29.1	GPIOEx Firmware driver defines.....	394
29.1.1	GPIOEx	394
30	HAL HASH Generic Driver	397
30.1	HASH Firmware driver registers structures.....	397
30.1.1	HASH_InitTypeDef	397
30.1.2	HASH_HandleTypeDef.....	397
30.2	HASH Firmware driver API description	398
30.2.1	How to use this driver	398
30.2.2	Initialization and de-initialization functions	399
30.2.3	Polling mode HASH processing functions.....	400
30.2.4	Interrupt mode HASH processing functions	400
30.2.5	DMA mode HASH processing functions.....	401
30.2.6	Polling mode HMAC processing functions	401
30.2.7	Interrupt mode HMAC processing functions	401
30.2.8	DMA mode HMAC processing functions.....	402
30.2.9	Peripheral State methods.....	402
30.2.10	Detailed description of functions	402
30.3	HASH Firmware driver defines.....	415
30.3.1	HASH.....	415
31	HAL HASH Extension Driver.....	420
31.1	HASHEX Firmware driver API description	420
31.1.1	HASH peripheral extended features.....	420
31.1.2	Polling mode HASH extended processing functions.....	420
31.1.3	Interrupt mode HASH extended processing functions	421
31.1.4	DMA mode HASH extended processing functions.....	421
31.1.5	Polling mode HMAC extended processing functions	421
31.1.6	Interrupt mode HMAC extended processing functions.....	422
31.1.7	DMA mode HMAC extended processing functions	422
31.1.8	Multi-buffer DMA mode HMAC extended processing functions	422
31.1.9	Detailed description of functions	423
32	HAL HCD Generic Driver.....	436
32.1	HCD Firmware driver registers structures	436
32.1.1	HCD_HandleTypeDef.....	436

32.2	HCD Firmware driver API description	436
32.2.1	How to use this driver	436
32.2.2	Initialization and de-initialization functions	436
32.2.3	IO operation functions	437
32.2.4	Peripheral Control functions	437
32.2.5	Peripheral State functions	437
32.2.6	Detailed description of functions	437
32.3	HCD Firmware driver defines	442
32.3.1	HCD	442
33	HAL I2C Generic Driver	443
33.1	I2C Firmware driver registers structures	443
33.1.1	I2C_InitTypeDef	443
33.1.2	__I2C_HandleTypeDef	443
33.2	I2C Firmware driver API description	444
33.2.1	How to use this driver	444
33.2.2	Initialization and de-initialization functions	449
33.2.3	IO operation functions	449
33.2.4	Peripheral State, Mode and Error functions	451
33.2.5	Detailed description of functions	451
33.3	I2C Firmware driver defines	463
33.3.1	I2C	463
34	HAL I2C Extension Driver	469
34.1	I2CEx Firmware driver API description	469
34.1.1	I2C peripheral Extended features	469
34.1.2	How to use this driver	469
34.1.3	Extended features functions	469
34.1.4	Detailed description of functions	469
34.2	I2CEx Firmware driver defines	471
34.2.1	I2CEx	471
35	HAL IRDA Generic Driver	473
35.1	IRDA Firmware driver registers structures	473
35.1.1	IRDA_InitTypeDef	473
35.1.2	IRDA_HandleTypeDef	473
35.2	IRDA Firmware driver API description	474
35.2.1	How to use this driver	474
35.2.2	Initialization and Configuration functions	476
35.2.3	IO operation functions	476

35.2.4	Peripheral State and Error functions	478
35.2.5	Detailed description of functions	478
35.3	IRDA Firmware driver defines	486
35.3.1	IRDA	486
36	HAL IWDG Generic Driver	495
36.1	IWDG Firmware driver registers structures	495
36.1.1	IWDG_InitTypeDef	495
36.1.2	IWDG_HandleTypeDef	495
36.2	IWDG Firmware driver API description	495
36.2.1	IWDG Generic features	495
36.2.2	How to use this driver	496
36.2.3	Initialization and Start functions	496
36.2.4	IO operation functions	496
36.2.5	Detailed description of functions	497
36.3	IWDG Firmware driver defines	497
36.3.1	IWDG	497
37	HAL LCD Generic Driver	499
37.1	LCD Firmware driver registers structures	499
37.1.1	LCD_InitTypeDef	499
37.1.2	LCD_HandleTypeDef	500
37.2	LCD Firmware driver API description	500
37.2.1	How to use this driver	500
37.2.2	Initialization and Configuration functions	501
37.2.3	IO operation functions	501
37.2.4	Peripheral State functions	501
37.2.5	Detailed description of functions	502
37.3	LCD Firmware driver defines	504
37.3.1	LCD	504
38	HAL LPTIM Generic Driver	514
38.1	LPTIM Firmware driver registers structures	514
38.1.1	LPTIM_ClockConfigTypeDef	514
38.1.2	LPTIM_ULPClockConfigTypeDef	514
38.1.3	LPTIM_TriggerConfigTypeDef	514
38.1.4	LPTIM_InitTypeDef	515
38.1.5	LPTIM_HandleTypeDef	515
38.2	LPTIM Firmware driver API description	516

38.2.1	How to use this driver	516
38.2.2	Initialization and de-initialization functions	517
38.2.3	LPTIM Start Stop operation functions	517
38.2.4	LPTIM Read operation functions	518
38.2.5	LPTIM IRQ handler and callbacks.....	518
38.2.6	Peripheral State functions	518
38.2.7	Detailed description of functions	519
38.3	LPTIM Firmware driver defines	527
38.3.1	LPTIM	527
39	HAL LTDC Generic Driver	533
39.1	LTDC Firmware driver registers structures.....	533
39.1.1	LTDC_ColorTypeDef	533
39.1.2	LTDC_InitTypeDef	533
39.1.3	LTDC_LayerCfgTypeDef	534
39.1.4	LTDC_HandleTypeDef	535
39.2	LTDC Firmware driver API description	536
39.2.1	How to use this driver	536
39.2.2	Initialization and Configuration functions.....	537
39.2.3	IO operation functions	537
39.2.4	Peripheral Control functions	537
39.2.5	Peripheral State and Errors functions	538
39.2.6	Detailed description of functions	538
39.3	LTDC Firmware driver defines	549
39.3.1	LTDC	549
40	HAL LTDC Extension Driver	555
40.1	LTDCEx Firmware driver API description.....	555
40.1.1	Initialization and Configuration functions.....	555
40.1.2	Detailed description of functions	555
41	HAL NAND Generic Driver	556
41.1	NAND Firmware driver registers structures.....	556
41.1.1	NAND_IDTypeDef	556
41.1.2	NAND_AddressTypeDef.....	556
41.1.3	NAND_InfoTypeDef	556
41.1.4	NAND_HandleTypeDef	557
41.2	NAND Firmware driver API description	557
41.2.1	How to use this driver	557
41.2.2	NAND Initialization and de-initialization functions	558

41.2.3	NAND Input and Output functions	558
41.2.4	NAND Control functions	558
41.2.5	NAND State functions.....	558
41.2.6	Detailed description of functions	559
41.3	NAND Firmware driver defines.....	563
41.3.1	NAND.....	563
42	HAL NOR Generic Driver.....	564
42.1	NOR Firmware driver registers structures	564
42.1.1	NOR_IDTypeDef	564
42.1.2	NOR_CFTypeDef	564
42.1.3	NOR_HandleTypeDef.....	564
42.2	NOR Firmware driver API description	565
42.2.1	How to use this driver	565
42.2.2	NOR Initialization and de-initialization functions	565
42.2.3	NOR Input and Output functions	566
42.2.4	NOR Control functions.....	566
42.2.5	NOR State functions.....	566
42.2.6	Detailed description of functions	566
42.3	NOR Firmware driver defines.....	571
42.3.1	NOR.....	571
43	HAL OPAMP Generic Driver	572
43.1	OPAMP Firmware driver registers structures	572
43.1.1	OPAMP_InitTypeDef	572
43.1.2	OPAMP_HandleTypeDef.....	573
43.2	OPAMP Firmware driver API description	573
43.2.1	OPAMP Peripheral Features.....	573
43.2.2	How to use this driver	574
43.2.3	Initialization and de-initialization functions	575
43.2.4	IO operation functions	575
43.2.5	Peripheral Control functions	575
43.2.6	Peripheral State functions	576
43.2.7	Detailed description of functions	576
43.3	OPAMP Firmware driver defines.....	578
43.3.1	OPAMP	578
44	HAL OPAMP Extension Driver.....	580
44.1	OPAMPEX Firmware driver API description	580

44.1.1	Extended IO operation functions	580
44.1.2	Peripheral Control functions	580
44.1.3	Detailed description of functions	580
45	HAL OSPI Generic Driver	581
45.1	OSPI Firmware driver registers structures	581
45.1.1	OSPI_InitTypeDef.....	581
45.1.2	OSPI_HandleTypeDef.....	581
45.1.3	OSPI_RegularCmdTypeDef	582
45.1.4	OSPI_HyperbusCfgTypeDef	583
45.1.5	OSPI_HyperbusCmdTypeDef	583
45.1.6	OSPI_AutoPollingTypeDef	583
45.1.7	OSPI_MemoryMappedTypeDef	583
45.1.8	OSPI_M_CfgTypeDef.....	584
45.2	OSPI Firmware driver API description.....	584
45.2.1	How to use this driver	584
45.2.2	Initialization and Configuration functions.....	586
45.2.3	IO operation functions	586
45.2.4	Peripheral Control and State functions.....	587
45.2.5	IO Manager configuration function	587
45.2.6	Detailed description of functions	588
45.3	OSPI Firmware driver defines	595
45.3.1	OSPI	595
46	HAL PCD Generic Driver	603
46.1	PCD Firmware driver registers structures	603
46.1.1	PCD_HandleTypeDef.....	603
46.2	PCD Firmware driver API description.....	604
46.2.1	How to use this driver	604
46.2.2	Initialization and de-initialization functions	604
46.2.3	IO operation functions	604
46.2.4	Peripheral Control functions	604
46.2.5	Peripheral State functions	605
46.2.6	Detailed description of functions	605
46.3	PCD Firmware driver defines	611
46.3.1	PCD	611
47	HAL PCD Extension Driver	613
47.1	PCDEx Firmware driver API description	613
47.1.1	Extended features functions	613

47.1.2	Detailed description of functions	613
48	HAL QSPI Generic Driver	616
48.1	QSPI Firmware driver registers structures	616
48.1.1	QSPI_InitTypeDef.....	616
48.1.2	QSPI_HandleTypeDef.....	616
48.1.3	QSPI_CommandTypeDef.....	617
48.1.4	QSPI_AutoPollingTypeDef	617
48.1.5	QSPI_MemoryMappedTypeDef	618
48.2	QSPI Firmware driver API description.....	618
48.2.1	How to use this driver	618
48.2.2	Initialization and Configuration functions.....	620
48.2.3	IO operation functions	620
48.2.4	Peripheral Control and State functions.....	621
48.2.5	Detailed description of functions	621
48.3	QSPI Firmware driver defines	628
48.3.1	QSPI	628
49	HAL PWR Generic Driver	635
49.1	PWR Firmware driver registers structures	635
49.1.1	PWR_PVDTypeDef	635
49.2	PWR Firmware driver API description.....	635
49.2.1	Initialization and de-initialization functions	635
49.2.2	Peripheral Control functions	635
49.2.3	Detailed description of functions	638
49.3	PWR Firmware driver defines	643
49.3.1	PWR	643
50	HAL PWR Extension Driver	649
50.1	PWREx Firmware driver registers structures	649
50.1.1	PWR_PVMTypeDef.....	649
50.2	PWREx Firmware driver API description.....	649
50.2.1	Extended Peripheral Initialization and de-initialization functions....	649
50.2.2	Detailed description of functions	650
50.3	PWREx Firmware driver defines	660
50.3.1	PWREx	660
51	HAL RCC Generic Driver.....	672
51.1	RCC Firmware driver registers structures	672
51.1.1	RCC_PLLInitTypeDef	672

51.1.2	RCC_OscInitTypeDef	672
51.1.3	RCC_ClkInitTypeDef	673
51.2	RCC Firmware driver API description	674
51.2.1	RCC specific features	674
51.2.2	Initialization and de-initialization functions	674
51.2.3	Peripheral Control functions	675
51.2.4	Detailed description of functions	676
51.3	RCC Firmware driver defines	681
51.3.1	RCC	681
52	HAL RCC Extension Driver	724
52.1	RCCEX Firmware driver registers structures	724
52.1.1	RCC_PLLSAI1InitTypeDef	724
52.1.2	RCC_PLLSAI2InitTypeDef	724
52.1.3	RCC_PeriphCLKInitTypeDef	725
52.1.4	RCC_CRSSInitTypeDef	727
52.1.5	RCC_CRSSynchroInfoTypeDef	728
52.2	RCCEX Firmware driver API description	728
52.2.1	Extended Peripheral Control functions	728
52.2.2	Extended clock management functions	729
52.2.3	Extended Clock Recovery System Control functions	729
52.2.4	Detailed description of functions	730
52.3	RCCEX Firmware driver defines	737
52.3.1	RCCEX	737
53	HAL RNG Generic Driver	774
53.1	RNG Firmware driver registers structures	774
53.1.1	RNG_InitTypeDef	774
53.1.2	NG_HandleTypeDef	774
53.2	RNG Firmware driver API description	774
53.2.1	How to use this driver	774
53.2.2	Initialization and de-initialization functions	774
53.2.3	Peripheral Control functions	775
53.2.4	Peripheral State functions	775
53.2.5	Detailed description of functions	775
53.3	RNG Firmware driver defines	778
53.3.1	RNG	778
54	HAL RTC Generic Driver	781
54.1	RTC Firmware driver registers structures	781

54.1.1	RTC_InitTypeDef	781
54.1.2	RTC_TimeTypeDef	781
54.1.3	RTC_DateTypeDef	782
54.1.4	RTC_AlarmTypeDef	783
54.1.5	RTC_HandleTypeDef	783
54.2	RTC Firmware driver API description	783
54.2.1	RTC Operating Condition	783
54.2.2	Backup Domain Reset	784
54.2.3	Backup Domain Access	784
54.2.4	How to use RTC Driver	784
54.2.5	RTC and low power modes	784
54.2.6	Initialization and de-initialization functions	785
54.2.7	RTC Time and Date functions	785
54.2.8	RTC Alarm functions	785
54.2.9	Peripheral Control functions	786
54.2.10	Peripheral State functions	786
54.2.11	Detailed description of functions	786
54.3	RTC Firmware driver defines	791
54.3.1	RTC	791
55	HAL RTC Extension Driver	801
55.1	RTCEX Firmware driver registers structures	801
55.1.1	RTC_TamperTypeDef	801
55.2	RTCEX Firmware driver API description	802
55.2.1	How to use this driver	802
55.2.2	RTC TimeStamp and Tamper functions	803
55.2.3	RTC Wake-up functions	803
55.2.4	Extended Peripheral Control functions	803
55.2.5	Extended features functions	804
55.2.6	Detailed description of functions	804
55.3	RTCEX Firmware driver defines	813
55.3.1	RTCEX	813
56	HAL SAI Generic Driver	833
56.1	SAI Firmware driver registers structures	833
56.1.1	SAI_PdmInitTypeDef	833
56.1.2	SAI_InitTypeDef	833
56.1.3	SAI_FrameInitTypeDef	835
56.1.4	SAI_SlotInitTypeDef	835

56.1.5	__SAI_HandleTypeDef	836
56.2	SAI Firmware driver API description	837
56.2.1	How to use this driver	837
56.2.2	Initialization and de-initialization functions	839
56.2.3	IO operation functions	839
56.2.4	Peripheral State and Errors functions	840
56.2.5	Detailed description of functions	841
56.3	SAI Firmware driver defines	846
56.3.1	SAI	846
57	HAL SAI Extension Driver	854
57.1	SAIEx Firmware driver registers structures	854
57.1.1	SAIEx_PdmMicDelayParamTypeDef	854
57.2	SAIEx Firmware driver API description	854
57.2.1	Extended features functions	854
57.2.2	Detailed description of functions	854
58	HAL SD Extension Driver	855
58.1	SDEx Firmware driver API description	855
58.1.1	How to use this driver	855
58.1.2	High Speed function	855
58.1.3	Multibuffer functions	855
58.1.4	Detailed description of functions	855
59	HAL SMARTCARD Generic Driver	859
59.1	SMARTCARD Firmware driver registers structures	859
59.1.1	SMARTCARD_InitTypeDef	859
59.1.2	SMARTCARD_AdvFeatureInitTypeDef	860
59.1.3	__SMARTCARD_HandleTypeDef	861
59.2	SMARTCARD Firmware driver API description	862
59.2.1	How to use this driver	862
59.2.2	Initialization and Configuration functions	864
59.2.3	IO operation functions	865
59.2.4	Peripheral State and Errors functions	868
59.2.5	Detailed description of functions	868
59.3	SMARTCARD Firmware driver defines	876
59.3.1	SMARTCARD	876
60	HAL SMARTCARD Extension Driver	887
60.1	SMARTCARDEx Firmware driver API description	887

60.1.1	SMARTCARD peripheral extended features.....	887
60.1.2	IO operation functions	887
60.1.3	Peripheral Control functions	887
60.1.4	Detailed description of functions	887
60.2	SMARTCARDEx Firmware driver defines	890
60.2.1	SMARTCARDEx.....	890
61	HAL SMBUS Generic Driver.....	893
61.1	SMBUS Firmware driver registers structures	893
61.1.1	SMBUS_InitTypeDef	893
61.1.2	SMBUS_HandleTypeDef.....	894
61.2	SMBUS Firmware driver API description	894
61.2.1	How to use this driver	894
61.2.2	Initialization and de-initialization functions	896
61.2.3	IO operation functions	897
61.2.4	Peripheral State and Errors functions	898
61.2.5	Detailed description of functions	898
61.3	SMBUS Firmware driver defines	905
61.3.1	SMBUS	905
62	HAL SPI Generic Driver.....	912
62.1	SPI Firmware driver registers structures	912
62.1.1	SPI_InitTypeDef	912
62.1.2	__SPI_HandleTypeDef.....	913
62.2	SPI Firmware driver API description	914
62.2.1	How to use this driver	914
62.2.2	Initialization and de-initialization functions	915
62.2.3	IO operation functions	915
62.2.4	Peripheral State and Errors functions	916
62.2.5	Detailed description of functions	916
62.3	SPI Firmware driver defines	923
62.3.1	SPI	923
63	HAL SPI Extension Driver	930
63.1	SPIEx Firmware driver API description	930
63.1.1	IO operation functions	930
63.1.2	Detailed description of functions	930
64	HAL SRAM Generic Driver	931
64.1	SRAM Firmware driver registers structures.....	931

64.1.1	SRAM_HandleTypeDef	931
64.2	SRAM Firmware driver API description	931
64.2.1	How to use this driver	931
64.2.2	SRAM Initialization and de-initialization functions	932
64.2.3	SRAM Input and Output functions	932
64.2.4	SRAM Control functions	932
64.2.5	SRAM State functions	933
64.2.6	Detailed description of functions	933
64.3	SRAM Firmware driver defines	937
64.3.1	SRAM	937
65	HAL TIM Generic Driver	938
65.1	TIM Firmware driver registers structures	938
65.1.1	TIM_Base_InitTypeDef	938
65.1.2	TIM_OC_InitTypeDef	938
65.1.3	TIM_OnePulse_InitTypeDef	939
65.1.4	TIM_IC_InitTypeDef	940
65.1.5	TIM_Encoder_InitTypeDef	940
65.1.6	TIM_ClockConfigTypeDef	941
65.1.7	TIM_ClearInputConfigTypeDef	941
65.1.8	TIM_MasterConfigTypeDef	942
65.1.9	TIM_SlaveConfigTypeDef	942
65.1.10	TIM_BreakDeadTimeConfigTypeDef	943
65.1.11	TIM_HandleTypeDef	943
65.2	TIM Firmware driver API description	944
65.2.1	TIMER Generic features	944
65.2.2	How to use this driver	944
65.2.3	Time Base functions	945
65.2.4	Time Output Compare functions	946
65.2.5	Time PWM functions	946
65.2.6	Time Input Capture functions	947
65.2.7	Time One Pulse functions	947
65.2.8	Time Encoder functions	947
65.2.9	IRQ handler management	948
65.2.10	Peripheral Control functions	948
65.2.11	TIM Callbacks functions	949
65.2.12	Peripheral State functions	949
65.2.13	Detailed description of functions	949
65.3	TIM Firmware driver defines	974

65.3.1	TIM.....	974
66	HAL TIM Extension Driver.....	996
66.1	TIMEx Firmware driver registers structures.....	996
66.1.1	TIM_HallSensor_InitTypeDef	996
66.1.2	TIMEx_BreakInputConfigTypeDef.....	996
66.2	TIMEx Firmware driver API description.....	996
66.2.1	TIMER Extended features	996
66.2.2	How to use this driver	997
66.2.3	Timer Hall Sensor functions	997
66.2.4	Timer Complementary Output Compare functions.....	998
66.2.5	Timer Complementary PWM functions.....	998
66.2.6	Timer Complementary One Pulse functions.....	999
66.2.7	Peripheral Control functions	999
66.2.8	Extended Callbacks functions	999
66.2.9	Extended Peripheral State functions	999
66.2.10	Detailed description of functions	1000
66.3	TIMEx Firmware driver defines	1011
66.3.1	TIMEx	1011
67	HAL TSC Generic Driver	1013
67.1	TSC Firmware driver registers structures.....	1013
67.1.1	TSC_InitTypeDef	1013
67.1.2	TSC_IOConfigTypeDef.....	1014
67.1.3	TSC_HandleTypeDef	1014
67.2	TSC Firmware driver API description.....	1014
67.2.1	TSC specific features	1014
67.2.2	How to use this driver	1015
67.2.3	Initialization and de-initialization functions	1015
67.2.4	IO Operation functions.....	1015
67.2.5	Peripheral Control functions	1016
67.2.6	State and Errors functions.....	1016
67.2.7	Detailed description of functions	1016
67.3	TSC Firmware driver defines.....	1020
67.3.1	TSC.....	1020
68	HAL UART Generic Driver.....	1029
68.1	UART Firmware driver registers structures	1029
68.1.1	UART_InitTypeDef	1029
68.1.2	UART_AdvFeatureInitTypeDef.....	1030

68.1.3	__UART_HandleTypeDef	1030
68.2	UART Firmware driver API description	1032
68.2.1	How to use this driver	1032
68.2.2	Initialization and Configuration functions	1033
68.2.3	IO operation functions	1034
68.2.4	Peripheral Control functions	1034
68.2.5	Peripheral State and Error functions	1035
68.2.6	Detailed description of functions	1035
68.3	UART Firmware driver defines	1045
68.3.1	UART	1045
69	HAL UART Extension Driver	1061
69.1	UARTEEx Firmware driver registers structures	1061
69.1.1	UART_WakeUpTypeDef	1061
69.2	UARTEEx Firmware driver API description	1061
69.2.1	UART peripheral extended features	1061
69.2.2	Initialization and Configuration functions	1061
69.2.3	IO operation functions	1062
69.2.4	Peripheral Control functions	1062
69.2.5	Detailed description of functions	1063
69.3	UARTEEx Firmware driver defines	1067
69.3.1	UARTEEx	1067
70	HAL USART Generic Driver	1069
70.1	USART Firmware driver registers structures	1069
70.1.1	USART_InitTypeDef	1069
70.1.2	__USART_HandleTypeDef	1070
70.2	USART Firmware driver API description	1071
70.2.1	How to use this driver	1071
70.2.2	Initialization and Configuration functions	1072
70.2.3	IO operation functions	1072
70.2.4	Peripheral State and Error functions	1074
70.2.5	Detailed description of functions	1074
70.3	USART Firmware driver defines	1080
70.3.1	USART	1080
71	HAL USART Extension Driver	1091
71.1	USARTEEx Firmware driver API description	1091
71.1.1	USART peripheral extended features	1091
71.1.2	IO operation functions	1091

71.1.3	Peripheral Control functions	1091
71.1.4	Detailed description of functions	1091
71.2	USARTEx Firmware driver defines	1094
71.2.1	USARTEx	1094
72	HAL WWDG Generic Driver	1095
72.1	WWDG Firmware driver registers structures	1095
72.1.1	WWDG_InitTypeDef	1095
72.1.2	WWDG_HandleTypeDef	1095
72.2	WWDG Firmware driver API description	1095
72.2.1	WWDG specific features	1095
72.2.2	How to use this driver	1096
72.2.3	Initialization and Configuration functions	1096
72.2.4	IO operation functions	1097
72.2.5	Detailed description of functions	1097
72.3	WWDG Firmware driver defines	1098
72.3.1	WWDG	1098
73	LL ADC Generic Driver	1101
73.1	ADC Firmware driver registers structures	1101
73.1.1	LL_ADC_CommonInitTypeDef	1101
73.1.2	LL_ADC_InitTypeDef	1101
73.1.3	LL_ADC_REG_InitTypeDef	1101
73.1.4	LL_ADC_INJ_InitTypeDef	1102
73.2	ADC Firmware driver API description	1103
73.2.1	Detailed description of functions	1103
73.3	ADC Firmware driver defines	1190
73.3.1	ADC	1190
74	LL BUS Generic Driver	1227
74.1	BUS Firmware driver API description	1227
74.1.1	Detailed description of functions	1227
74.2	BUS Firmware driver defines	1258
74.2.1	BUS	1258
75	LL COMP Generic Driver	1261
75.1	COMP Firmware driver registers structures	1261
75.1.1	LL_COMP_InitTypeDef	1261
75.2	COMP Firmware driver API description	1261
75.2.1	Detailed description of functions	1261

75.3	COMP Firmware driver defines	1272
75.3.1	COMP	1272
76	LL CORTEX Generic Driver.....	1276
76.1	CORTEX Firmware driver API description	1276
76.1.1	Detailed description of functions	1276
76.2	CORTEX Firmware driver defines.....	1283
76.2.1	CORTEX.....	1283
77	LL CRC Generic Driver.....	1286
77.1	CRC Firmware driver API description	1286
77.1.1	Detailed description of functions	1286
77.2	CRC Firmware driver defines	1292
77.2.1	CRC	1292
78	LL CRS Generic Driver	1294
78.1	CRS Firmware driver API description.....	1294
78.1.1	Detailed description of functions	1294
78.2	CRS Firmware driver defines	1305
78.2.1	CRS	1305
79	LL DAC Generic Driver.....	1308
79.1	DAC Firmware driver registers structures	1308
79.1.1	LL_DAC_InitTypeDef.....	1308
79.2	DAC Firmware driver API description.....	1309
79.2.1	Detailed description of functions	1309
79.3	DAC Firmware driver defines	1334
79.3.1	DAC	1334
80	LL DMA2D Generic Driver.....	1341
80.1	DMA2D Firmware driver registers structures	1341
80.1.1	LL_DMA2D_InitTypeDef.....	1341
80.1.2	LL_DMA2D_LayerCfgTypeDef.....	1343
80.1.3	LL_DMA2D_ColorTypeDef.....	1344
80.2	DMA2D Firmware driver API description.....	1345
80.2.1	Detailed description of functions	1345
80.3	DMA2D Firmware driver defines	1384
80.3.1	DMA2D	1384
81	LL DMAMUX Generic Driver	1388
81.1	DMAMUX Firmware driver API description	1388

81.1.1	Detailed description of functions	1388
81.2	DMAMUX Firmware driver defines	1417
81.2.1	DMAMUX	1417
82	LL DMA Generic Driver	1428
82.1	DMA Firmware driver registers structures	1428
82.1.1	LL_DMA_InitTypeDef	1428
82.2	DMA Firmware driver API description	1429
82.2.1	Detailed description of functions	1429
82.3	DMA Firmware driver defines	1466
82.3.1	DMA	1466
83	LL EXTI Generic Driver	1471
83.1	EXTI Firmware driver registers structures	1471
83.1.1	LL_EXTI_InitTypeDef	1471
83.2	EXTI Firmware driver API description	1471
83.2.1	Detailed description of functions	1471
83.3	EXTI Firmware driver defines	1493
83.3.1	EXTI	1493
84	LL GPIO Generic Driver	1496
84.1	GPIO Firmware driver registers structures	1496
84.1.1	LL_GPIO_InitTypeDef	1496
84.2	GPIO Firmware driver API description	1496
84.2.1	Detailed description of functions	1496
84.3	GPIO Firmware driver defines	1511
84.3.1	GPIO	1511
85	LL I2C Generic Driver	1514
85.1	I2C Firmware driver registers structures	1514
85.1.1	LL_I2C_InitTypeDef	1514
85.2	I2C Firmware driver API description	1515
85.2.1	Detailed description of functions	1515
85.3	I2C Firmware driver defines	1555
85.3.1	I2C	1555
86	LL IWDG Generic Driver	1560
86.1	IWDG Firmware driver API description	1560
86.1.1	Detailed description of functions	1560
86.2	IWDG Firmware driver defines	1564

86.2.1	IWDG	1564
87	LL LPTIM Generic Driver	1565
87.1	LPTIM Firmware driver registers structures	1565
87.1.1	LL_LPTIM_InitTypeDef	1565
87.2	LPTIM Firmware driver API description	1565
87.2.1	Detailed description of functions	1565
87.3	LPTIM Firmware driver defines	1589
87.3.1	LPTIM	1589
88	LL LPUART Generic Driver	1593
88.1	LPUART Firmware driver registers structures	1593
88.1.1	LL_LPUART_InitTypeDef	1593
88.2	LPUART Firmware driver API description	1593
88.2.1	Detailed description of functions	1593
88.3	LPUART Firmware driver defines	1639
88.3.1	LPUART	1639
89	LL OPAMP Generic Driver	1644
89.1	OPAMP Firmware driver registers structures	1644
89.1.1	LL_OPAMP_InitTypeDef	1644
89.2	OPAMP Firmware driver API description	1644
89.2.1	Detailed description of functions	1644
89.3	OPAMP Firmware driver defines	1654
89.3.1	OPAMP	1654
90	LL PWR Generic Driver	1658
90.1	PWR Firmware driver API description	1658
90.1.1	Detailed description of functions	1658
90.2	PWR Firmware driver defines	1683
90.2.1	PWR	1683
91	LL RCC Generic Driver	1686
91.1	RCC Firmware driver registers structures	1686
91.1.1	LL_RCC_ClocksTypeDef	1686
91.2	RCC Firmware driver API description	1686
91.2.1	Detailed description of functions	1686
91.3	RCC Firmware driver defines	1756
91.3.1	RCC	1756
92	LL RNG Generic Driver	1780

92.1	RNG Firmware driver registers structures	1780
92.1.1	LL_RNG_InitTypeDef	1780
92.2	RNG Firmware driver API description	1780
92.2.1	Detailed description of functions	1780
92.3	RNG Firmware driver defines	1785
92.3.1	RNG	1785
93	LL RTC Generic Driver	1786
93.1	RTC Firmware driver registers structures	1786
93.1.1	LL_RTC_InitTypeDef	1786
93.1.2	LL_RTC_TimeTypeDef	1786
93.1.3	LL_RTC_DateTypeDef	1787
93.1.4	LL_RTC_AlarmTypeDef	1787
93.2	RTC Firmware driver API description	1788
93.2.1	Detailed description of functions	1788
93.3	RTC Firmware driver defines	1856
93.3.1	RTC	1856
94	LL SPI Generic Driver	1866
94.1	SPI Firmware driver registers structures	1866
94.1.1	LL_SPI_InitTypeDef	1866
94.2	SPI Firmware driver API description	1867
94.2.1	Detailed description of functions	1867
94.3	SPI Firmware driver defines	1889
94.3.1	SPI	1889
95	LL SYSTEM Generic Driver	1893
95.1	SYSTEM Firmware driver API description	1893
95.1.1	Detailed description of functions	1893
95.2	SYSTEM Firmware driver defines	1916
95.2.1	SYSTEM	1916
96	LL TIM Generic Driver	1923
96.1	TIM Firmware driver registers structures	1923
96.1.1	LL_TIM_InitTypeDef	1923
96.1.2	LL_TIM_OC_InitTypeDef	1923
96.1.3	LL_TIM_IC_InitTypeDef	1924
96.1.4	LL_TIM_ENCODER_InitTypeDef	1925
96.1.5	LL_TIM_HALLSENSOR_InitTypeDef	1926
96.1.6	LL_TIM_BDTR_InitTypeDef	1926

96.2	TIM Firmware driver API description	1928
96.2.1	Detailed description of functions	1928
96.3	TIM Firmware driver defines.....	2003
96.3.1	TIM.....	2003
97	LL USART Generic Driver	2020
97.1	USART Firmware driver registers structures.....	2020
97.1.1	LL_USART_InitTypeDef	2020
97.1.2	LL_USART_ClockInitTypeDef	2020
97.2	USART Firmware driver API description	2021
97.2.1	Detailed description of functions	2021
97.3	USART Firmware driver defines.....	2103
97.3.1	USART.....	2103
98	LL UTILS Generic Driver	2111
98.1	UTILS Firmware driver registers structures.....	2111
98.1.1	LL_UTILS_PLLInitTypeDef	2111
98.1.2	LL_UTILS_ClkInitTypeDef.....	2111
98.2	UTILS Firmware driver API description	2111
98.2.1	System Configuration functions.....	2111
98.2.2	Detailed description of functions	2112
98.3	UTILS Firmware driver defines.....	2116
98.3.1	UTILS.....	2116
99	LL WWDG Generic Driver	2117
99.1	WWDG Firmware driver API description	2117
99.1.1	Detailed description of functions	2117
99.2	WWDG Firmware driver defines.....	2121
99.2.1	WWDG.....	2121
100	Correspondence between API registers and API low-layer driver functions.....	2122
100.1	ADC	2122
100.2	BUS.....	2131
100.3	COMP	2145
100.4	CORTEX	2146
100.5	CRC	2147
100.6	CRS	2148
100.7	DAC	2149

100.8	DMA	2152
100.9	DMA2D	2155
100.10	DMAMUX	2159
100.11	EXTI	2161
100.12	GPIO	2162
100.13	I2C	2163
100.14	IWDG	2167
100.15	LPTIM	2168
100.16	LPUART	2170
100.17	OPAMP	2175
100.18	PWR.....	2176
100.19	RCC	2181
100.20	RNG	2188
100.21	RTC.....	2188
100.22	SPI	2197
100.23	SYSTEM	2199
100.24	TIM.....	2202
100.25	USART	2214
100.26	WWDG.....	2222
101	FAQs.....	2224
102	Revision history	2228

List of tables

Table 1: Acronyms and definitions.....	34
Table 2: HAL driver files.....	36
Table 3: User-application files	37
Table 4: API classification.....	41
Table 5: List of STM32L4 Series devices supported by HAL drivers	42
Table 6: List of STM32L4+ Series devices supported by HAL drivers	45
Table 7: HAL API naming rules	47
Table 8: Macros handling interrupts and specific clock configurations	49
Table 9: Callback functions.....	50
Table 10: HAL generic APIs	51
Table 11: HAL extension APIs	52
Table 12: Define statements used for HAL configuration	55
Table 13: Description of GPIO_InitTypeDef structure	58
Table 14: Description of EXTI configuration macros	60
Table 15: MSP functions.....	64
Table 16: Timeout values	69
Table 17: LL driver files.....	73
Table 18: Common peripheral initialization functions	76
Table 19: Optional peripheral initialization functions	77
Table 20: Specific Interrupt, DMA request and status flags management	79
Table 21: Available function formats.....	79
Table 22: Peripheral clock activation/deactivation management	79
Table 23: Peripheral activation/deactivation management.....	80
Table 24: Peripheral configuration management.....	80
Table 25: Peripheral register management	80
Table 26: Correspondence between ADC registers and ADC low-layer driver functions	2122
Table 27: Correspondence between BUS registers and BUS low-layer driver functions.....	2131
Table 28: Correspondence between COMP registers and COMP low-layer driver functions.....	2145
Table 29: Correspondence between CORTEX registers and CORTEX low-layer driver functions	2146
Table 30: Correspondence between CRC registers and CRC low-layer driver functions.....	2147
Table 31: Correspondence between CRS registers and CRS low-layer driver functions	2148
Table 32: Correspondence between DAC registers and DAC low-layer driver functions	2149
Table 33: Correspondence between DMA registers and DMA low-layer driver functions	2152
Table 34: Correspondence between DMA2D registers and DMA2D low-layer driver functions	2155
Table 35: Correspondence between DMAMUX registers and DMAMUX low-layer driver functions	2159
Table 36: Correspondence between EXTI registers and EXTI low-layer driver functions	2161
Table 37: Correspondence between GPIO registers and GPIO low-layer driver functions	2162
Table 38: Correspondence between I2C registers and I2C low-layer driver functions	2163
Table 39: Correspondence between IWDG registers and IWDG low-layer driver functions.....	2167
Table 40: Correspondence between LPTIM registers and LPTIM low-layer driver functions	2168
Table 41: Correspondence between LPUART registers and LPUART low-layer driver functions.....	2170
Table 42: Correspondence between OPAMP registers and OPAMP low-layer driver functions	2175
Table 43: Correspondence between PWR registers and PWR low-layer driver functions.....	2176
Table 44: Correspondence between RCC registers and RCC low-layer driver functions.....	2181
Table 45: Correspondence between RNG registers and RNG low-layer driver functions	2188
Table 46: Correspondence between RTC registers and RTC low-layer driver functions.....	2188
Table 47: Correspondence between SPI registers and SPI low-layer driver functions.....	2197
Table 48: Correspondence between SYSTEM registers and SYSTEM low-layer driver functions.....	2199
Table 49: Correspondence between TIM registers and TIM low-layer driver functions	2202
Table 50: Correspondence between USART registers and USART low-layer driver functions	2214
Table 51: Correspondence between WWDG registers and WWDG low-layer driver functions	2222
Table 52: Document revision history	2228

List of figures

Figure 1: Example of project template	38
Figure 2: Adding device-specific functions	52
Figure 3: Adding family-specific functions	53
Figure 4: Adding new peripherals	53
Figure 5: Updating existing APIs	53
Figure 6: File inclusion model	54
Figure 7: HAL driver model.....	62
Figure 8: Low-layer driver folders	74
Figure 9: Low-layer driver CMSIS files	75

1 Acronyms and definitions

Table 1: Acronyms and definitions

Acronym	Definition
ADC	Analog-to-digital converter
AES	Advanced encryption standard
ANSI	American National Standards Institute
API	Application Programming Interface
BSP	Board Support Package
CAN	Controller area network
CMSIS	Cortex Microcontroller Software Interface Standard
COMP	Comparator
CPU	Central Processing Unit
CRC	CRC calculation unit
CSS	Clock security system
DAC	Digital to analog converter
DFSDM	Digital filter sigma delta modulator
DMA	Direct Memory Access
EXTI	External interrupt/event controller
FLASH	Flash memory
FMC	Flexible memory controller
FW	Firewall
GPIO	General purpose I/Os
HAL	Hardware abstraction layer
HCD	USB Host Controller Driver
I2C	Inter-integrated circuit
I2S	Inter-integrated sound
IRDA	InfraRed Data Association
IWDG	Independent watchdog
LCD	Liquid Crystal Display Controller
LPTIM	Low-power timer
LPUART	Low-power universal asynchronous receiver/transmitter
MCO	Microcontroller clock output
MPU	Memory protection unit
MSP	MCU Specific Package
NAND	NAND Flash memory
NOR	Nor Flash memory
NVIC	Nested Vectored Interrupt Controller

Acronym	Definition
OPAMP	Operational amplifier
OTG-FS	USB on-the-go full-speed
PCD	USB Peripheral Controller Driver
PWR	Power controller
QSPI	QuadSPI Flash memory
RCC	Reset and clock controller
RNG	Random number generator
RTC	Real-time clock
SAI	Serial audio interface
SD	Secure Digital
SDMMC	SD/SDIO/MultiMediaCard card host interface
SRAM	SRAM external memory
SMARTCARD	Smartcard IC
SPI	Serial Peripheral interface
SWPMI	Serial Wire Protocol master interface
SysTick	System tick timer
TIM	Advanced-control, general-purpose or basic timer
TSC	Touch sensing controller
UART	Universal asynchronous receiver/transmitter
USART	Universal synchronous receiver/transmitter
WWDG	Window watchdog
USB	Universal Serial Bus
PPP	STM32 peripheral or block

2 Overview of HAL drivers

The HAL drivers are designed to offer a rich set of APIs and to interact easily with the application upper layers.

Each driver consists of a set of functions covering the most common peripheral features. The development of each driver is driven by a common API which standardizes the driver structure, the functions and the parameter names.

The HAL drivers include a set of driver modules, each module being linked to a standalone peripheral. However, in some cases, the module is linked to a peripheral functional mode. As an example, several modules exist for the USART peripheral: UART driver module, USART driver module, SMARTCARD driver module and IRDA driver module.

The HAL main features are the following:

- Cross-family portable set of APIs covering the common peripheral features as well as extension APIs in case of specific peripheral features.
- Three API programming models: polling, interrupt and DMA.
- APIs are RTOS compliant:
 - Fully reentrant APIs
 - Systematic usage of timeouts in polling mode.
- Support of peripheral multi-instance allowing concurrent API calls for multiple instances of a given peripheral (USART1, USART2...)
- All HAL APIs implement user-callback functions mechanism:
 - Peripheral Init/DeInit HAL APIs can call user-callback functions to perform peripheral system level Initialization/De-Initialization (clock, GPIOs, interrupt, DMA)
 - Peripherals interrupt events
 - Error events.
- Object locking mechanism: safe hardware access to prevent multiple spurious accesses to shared resources.
- Timeout used for all blocking processes: the timeout can be a simple counter or a timebase.

2.1 HAL and user-application files

2.1.1 HAL driver files

A HAL drivers are composed of the following set of files:

Table 2: HAL driver files

File	Description
<i>stm32l4xx_hal_ppp.c</i>	Main peripheral/module driver file. It includes the APIs that are common to all STM32 devices. <i>Example: stm32l4xx_hal_adc.c, stm32l4xx_hal_irda.c, ...</i>
<i>stm32l4xx_hal_ppp.h</i>	Header file of the main driver C file It includes common data, handle and enumeration structures, define statements and macros, as well as the exported generic APIs. <i>Example: stm32l4xx_hal_adc.h, stm32l4xx_hal_irda.h, ...</i>

File	Description
<i>stm32l4xx_hal_ppp_ex.c</i>	Extension file of a peripheral/module driver. It includes the specific APIs for a given part number or family, as well as the newly defined APIs that overwrite the default generic APIs if the internal process is implemented in different way. <i>Example: stm32l4xx_hal_adc_ex.c, stm32l4xx_hal_flash_ex.c, ...</i>
<i>stm32l4xx_hal_ppp_ex.h</i>	Header file of the extension C file. It includes the specific data and enumeration structures, define statements and macros, as well as the exported device part number specific APIs <i>Example: stm32l4xx_hal_adc_ex.h, stm32l4xx_hal_flash_ex.h, ...</i>
<i>stm32l4xx_hal.c</i>	This file is used for HAL initialization and contains DBGMCU, Remap and Time Delay based on SysTick APIs.
<i>stm32l4xx_hal.h</i>	<i>stm32l4xx_hal.c</i> header file
<i>stm32l4xx_hal_msp_template.c</i>	Template file to be copied to the user application folder. It contains the MSP initialization and de-initialization (main routine and callbacks) of the peripheral used in the user application.
<i>stm32l4xx_hal_conf_template.h</i>	Template file allowing to customize the drivers for a given application.
<i>stm32l4xx_hal_def.h</i>	Common HAL resources such as common define statements, enumerations, structures and macros.

2.1.2 User-application files

The minimum files required to build an application using the HAL are listed in the table below:

Table 3: User-application files

File	Description
<i>system_stm32l4xx.c</i>	This file contains SystemInit() which is called at startup just after reset and before branching to the main program. It does not configure the system clock at startup (contrary to the standard library). This is to be done using the HAL APIs in the user files. It allows relocating the vector table in internal SRAM.
<i>startup_stm32l4xx.s</i>	Toolchain specific file that contains reset handler and exception vectors. For some toolchains, it allows adapting the stack/heap size to fit the application requirements.
<i>stm32l4xx_flash.icf</i> (optional)	Linker file for EWARM toolchain allowing mainly adapting the stack/heap size to fit the application requirements.
<i>stm32l4xx_hal_msp.c</i>	This file contains the MSP initialization and de-initialization (main routine and callbacks) of the peripheral used in the user application.
<i>stm32l4xx_hal_conf.h</i>	This file allows the user to customize the HAL drivers for a specific application. It is not mandatory to modify this configuration. The application can use the default configuration without any modification.

File	Description
<i>stm32l4xx_it.c/.h</i>	This file contains the exceptions handler and peripherals interrupt service routine, and calls HAL_IncTick() at regular time intervals to increment a local variable (declared in <i>stm32l4xx_hal.c</i>) used as HAL timebase. By default, this function is called each 1ms in SysTick ISR. . The PPP_IRQHandler() routine must call HAL_PPP_IRQHandler() if an interrupt based process is used within the application.
<i>main.c/.h</i>	This file contains the main program routine, mainly: - the call to HAL_Init() - assert_failed() implementation - system clock configuration - peripheral HAL initialization and user application code.

The STM32Cube package comes with ready-to-use project templates, one for each supported board. Each project contains the files listed above and a preconfigured project for the supported toolchains.

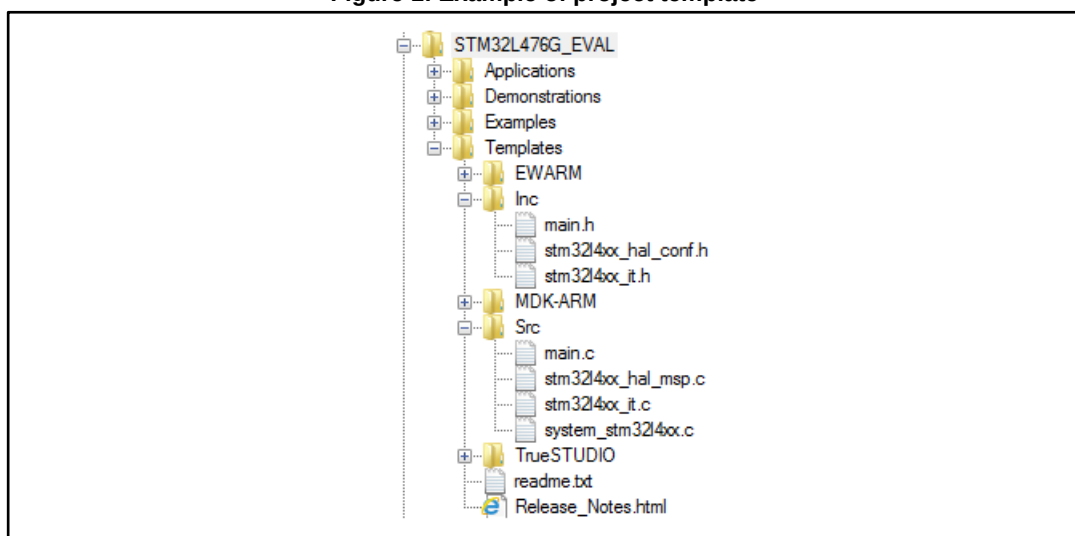
Each project template provides empty main loop function and can be used as a starting point to get familiar with project settings for STM32Cube. Its features are the following:

- It contains the sources of HAL, CMSIS and BSP drivers which are the minimal components to develop a code on a given board.
- It contains the include paths for all the firmware components.
- It defines the STM32 device supported, and allows configuring the CMSIS and HAL drivers accordingly.
- It provides ready to use user files preconfigured as defined below:
 - HAL is initialized
 - SysTick ISR implemented for HAL_GetTick()
 - System clock configured with the selected device frequency.



If an existing project is copied to another location, then include paths must be updated.

Figure 1: Example of project template



2.2 HAL data structures

Each HAL driver can contain the following data structures:

- Peripheral handle structures
- Initialization and configuration structures
- Specific process structures.

2.2.1 Peripheral handle structures

The APIs have a modular generic multi-instance architecture that allows working with several IP instances simultaneously.

PPP_HandleTypeDef *handle is the main structure that is implemented in the HAL drivers. It handles the peripheral/module configuration and registers and embeds all the structures and variables needed to follow the peripheral device flow.

The peripheral handle is used for the following purposes:

- Multi instance support: each peripheral/module instance has its own handle. As a result instance resources are independent.
- Peripheral process intercommunication: the handle is used to manage shared data resources between the process routines.
Example: global pointers, DMA handles, state machine.
- Storage: this handle is used also to manage global variables within a given HAL driver.

An example of peripheral structure is shown below:

```
typedef struct
{
  USART_TypeDef *Instance; /* USART registers base address */
  USART_InitTypeDef Init; /* Usart communication parameters */
  uint8_t *pTxBuffPtr; /* Pointer to Usart Tx transfer Buffer */
  uint16_t TxXferSize; /* Usart Tx Transfer size */
  __IO uint16_t TxXferCount; /* Usart Tx Transfer Counter */
  uint8_t *pRxBuffPtr; /* Pointer to Usart Rx transfer Buffer */
  uint16_t RxXferSize; /* Usart Rx Transfer size */
  __IO uint16_t RxXferCount; /* Usart Rx Transfer Counter */
  DMA_HandleTypeDef *hdmatx; /* Usart Tx DMA Handle parameters */
  DMA_HandleTypeDef *hdmarx; /* Usart Rx DMA Handle parameters */
  HAL_LockTypeDef Lock; /* Locking object */
  __IO HAL_USART_StateTypeDef State; /* Usart communication state */
  __IO HAL_USART_ErrorTypeDef ErrorCode; /* USART Error code */
}USART_HandleTypeDef;
```



1) The multi-instance feature implies that all the APIs used in the application are re-entrant and avoid using global variables because subroutines can fail to be re-entrant if they rely on a global variable to remain unchanged but that variable is modified when the subroutine is recursively invoked. For this reason, the following rules are respected:

- Re-entrant code does not hold any static (or global) non-constant data: re-entrant functions can work with global data. For example, a re-entrant interrupt service routine can grab a piece of hardware status to work with (e.g. serial port read buffer) which is not only global, but volatile. Still, typical use of static variables and global data is not advised, in the sense that only atomic read-modify-write instructions should be used in these variables. It should not be possible for an interrupt or signal to occur during the execution of such an instruction.
- Reentrant code does not modify its own code.



2) When a peripheral can manage several processes simultaneously using the DMA (full duplex case), the DMA interface handle for each process is added in the PPP_HandleTypeDef.



3) For the shared and system peripherals, no handle or instance object is used. The peripherals concerned by this exception are the following:

- GPIO
- SYSTICK
- NVIC
- PWR
- RCC
- FLASH.

2.2.2 Initialization and configuration structure

These structures are defined in the generic driver header file when it is common to all part numbers. When they can change from one part number to another, the structures are defined in the extension header file for each part number.

```
typedef struct
{
  uint32_t BaudRate; /*!< This member configures the UART communication baudrate.*/
  uint32_t WordLength; /*!< Specifies the number of data bits transmitted or received
  in a frame.*/
  uint32_t StopBits; /*!< Specifies the number of stop bits transmitted.*/
  uint32_t Parity; /*!< Specifies the parity mode. */
  uint32_t Mode; /*!< Specifies whether the Receive or Transmit mode is enabled or
  disabled.*/
  uint32_t HwFlowCtl; /*!< Specifies whether the hardware flow control mode is enabled
  or disabled.*/
  uint32_t OverSampling; /*!< Specifies whether the Over sampling 8 is enabled or
  disabled,
  to achieve higher speed (up to fPCLK/8).*/
}UART_InitTypeDef;
```



The config structure is used to initialize the sub-modules or sub-instances. See below example:

```
HAL_ADC_ConfigChannel (ADC_HandleTypeDef* hadc, ADC_ChannelConfTypeDef*
sConfig)
```

2.2.3 Specific process structures

The specific process structures are used for specific process (common APIs). They are defined in the generic driver header file.

Example:

```
HAL_PPP_Process (PPP_HandleTypeDef* hadc, PPP_ProcessConfig* sConfig)
```

2.3 API classification

The HAL APIs are classified into three categories:

- **Generic APIs:** common generic APIs applying to all STM32 devices. These APIs are consequently present in the generic HAL driver files of all STM32 microcontrollers.

```
HAL_StatusTypeDef HAL_ADC_Init(ADC_HandleTypeDef* hadc); HAL_StatusTypeDef
HAL_ADC_DeInit(ADC_HandleTypeDef *hadc); HAL_StatusTypeDef
HAL_ADC_Start(ADC_HandleTypeDef* hadc); HAL_StatusTypeDef
HAL_ADC_Stop(ADC_HandleTypeDef* hadc); HAL_StatusTypeDef
HAL_ADC_Start_IT(ADC_HandleTypeDef* hadc); HAL_StatusTypeDef
HAL_ADC_Stop_IT(ADC_HandleTypeDef* hadc); void HAL_ADC_IRQHandler(ADC_HandleTypeDef*
hadc);
```

- **Extension APIs:** This set of API is divided into two sub-categories:
 - **Family specific APIs:** APIs applying to a given family. They are located in the extension HAL driver file (see example below related to the ADC).

```
HAL_StatusTypeDef HAL_ADCEX_Calibration_Start(ADC_HandleTypeDef* hadc, uint32_t
SingleDiff);
uint32_t HAL_ADCEX_Calibration_GetValue(ADC_HandleTypeDef* hadc, uint32_t
SingleDiff);
```

- **Device part number specific APIs:** These APIs are implemented in the extension file and delimited by specific define statements relative to a given part number.

```
#if defined(STM32L475xx) || defined(STM32L476xx) || defined(STM32L486xx)
void HAL_PWREx_EnableVddUSB(void);
void HAL_PWREx_DisableVddUSB(void);
#endif /* STM32L475xx || STM32L476xx || STM32L486xx */
```



The data structure related to the specific APIs is delimited by the device part number define statement. It is located in the corresponding extension header C file.

The following table summarizes the location of the different categories of HAL APIs in the driver files.

Table 4: API classification

	Generic file	Extension file
Common APIs	X	X
Family specific APIs		X
Device specific APIs		X



Family specific APIs are only related to a given family. This means that if a specific API is implemented in another family, and the arguments of this latter family are different, additional structures and arguments might need to be added.



The IRQ handlers are used for common and family specific processes.

2.4 Devices supported by HAL drivers

Table 5: List of STM32L4 Series devices supported by HAL drivers

IP/ Module	STM32L431xx	STM32L432xx	STM32L442xx	STM32L433xx	STM32L443xx	STM32L451xx	STM32L452xx	STM32L462xx	STM32L471xx	STM32L475xx	STM32L476xx	STM32L485xx	STM32L486xx	STM32L496xx	STM32L4A6xx
stm32l4xx_hal.c	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
stm32l4xx_hal_adc.c	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
stm32l4xx_hal_adc_ex.c	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
stm32l4xx_hal_can.c	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
stm32l4xx_hal_comp.c	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
stm32l4xx_hal_cortex.c	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
stm32l4xx_hal_crc.c	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
stm32l4xx_hal_crc_ex.c	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
stm32l4xx_hal_cryp.c	No	No	Yes	No	Yes	No	No	Yes	No	No	No	Yes	Yes	No	Yes
stm32l4xx_hal_cryp_ex.c	No	No	Yes	No	Yes	No	No	Yes	No	No	No	Yes	Yes	No	Yes
stm32l4xx_hal_dac.c	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
stm32l4xx_hal_dac_ex.c	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
stm32l4xx_hal_dcmi.c	No	No	No	No	No	No	No	No	No	No	No	No	No	Yes	Yes
stm32l4xx_hal_dfstm.c	No	No	No	No	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
stm32l4xx_hal_dfstm_ex.c	No	No	No	No	No	No	No	No	No	No	No	No	No	No	No
stm32l4xx_hal_dma.c	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
stm32l4xx_hal_dma_ex.c	No	No	No	No	No	No	No	No	No	No	No	No	No	No	No
stm32l4xx_hal_dma2d.c	No	No	No	No	No	No	No	No	No	No	No	No	No	Yes	Yes
stm32l4xx_hal_dsi.c	No	No	No	No	No	No	No	No	No	No	No	No	No	No	No
stm32l4xx_hal_firewall.c	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
stm32l4xx_hal_flash.c	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
stm32l4xx_hal_flash_ex.c	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes

IP/ Module	STM32L431xx	STM32L432xx	STM32L442xx	STM32L433xx	STM32L443xx	STM32L451xx	STM32L452xx	STM32L462xx	STM32L471xx	STM32L475xx	STM32L476xx	STM32L485xx	STM32L486xx	STM32L496xx	STM32L4A6xx
stm32l4xx_hal_flash_ramfunc.c	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
stm32l4xx_hal_gfxmmu.c	No	No	No	No	No	No	No	No	No	No	No	No	No	No	No
stm32l4xx_hal_gpio.c	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
stm32l4xx_hal_hash.c	No	No	No	No	No	No	No	No	No	No	No	No	No	No	Yes
stm32l4xx_hal_hash_ex.c	No	No	No	No	No	No	No	No	No	No	No	No	No	No	Yes
stm32l4xx_hal_hcd.c	No	No	No	No	No	No	No	No	No	Yes	Yes	Yes	Yes	Yes	Yes
stm32l4xx_hal_i2c.c	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
stm32l4xx_hal_i2c_ex.c	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
stm32l4xx_hal_irda.c	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
stm32l4xx_hal_iwdg.c	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
stm32l4xx_hal_lcd.c	No	No	No	Yes	Yes	No	No	No	No	No	Yes	No	Yes	Yes	Yes
stm32l4xx_hal_lptim.c	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
stm32l4xx_hal_ltdc.c	No	No	No	No	No	No	No	No	No	No	No	No	No	No	No
stm32l4xx_hal_ltdc_ex.c	No	No	No	No	No	No	No	No	No	No	No	No	No	No	No
stm32l4xx_hal_msp_template.c	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA
stm32l4xx_hal_nand.c	No	No	No	No	No	No	No	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes
stm32l4xx_hal_nor.c	No	No	No	No	No	No	No	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes
stm32l4xx_hal_opamp.c	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
stm32l4xx_hal_opamp_ex.c	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
stm32l4xx_hal_ospi.c	No	No	No	No	No	No	No	No	No	No	No	No	No	No	No
stm32l4xx_hal_pcd.c	No	Yes	Yes	Yes	Yes	No	Yes	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes
stm32l4xx_hal_pcd_ex.c	No	Yes	Yes	Yes	Yes	No	Yes	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes
stm32l4xx_hal_pwr.c	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes

IP/ Module	STM32L431xx	STM32L432xx	STM32L442xx	STM32L433xx	STM32L443xx	STM32L451xx	STM32L452xx	STM32L462xx	STM32L471xx	STM32L475xx	STM32L476xx	STM32L485xx	STM32L486xx	STM32L496xx	STM32L4A6xx
stm32l4xx_hal_pwr_ex.c	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
stm32l4xx_hal_qspi.c	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
stm32l4xx_hal_rcc.c	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
stm32l4xx_hal_rcc_ex.c	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
stm32l4xx_hal_rng.c	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
stm32l4xx_hal_rtc.c	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
stm32l4xx_hal_rtc_ex.c	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
stm32l4xx_hal_sai.c	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
stm32l4xx_hal_sai_ex.c	No	No	No	No	No	No	No	No	No	No	No	No	No	No	No
stm32l4xx_hal_sd.c	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
stm32l4xx_hal_sd_ex.c	No	No	No	No	No	No	No	No	No	No	No	No	No	No	No
stm32l4xx_hal_smartcard.c	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
stm32l4xx_hal_smartcard_ex.c	No	No	No	No	No	No	No	No	No	No	No	No	No	No	No
stm32l4xx_hal_smbus.c	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
stm32l4xx_hal_spi.c	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
stm32l4xx_hal_spi_ex.c	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
stm32l4xx_hal_sram.c	No	No	No	No	No	No	No	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes
stm32l4xx_hal_swpmi.c	Yes	Yes	Yes	Yes	Yes	No	No	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes
stm32l4xx_hal_tim.c	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
stm32l4xx_hal_tim_ex.c	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
stm32l4xx_hal_tsc.c	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
stm32l4xx_hal_uart.c	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
stm32l4xx_hal_uart_ex.c	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes

IP/ Module	STM32L431xx	STM32L432xx	STM32L442xx	STM32L433xx	STM32L443xx	STM32L451xx	STM32L452xx	STM32L462xx	STM32L471xx	STM32L475xx	STM32L476xx	STM32L485xx	STM32L486xx	STM32L496xx	STM32L4A6xx
stm32l4xx_hal_usart.c	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
stm32l4xx_hal_usart_ex.c	No	No	No	No	No	No	No	No	No	No	No	No	No	No	No
stm32l4xx_hal_wwdg.c	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes

Table 6: List of STM32L4+ Series devices supported by HAL drivers

IP/Module	STM32L4R5xx	STM32L4R7xx	STM32L4R9xx	STM32L4S5xx	STM32L4S7xx	STM32L4S9xx
stm32l4xx_hal.c	Yes	Yes	Yes	Yes	Yes	Yes
stm32l4xx_hal_adc.c	Yes	Yes	Yes	Yes	Yes	Yes
stm32l4xx_hal_adc_ex.c	Yes	Yes	Yes	Yes	Yes	Yes
stm32l4xx_hal_can.c	Yes	Yes	Yes	Yes	Yes	Yes
stm32l4xx_hal_comp.c	Yes	Yes	Yes	Yes	Yes	Yes
stm32l4xx_hal_cortex.c	Yes	Yes	Yes	Yes	Yes	Yes
stm32l4xx_hal_crc.c	Yes	Yes	Yes	Yes	Yes	Yes
stm32l4xx_hal_crc_ex.c	Yes	Yes	Yes	Yes	Yes	Yes
stm32l4xx_hal_cryp.c	No	No	No	Yes	Yes	Yes
stm32l4xx_hal_cryp_ex.c	No	No	No	Yes	Yes	Yes
stm32l4xx_hal_dac.c	Yes	Yes	Yes	Yes	Yes	Yes
stm32l4xx_hal_dac_ex.c	Yes	Yes	Yes	Yes	Yes	Yes
stm32l4xx_hal_dcmi.c	Yes	Yes	Yes	Yes	Yes	Yes
stm32l4xx_hal_dfsdm.c	Yes	Yes	Yes	Yes	Yes	Yes
stm32l4xx_hal_dfsdm_ex.c	Yes	Yes	Yes	Yes	Yes	Yes
stm32l4xx_hal_dma.c	Yes	Yes	Yes	Yes	Yes	Yes
stm32l4xx_hal_dma_ex.c	Yes	Yes	Yes	Yes	Yes	Yes
stm32l4xx_hal_dma2d.c	Yes	Yes	Yes	Yes	Yes	Yes
stm32l4xx_hal_dsi.c	No	No	Yes	No	No	Yes
stm32l4xx_hal_firewall.c	Yes	Yes	Yes	Yes	Yes	Yes
stm32l4xx_hal_flash.c	Yes	Yes	Yes	Yes	Yes	Yes
stm32l4xx_hal_flash_ex.c	Yes	Yes	Yes	Yes	Yes	Yes
stm32l4xx_hal_flash_ramfunc.c	Yes	Yes	Yes	Yes	Yes	Yes
stm32l4xx_hal_gfxmmu.c	No	Yes	Yes	No	Yes	Yes
stm32l4xx_hal_gpio.c	Yes	Yes	Yes	Yes	Yes	Yes

IP/Module	STM32L4R5xx	STM32L4R7xx	STM32L4R9xx	STM32L4S5xx	STM32L4S7xx	STM32L4S9xx
stm32l4xx_hal_hash.c	No	No	No	Yes	Yes	Yes
stm32l4xx_hal_hash_ex.c	No	No	No	Yes	Yes	Yes
stm32l4xx_hal_hcd.c	Yes	Yes	Yes	Yes	Yes	Yes
stm32l4xx_hal_i2c.c	Yes	Yes	Yes	Yes	Yes	Yes
stm32l4xx_hal_i2c_ex.c	Yes	Yes	Yes	Yes	Yes	Yes
stm32l4xx_hal_irda.c	Yes	Yes	Yes	Yes	Yes	Yes
stm32l4xx_hal_iwdg.c	Yes	Yes	Yes	Yes	Yes	Yes
stm32l4xx_hal_lcd.c	No	No	No	No	No	No
stm32l4xx_hal_lptim.c	Yes	Yes	Yes	Yes	Yes	Yes
stm32l4xx_hal_itdc.c	No	Yes	Yes	No	Yes	Yes
stm32l4xx_hal_itdc_ex.c	No	Yes	Yes	No	Yes	Yes
stm32l4xx_hal_msp_template.c	NA	NA	NA	NA	NA	NA
stm32l4xx_hal_nand.c	Yes	Yes	Yes	Yes	Yes	Yes
stm32l4xx_hal_nor.c	Yes	Yes	Yes	Yes	Yes	Yes
stm32l4xx_hal_opamp.c	Yes	Yes	Yes	Yes	Yes	Yes
stm32l4xx_hal_opamp_ex.c	Yes	Yes	Yes	Yes	Yes	Yes
stm32l4xx_hal_ospi.c	Yes	Yes	Yes	Yes	Yes	Yes
stm32l4xx_hal_pcd.c	Yes	Yes	Yes	Yes	Yes	Yes
stm32l4xx_hal_pcd_ex.c	Yes	Yes	Yes	Yes	Yes	Yes
stm32l4xx_hal_pwr.c	Yes	Yes	Yes	Yes	Yes	Yes
stm32l4xx_hal_pwr_ex.c	Yes	Yes	Yes	Yes	Yes	Yes
stm32l4xx_hal_qspi.c	No	No	No	No	No	No
stm32l4xx_hal_rcc.c	Yes	Yes	Yes	Yes	Yes	Yes
stm32l4xx_hal_rcc_ex.c	Yes	Yes	Yes	Yes	Yes	Yes
stm32l4xx_hal_rng.c	Yes	Yes	Yes	Yes	Yes	Yes
stm32l4xx_hal_rtc.c	Yes	Yes	Yes	Yes	Yes	Yes
stm32l4xx_hal_rtc_ex.c	Yes	Yes	Yes	Yes	Yes	Yes
stm32l4xx_hal_sai.c	Yes	Yes	Yes	Yes	Yes	Yes
stm32l4xx_hal_sai_ex.c	Yes	Yes	Yes	Yes	Yes	Yes
stm32l4xx_hal_sd.c	Yes	Yes	Yes	Yes	Yes	Yes
stm32l4xx_hal_sd_ex.c	Yes	Yes	Yes	Yes	Yes	Yes
stm32l4xx_hal_smartcard.c	Yes	Yes	Yes	Yes	Yes	Yes
stm32l4xx_hal_smartcard_ex.c	Yes	Yes	Yes	Yes	Yes	Yes
stm32l4xx_hal_smbus.c	Yes	Yes	Yes	Yes	Yes	Yes
stm32l4xx_hal_spi.c	Yes	Yes	Yes	Yes	Yes	Yes
stm32l4xx_hal_spi_ex.c	Yes	Yes	Yes	Yes	Yes	Yes

IP/Module	STM32L4R5xx	STM32L4R7xx	STM32L4R9xx	STM32L4S5xx	STM32L4S7xx	STM32L4S9xx
stm32l4xx_hal_sram.c	Yes	Yes	Yes	Yes	Yes	Yes
stm32l4xx_hal_swpmi.c	No	No	No	No	No	No
stm32l4xx_hal_tim.c	Yes	Yes	Yes	Yes	Yes	Yes
stm32l4xx_hal_tim_ex.c	Yes	Yes	Yes	Yes	Yes	Yes
stm32l4xx_hal_tsc.c	Yes	Yes	Yes	Yes	Yes	Yes
stm32l4xx_hal_uart.c	Yes	Yes	Yes	Yes	Yes	Yes
stm32l4xx_hal_uart_ex.c	Yes	Yes	Yes	Yes	Yes	Yes
stm32l4xx_hal_usart.c	Yes	Yes	Yes	Yes	Yes	Yes
stm32l4xx_hal_usart_ex.c	Yes	Yes	Yes	Yes	Yes	Yes
stm32l4xx_hal_wwdg.c	Yes	Yes	Yes	Yes	Yes	Yes

2.5 HAL driver rules

2.5.1 HAL API naming rules

The following naming rules are used in HAL drivers:

Table 7: HAL API naming rules

	Generic	Family specific	Device specific
File names	<i>stm32l4xx_hal_ppp (c/h)</i>	<i>stm32l4xx_hal_ppp_ex (c/h)</i>	<i>stm32l4xx_hal_ppp_ex (c/h)</i>
Module name	<i>HAL_PPP_MODULE</i>		
Function name	<i>HAL_PPP_Function</i> <i>HAL_PPP_FeatureFunction_MODE</i>	<i>HAL_PPPEX_Function</i> <i>HAL_PPPEX_FeatureFunction_MODE</i>	<i>HAL_PPPEX_Function</i> <i>HAL_PPPEX_FeatureFunction_MODE</i>
Handle name	<i>PPP_HandleTypeDef</i>	NA	NA
Init structure name	<i>PPP_InitTypeDef</i>	NA	<i>PPP_InitTypeDef</i>
Enum name	<i>HAL_PPP_StructnameTypeDef</i>	NA	NA

- The **PPP** prefix refers to the peripheral functional mode and not to the peripheral itself. For example, if the USART, PPP can be USART, IRDA, UART or SMARTCARD depending on the peripheral mode.
- The constants used in one file are defined within this file. A constant used in several files is defined in a header file. All constants are written in uppercase, except for peripheral driver function parameters.
- typedef variable names should be suffixed with *_TypeDef*.

- Registers are considered as constants. In most cases, their name is in uppercase and uses the same acronyms as in the STM32L4 series and STM32L4+ series reference manuals.
- Peripheral registers are declared in the PPP_TypeDef structure (e.g. ADC_TypeDef) in stm32l4xxx.h header file. stm32l4xxx.h corresponds to stm32l471xx.h, stm32l475xx.h, stm32l476xx.h, stm32l485xx.h and stm32l486xx.h.
- Peripheral function names are prefixed by HAL_, then the corresponding peripheral acronym in uppercase followed by an underscore. The first letter of each word is in uppercase (e.g. HAL_UART_Transmit()). Only one underscore is allowed in a function name to separate the peripheral acronym from the rest of the function name.
- The structure containing the PPP peripheral initialization parameters are named PPP_InitTypeDef (e.g. ADC_InitTypeDef).
- The structure containing the Specific configuration parameters for the PPP peripheral are named PPP_xxxxConfTypeDef (e.g. ADC_ChannelConfTypeDef).
- Peripheral handle structures are named PPP_HandleTypeDef (e.g. DMA_HandleTypeDef).
- The functions used to initialize the PPP peripheral according to parameters specified in PPP_InitTypeDef are named HAL_PPP_Init (e.g. HAL_TIM_Init()).
- The functions used to reset the PPP peripheral registers to their default values are named HAL_PPP_DeInit (e.g. HAL_TIM_DeInit()).
- The **MODE** suffix refers to the process mode, which can be polling, interrupt or DMA. As an example, when the DMA is used in addition to the native resources, the function should be called: *HAL_PPP_Function_DMA()*.
- The **Feature** prefix should refer to the new feature.
Example: *HAL_ADCEx_InjectedStart()* refers to the injection mode

2.5.2 HAL general naming rules

- For the shared and system peripherals, no handle or instance object is used. This rule applies to the following peripherals:
 - GPIO
 - SYSTICK
 - NVIC
 - RCC
 - FLASH.

Example: The *HAL_GPIO_Init()* requires only the GPIO address and its configuration parameters.

```
HAL_StatusTypeDef HAL_GPIO_Init (GPIO_TypeDef* GPIOx, GPIO_InitTypeDef *Init)
{
  /*GPIO Initialization body */
}
```

- The macros that handle interrupts and specific clock configurations are defined in each peripheral/module driver. These macros are exported in the peripheral driver header files so that they can be used by the extension file. The list of these macros is defined below: This list is not exhaustive and other macros related to peripheral features can be added, so that they can be used in the user application.

Table 8: Macros handling interrupts and specific clock configurations

Macros	Description
<code>__HAL_PPP_ENABLE_IT(__HANDLE__, __INTERRUPT__)</code>	Enables a specific peripheral interrupt
<code>__HAL_PPP_DISABLE_IT(__HANDLE__, __INTERRUPT__)</code>	Disables a specific peripheral interrupt
<code>__HAL_PPP_GET_IT (__HANDLE__, __INTERRUPT__)</code>	Gets a specific peripheral interrupt status
<code>__HAL_PPP_CLEAR_IT (__HANDLE__, __INTERRUPT__)</code>	Clears a specific peripheral interrupt status
<code>__HAL_PPP_GET_FLAG (__HANDLE__, __FLAG__)</code>	Gets a specific peripheral flag status
<code>__HAL_PPP_CLEAR_FLAG (__HANDLE__, __FLAG__)</code>	Clears a specific peripheral flag status
<code>__HAL_PPP_ENABLE(__HANDLE__)</code>	Enables a peripheral
<code>__HAL_PPP_DISABLE(__HANDLE__)</code>	Disables a peripheral
<code>__HAL_PPP_XXXX (__HANDLE__, __PARAM__)</code>	Specific PPP HAL driver macro
<code>__HAL_PPP_GET_IT_SOURCE (__HANDLE__, __INTERRUPT__)</code>	Checks the source of specified interrupt

- NVIC and SYSTICK are two Arm Cortex core features. The APIs related to these features are located in the `stm32l4xx_hal_cortex.c` file.
- When a status bit or a flag is read from registers, it is composed of shifted values depending on the number of read values and of their size. In this case, the returned status width is 32 bits. Example: `STATUS = XX | (YY << 16)` or `STATUS = XX | (YY << 8) | (YY << 16) | (YY << 24)`.
- The PPP handles are valid before using the `HAL_PPP_Init()` API. The init function performs a check before modifying the handle fields.

```
HAL_PPP_Init(PPP_HandleTypeDef) if(hppp == NULL) { return HAL_ERROR; }
```

- The macros defined below are used:
 - Conditional macro:

```
#define ABS(x) ((x) > 0) ? (x) : -(x)
```

- Pseudo-code macro (multiple instructions macro):

```
#define __HAL_LINKDMA(__HANDLE__, __PPP_DMA_FIELD__, __DMA_HANDLE__) \
do{ \ ( __HANDLE__ )-> PPP_DMA_FIELD = &( __DMA_HANDLE__ ); \ \
( __DMA_HANDLE__ ).Parent = ( __HANDLE__ ); \ \
} while(0)
```

2.5.3 HAL interrupt handler and callback functions

Besides the APIs, HAL peripheral drivers include:

- HAL_PPP_IRQHandler() peripheral interrupt handler that should be called from stm32l4xx_it.c
- User callback functions.

The user callback functions are defined as empty functions with “weak” attribute. They have to be defined in the user code.

There are three types of user callbacks functions:

- Peripheral system level initialization/ de-Initialization callbacks: HAL_PPP_MspInit() and HAL_PPP_MspDeInit
- Process complete callbacks: HAL_PPP_ProcessCpltCallback
- Error callback: HAL_PPP_ErrorCallback.

Table 9: Callback functions

Callback functions	Example
HAL_PPP_MspInit() / _DeInit()	Ex: HAL_USART_MspInit() Called from HAL_PPP_Init() API function to perform peripheral system level initialization (GPIOs, clock, DMA, interrupt)
HAL_PPP_ProcessCpltCallback	Ex: HAL_USART_TxCpltCallback Called by peripheral or DMA interrupt handler when the process completes
HAL_PPP_ErrorCallback	Ex: HAL_USART_ErrorCallback Called by peripheral or DMA interrupt handler when an error occurs

2.6 HAL generic APIs

The generic APIs provide common generic functions applying to all STM32 devices. They are composed of four APIs groups:

- **Initialization and de-initialization functions:** HAL_PPP_Init(), HAL_PPP_DeInit()
- **IO operation functions:** HAL_PPP_Read(), HAL_PPP_Write(), HAL_PPP_Transmit(), HAL_PPP_Receive()
- **Control functions:** HAL_PPP_Set (), HAL_PPP_Get ().
- **State and Errors functions:** HAL_PPP_GetState (), HAL_PPP_GetError ().

For some peripheral/module drivers, these groups are modified depending on the peripheral/module implementation.

Example: in the timer driver, the API grouping is based on timer features (PWM, OC, IC...).

The initialization and de-initialization functions allow initializing a peripheral and configuring the low-level resources, mainly clocks, GPIO, alternate functions (AF) and possibly DMA and interrupts. The *HAL_DeInit()* function restores the peripheral default state, frees the low-level resources and removes any direct dependency with the hardware.

The IO operation functions perform a row access to the peripheral payload data in write and read modes.

The control functions are used to change dynamically the peripheral configuration and set another operating mode.

The peripheral state and errors functions allow retrieving in runtime the peripheral and data flow states, and identifying the type of errors that occurred. The example below is based on the ADC peripheral. The list of generic APIs is not exhaustive. It is only given as an example.

Table 10: HAL generic APIs

Function group	Common API name	Description
<i>Initialization group</i>	<i>HAL_ADC_Init()</i>	This function initializes the peripheral and configures the low-level resources (clocks, GPIO, AF..)
	<i>HAL_ADC_DeInit()</i>	This function restores the peripheral default state, frees the low-level resources and removes any direct dependency with the hardware.
<i>IO operation group</i>	<i>HAL_ADC_Start ()</i>	This function starts ADC conversions when the polling method is used
	<i>HAL_ADC_Stop ()</i>	This function stops ADC conversions when the polling method is used
	<i>HAL_ADC_PollForConversion()</i>	This function allows waiting for the end of conversions when the polling method is used. In this case, a timeout value is specified by the user according to the application.
	<i>HAL_ADC_Start_IT()</i>	This function starts ADC conversions when the interrupt method is used
	<i>HAL_ADC_Stop_IT()</i>	This function stops ADC conversions when the interrupt method is used
	<i>HAL_ADC_IRQHandler()</i>	This function handles ADC interrupt requests
	<i>HAL_ADC_ConvCpltCallback()</i>	Callback function called in the IT subroutine to indicate the end of the current process or when a DMA transfer has completed
	<i>HAL_ADC_ErrorCallback()</i>	Callback function called in the IT subroutine if a peripheral error or a DMA transfer error occurred
<i>Control group</i>	<i>HAL_ADC_ConfigChannel()</i>	This function configures the selected ADC regular channel, the corresponding rank in the sequencer and the sample time
	<i>HAL_ADC_AnalogWDGConfig</i>	This function configures the analog watchdog for the selected ADC
<i>State and Errors group</i>	<i>HAL_ADC_GetState()</i>	This function allows getting in runtime the peripheral and the data flow states.
	<i>HAL_ADC_GetError()</i>	This function allows getting in runtime the error that occurred during IT routine

2.7 HAL extension APIs

2.7.1 HAL extension model overview

The extension APIs provide specific functions or overwrite modified APIs for a specific family (series) or specific part number within the same family.

The extension model consists of an additional file, `stm32l4xx_hal_ppp_ex.c`, that includes all the specific functions and define statements (`stm32l4xx_hal_ppp_ex.h`) for a given part number.

Below an example based on the ADC peripheral:

Table 11: HAL extension APIs

Function Group	Common API Name
<code>HAL_ADCEx_CalibrationStart()</code>	This function is used to start the automatic ADC calibration

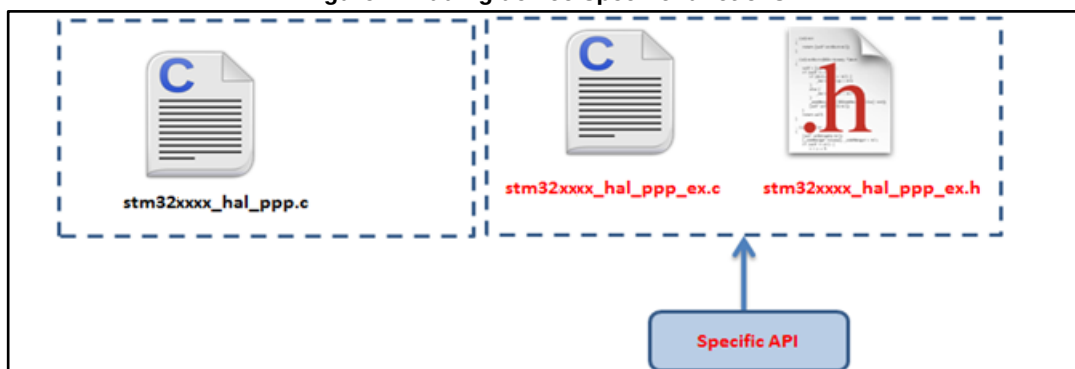
2.7.2 HAL extension model cases

The specific IP features can be handled by the HAL drivers in five different ways. They are described below.

Case 1: Adding a part number-specific function

When a new feature specific to a given device is required, the new APIs are added in the `stm32l4xx_hal_ppp_ex.c` extension file. They are named `HAL_PPPEX_Function()`.

Figure 2: Adding device-specific functions



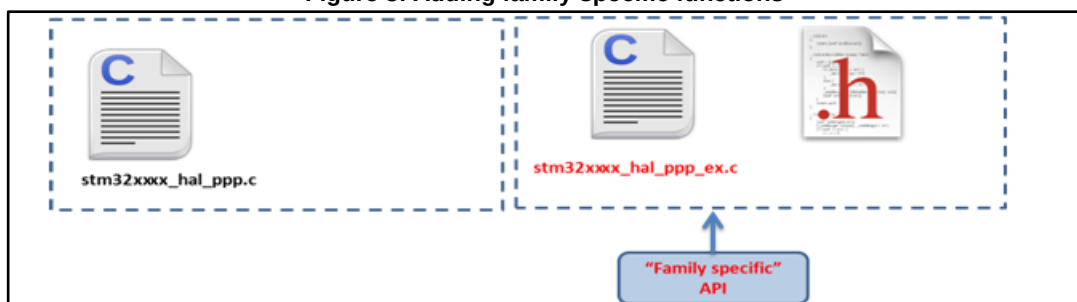
Example: `stm32l4xx_hal_adc_ex.c/h`

```
#if defined(STM32L475xx) || defined(STM32L476xx) || defined(STM32L486xx)
void HAL_PWREx_EnableVddUSB(void);
void HAL_PWREx_DisableVddUSB(void);
#endif /* STM32L475xx || STM32L476xx || STM32L486xx */
```

Case 2: Adding a family-specific function

In this case, the API is added in the extension driver C file and named `HAL_PPPEX_Function()`.

Figure 3: Adding family-specific functions

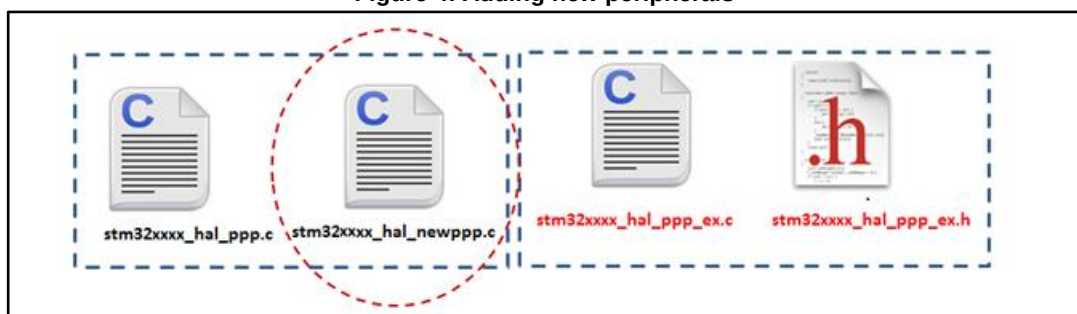


Case 3: Adding a new peripheral (specific to a device belonging to a given family)

When a peripheral which is available only in a specific device is required, the APIs corresponding to this new peripheral/module (newPPP) are added in a new `stm32l4xx_hal_newppp.c`. However the inclusion of this file is selected in the `stm32lxx_hal_conf.h` using the macro:

```
#define HAL_NEWPPP_MODULE_ENABLED
```

Figure 4: Adding new peripherals

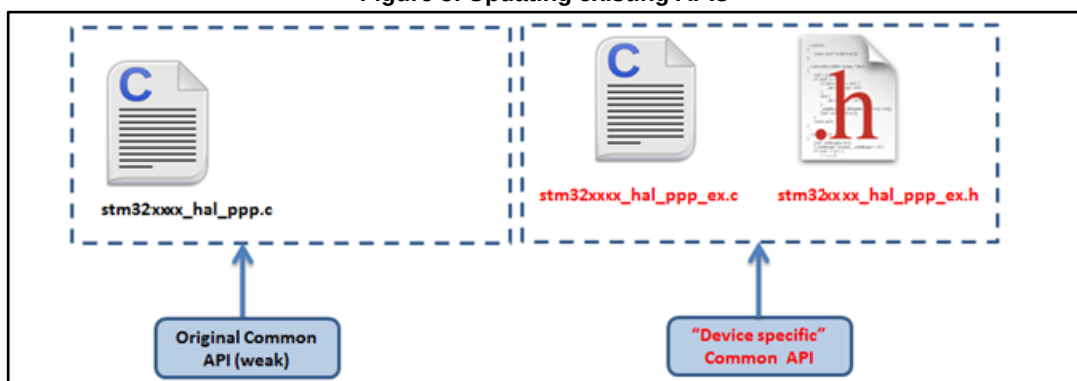


Example: `stm32l4xx_hal_lcd.c/h`

Case 4: Updating existing common APIs

In this case, the routines are defined with the same names in the `stm32l4xx_hal_ppp_ex.c` extension file, while the generic API is defined as *weak*, so that the compiler will overwrite the original routine by the new defined function.

Figure 5: Updating existing APIs



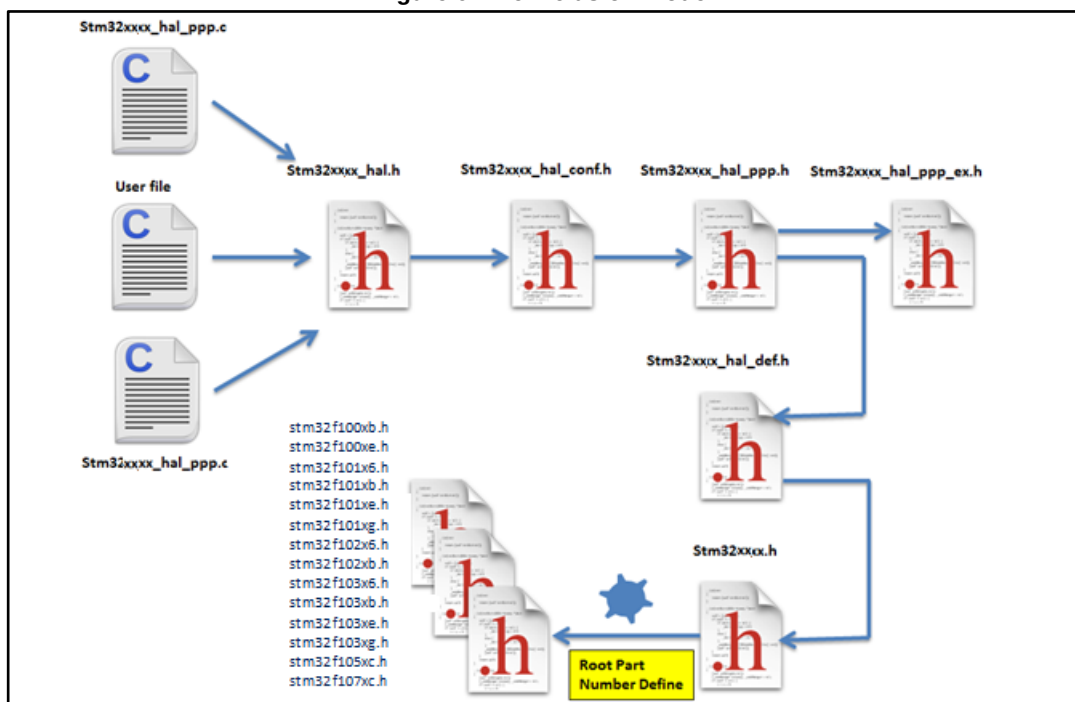
Case 5: Updating existing data structures

The data structure for a specific device part number (e.g. PPP_InitTypeDef) can have different fields. In this case, the data structure is defined in the extension header file and delimited by the specific part number define statement.

2.8 File inclusion model

The header of the common HAL driver file (stm32l4xx_hal.h) includes the common configurations for the whole HAL library. It is the only header file that is included in the user sources and the HAL C sources files to be able to use the HAL resources.

Figure 6: File inclusion model



A PPP driver is a standalone module which is used in a project. The user must enable the corresponding `USE_HAL_PPP_MODULE` define statement in the configuration file.

```

/*****
 * @file stm32l4xx_hal_conf.h
 * @author MCD Application Team
 * @version VX.Y.Z * @date dd-mm-yyyy
 * @brief This file contains the modules to be used
 *****/
(...)
#define HAL_USART_MODULE_ENABLED
#define HAL_IRDA_MODULE_ENABLED
#define HAL_DMA_MODULE_ENABLED
#define HAL_RCC_MODULE_ENABLED
(...)

```

2.9 HAL common resources

The common HAL resources, such as common define enumerations, structures and macros, are defined in *stm32l4xx_hal_def.h*. The main common define enumeration is *HAL_StatusTypeDef*.

- **HAL Status** The HAL status is used by almost all HAL APIs, except for boolean functions and IRQ handler. It returns the status of the current API operations. It has four possible values as described below:

```
typedef enum
{ HAL_OK = 0x00, HAL_ERROR = 0x01, HAL_BUSY = 0x02, HAL_TIMEOUT = 0x03
} HAL_StatusTypeDef;
```

- **HAL Locked** The HAL lock is used by all HAL APIs to prevent accessing by accident shared resources.

```
typedef enum
{ HAL_UNLOCKED = 0x00, /*!<Resources unlocked */
  HAL_LOCKED = 0x01 /*!< Resources locked */
} HAL_LockTypeDef;
```

In addition to common resources, the *stm32l4xx_hal_def.h* file calls the *stm32l4xx.h* file in CMSIS library to get the data structures and the address mapping for all peripherals:

- Declarations of peripheral registers and bits definition.
- Macros to access peripheral registers hardware (Write register, Read register...etc.).

- **Common macros**
 - Macro defining *HAL_MAX_DELAY*

```
#define HAL_MAX_DELAY 0xFFFFFFFF
```

- Macro linking a PPP peripheral to a DMA structure pointer:

```
#define __HAL_LINKDMA(__HANDLE__, __PPP_DMA_FIELD__, __DMA_HANDLE__) \
do{ \
( __HANDLE__ )-> __PPP_DMA_FIELD__ = &( __DMA_HANDLE__ ); \
( __DMA_HANDLE__ ).Parent = ( __HANDLE__ ); \
} while(0)
```

2.10 HAL configuration

The configuration file, *stm32l4xx_hal_conf.h*, allows customizing the drivers for the user application. Modifying this configuration is not mandatory: the application can use the default configuration without any modification.

To configure these parameters, the user should enable, disable or modify some options by uncommenting, commenting or modifying the values of the related define statements as described in the table below:

Table 12: Define statements used for HAL configuration

Configuration item	Description	Default Value
HSE_VALUE	Defines the value of the external oscillator (HSE) expressed in Hz. The user must adjust this define statement when using a different crystal value.	8 000 000 Hz
HSE_STARTUP_TIMEOUT	Timeout for HSE start-up, expressed in ms	100

Configuration item	Description	Default Value
HSI_VALUE	Defines the value of the internal oscillator (HSI) expressed in Hz.	16 000 000 Hz
MSI_VALUE	Defines the default value of the Multiplespeed internal oscillator (MSI) expressed in Hz.	4 000 000 Hz
LSI_VALUE	Defines the default value of the Low-speed internal oscillator (LSI) expressed in Hz.	32000 Hz
LSE_VALUE	Defines the value of the external oscillator (LSE) expressed in Hz. The user must adjust this define statement when using a different crystal value.	32768 Hz
LSE_STARTUP_TIMEOUT	Timeout for LSE start-up, expressed in ms	5000
VDD_VALUE	VDD value	3300 (mV)
USE_RTOS	Enables the use of RTOS	FALSE (for future use)
PREFETCH_ENABLE	Enables prefetch feature	FALSE
INSTRUCTION_CACHE_ENABLE	Enables I-cache feature	TRUE
DATA_CACHE_ENABLE	Enables D-cache feature	TRUE



The `stm32l4xx_hal_conf_template.h` file is located in the HAL drivers *Inc* folder. It should be copied to the user folder, renamed and modified as described above.



By default, the values defined in the `stm32l4xx_hal_conf_template.h` file are the same as the ones used for the examples and demonstrations. All HAL include files are enabled so that they can be used in the user code without modifications.

2.11 HAL system peripheral handling

This chapter gives an overview of how the system peripherals are handled by the HAL drivers. The full API list is provided within each peripheral driver description section.

2.11.1 Clock

Two main functions can be used to configure the system clock:

- `HAL_RCC_OscConfig` (`RCC_OscInitTypeDef *RCC_OscInitStruct`). This function configures/enables multiple clock sources (HSE, HSI, MSI, LSE, LSI, PLL).
- `HAL_RCC_ClockConfig` (`RCC_ClkInitTypeDef *RCC_ClkInitStruct, uint32_t FLatency`). This function
 - selects the system clock source
 - configures AHB, APB1 and APB2 clock dividers
 - configures the number of Flash memory wait states
 - updates the SysTick configuration when HCLK clock changes.

Some peripheral clocks are not derived from the system clock (RTC, USB...). In this case, the clock configuration is performed by an extended API defined in `stm32l4xx_hal_rcc_ex.c`: `HAL_RCCEx_PeriphCLKConfig(RCC_PeriphCLKInitTypeDef *PeriphClkInit)`.

Additional RCC HAL driver functions are available:

- `HAL_RCC_DeInit()` Clock de-initialization function that returns clock configuration to reset state
- Get clock functions that allow retrieving various clock configurations (system clock, HCLK, PCLK1, PCLK2, ...)
- MCO and CSS configuration functions

A set of macros are defined in `stm32l4xx_hal_rcc.h` and `stm32l4xx_hal_rcc_ex.h`. They allow executing elementary operations on RCC block registers, such as peripherals clock gating/reset control:

- `__HAL_PPP_CLK_ENABLE/ __HAL_PPP_CLK_DISABLE` to enable/disable the peripheral clock
- `__HAL_PPP_FORCE_RESET/ __HAL_PPP_RELEASE_RESET` to force/release peripheral reset
- `__HAL_PPP_CLK_SLEEP_ENABLE/ __HAL_PPP_CLK_SLEEP_DISABLE` to enable/disable the peripheral clock during low power (Sleep) mode.
- `__HAL_PPP_IS_CLK_ENABLED/ __HAL_PPP_IS_CLK_DISABLED` to query about the enabled/disabled status of the peripheral clock.
- `__HAL_PPP_IS_CLK_SLEEP_ENABLED/ __HAL_PPP_IS_CLK_SLEEP_DISABLED` to query about the enabled/disabled status of the peripheral clock during low power (Sleep) mode.

2.11.2 GPIOs

GPIO HAL APIs are the following:

- `HAL_GPIO_Init()` / `HAL_GPIO_DeInit()`
- `HAL_GPIO_ReadPin()` / `HAL_GPIO_WritePin()`
- `HAL_GPIO_TogglePin()`.

In addition to standard GPIO modes (input, output, analog), the pin mode can be configured as EXTI with interrupt or event generation.

When selecting EXTI mode with interrupt generation, the user must call `HAL_GPIO_EXTI_IRQHandler()` from `stm32l4xx_it.c` and implement `HAL_GPIO_EXTI_Callback()`

The table below describes the `GPIO_InitTypeDef` structure field.

Table 13: Description of GPIO_InitTypeDef structure

Structure field	Description
Pin	Specifies the GPIO pins to be configured. Possible values: GPIO_PIN_x or GPIO_PIN_All, where x[0..15]
Mode	Specifies the operating mode for the selected pins: GPIO mode or EXTI mode. Possible values are: <u>GPIO mode</u> - GPIO_MODE_INPUT: Input floating - GPIO_MODE_OUTPUT_PP: Output push-pull - GPIO_MODE_OUTPUT_OD: Output open drain - GPIO_MODE_AF_PP: Alternate Function push-pull - GPIO_MODE_AF_OD: Alternate Function open drain - GPIO_MODE_ANALOG: Analog mode - GPIO_MODE_ANALOG_ADC_CONTROL: ADC analog mode <u>External Interrupt mode</u> - GPIO_MODE_IT_RISING: Rising edge trigger detection - GPIO_MODE_IT_FALLING: Falling edge trigger detection - GPIO_MODE_IT_RISING_FALLING: Rising/Falling edge trigger detection <u>External Event mode</u> - GPIO_MODE_EVT_RISING: Rising edge trigger detection - GPIO_MODE_EVT_FALLING: Falling edge trigger detection - GPIO_MODE_EVT_RISING_FALLING: Rising/Falling edge trigger detection
Pull	Specifies the Pull-up or Pull-down activation for the selected pins. Possible values are: GPIO_NOPULL GPIO_PULLUP GPIO_PULLDOWN
Speed	Specifies the speed for the selected pins Possible values are: GPIO_SPEED_FREQ_LOW GPIO_SPEED_FREQ_MEDIUM GPIO_SPEED_FREQ_HIGH GPIO_SPEED_FREQ_VERY_HIGH

Please find below typical GPIO configuration examples:

- Configuring GPIOs as output push-pull to drive external LEDs:

```
GPIO_InitStruct.Pin = GPIO_PIN_12 | GPIO_PIN_13 | GPIO_PIN_14 | GPIO_PIN_15;
GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
GPIO_InitStruct.Pull = GPIO_PULLUP;
GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_MEDIUM;
HAL_GPIO_Init(GPIOD, &GPIO_InitStruct);
```

- Configuring PA0 as external interrupt with falling edge sensitivity:

```
GPIO_InitStructure.Mode = GPIO_MODE_IT_FALLING;
GPIO_InitStructure.Pull = GPIO_NOPULL;
GPIO_InitStructure.Pin = GPIO_PIN_0;
HAL_GPIO_Init(GPIOA, &GPIO_InitStructure);
```

2.11.3 Cortex NVIC and SysTick timer

The Cortex HAL driver, `stm32l4xx_hal_cortex.c`, provides APIs to handle NVIC and SysTick. The supported APIs include:

- `HAL_NVIC_SetPriority()/ HAL_NVIC_SetPriorityGrouping()`
- `HAL_NVIC_GetPriority() / HAL_NVIC_GetPriorityGrouping()`
- `HAL_NVIC_EnableIRQ()/HAL_NVIC_DisableIRQ()`
- `HAL_NVIC_SystemReset()`
- `HAL_SYSTICK_IRQHandler()`
- `HAL_NVIC_GetPendingIRQ() / HAL_NVIC_SetPendingIRQ () / HAL_NVIC_ClearPendingIRQ()`
- `HAL_NVIC_GetActive(IRQn)`
- `HAL_SYSTICK_Config()`
- `HAL_SYSTICK_CLKSourceConfig()`
- `HAL_SYSTICK_Callback()`

2.11.4 PWR

The PWR HAL driver handles power management. The features shared between all STM32 Series are listed below:

- PVD configuration, enabling/disabling and interrupt handling
 - `HAL_PWR_ConfigPVD()`
 - `HAL_PWR_EnablePVD() / HAL_PWR_DisablePVD()`
 - `HAL_PWR_PVD_IRQHandler()`
 - `HAL_PWR_PVDCallback()`
- Wakeup pin configuration
 - `HAL_PWR_EnableWakeUpPin() / HAL_PWR_DisableWakeUpPin()`
- Low-power mode entry
 - `HAL_PWR_EnterSLEEPMode()`
 - `HAL_PWR_EnterSTOPMode()` (kept for compatibility with other family but identical to `HAL_PWREx_EnterSTOP0Mode()` or `HAL_PWREx_EnterSTOP1Mode()` (see hereafter))
 - `HAL_PWR_EnterSTANDBYMode()`
- STM32L4 series and STM32L4+ series new low-power management features:
 - `HAL_PWREx_EnterSTOP0Mode()`
 - `HAL_PWREx_EnterSTOP1Mode()`
 - `HAL_PWREx_EnterSTOP2Mode()`
 - `HAL_PWREx_EnterSHUTDOWNMode()`

2.11.5 EXTI

The EXTI is not considered as a standalone peripheral but rather as a service used by other peripheral. As a result there are no EXTI APIs but each peripheral HAL driver implements the associated EXTI configuration and EXTI function are implemented as macros in its header file.

The first 16 EXTI lines connected to the GPIOs are managed within the GPIO driver. The `GPIO_InitTypeDef` structure allows configuring an I/O as external interrupt or external event.

The EXTI lines connected internally to the PVD, RTC, USB, and Ethernet are configured within the HAL drivers of these peripheral through the macros given in the table below. The EXTI internal connections depend on the targeted STM32 microcontroller (refer to the product datasheet for more details):

Table 14: Description of EXTI configuration macros

Macros	Description
<code>__HAL_PPP_{SUBBLOCK}_EXTI_ENABLE_IT()</code>	Enables a given EXTI line interrupt Example: <code>__HAL_PWR_PVD_EXTI_ENABLE_IT()</code>
<code>__HAL_PPP_{SUBBLOCK}_EXTI_DISABLE_IT()</code>	Disables a given EXTI line. Example: <code>__HAL_PWR_PVD_EXTI_DISABLE_IT()</code>
<code>__HAL_PPP_{SUBBLOCK}_EXTI_GET_FLAG()</code>	Gets a given EXTI line interrupt flag pending bit status. Example: <code>__HAL_PWR_PVD_EXTI_GET_FLAG()</code>
<code>__HAL_PPP_{SUBBLOCK}_EXTI_CLEAR_FLAG()</code>	Clears a given EXTI line interrupt flag pending bit. Example; <code>__HAL_PWR_PVD_EXTI_CLEAR_FLAG()</code>
<code>__HAL_PPP_{SUBBLOCK}_EXTI_GENERATE_SWIT()</code>	Generates a software interrupt for a given EXTI line. Example: <code>__HAL_PWR_PVD_EXTI_GENERATE_SWIT ()</code>
<code>__HAL_PPP_SUBBLOCK_EXTI_ENABLE_EVENT()</code>	Enable a given EXTI line event Example: <code>__HAL_RTC_WAKEUP_EXTI_ENABLE_EVENT()</code>
<code>__HAL_PPP_SUBBLOCK_EXTI_DISABLE_EVENT()</code>	Disable a given EXTI line event Example: <code>__HAL_RTC_WAKEUP_EXTI_DISABLE_EVENT()</code>
<code>__HAL_PPP_SUBBLOCK_EXTI_ENABLE_RISING_EDGE()</code>	Configure an EXTI Interrupt or Event on rising edge
<code>__HAL_PPP_SUBBLOCK_EXTI_DISABLE_FALLING_EDGE()</code>	Enable an EXTI Interrupt or Event on Falling edge
<code>__HAL_PPP_SUBBLOCK_EXTI_DISABLE_RISING_EDGE()</code>	Disable an EXTI Interrupt or Event on rising edge
<code>__HAL_PPP_SUBBLOCK_EXTI_DISABLE_FALLING_EDGE()</code>	Disable an EXTI Interrupt or Event on Falling edge
<code>__HAL_PPP_SUBBLOCK_EXTI_ENABLE_RISING_FALLING_EDGE()</code>	Enable an EXTI Interrupt or Event on Rising/Falling edge
<code>__HAL_PPP_SUBBLOCK_EXTI_DISABLE_RISING_FALLING_EDGE()</code>	Disable an EXTI Interrupt or Event on Rising/Falling edge

If the EXTI interrupt mode is selected, the user application must call `HAL_PPP_FUNCTION_IRQHandler()` (for example `HAL_PWR_PVD_IRQHandler()`), from `stm32l4xx_it.c` file, and implement `HAL_PPP_FUNCTIONCallback()` callback function (for example `HAL_PWR_PVDCallback()`).

2.11.6 DMA

The DMA HAL driver allows enabling and configuring the peripheral to be connected to the DMA Channels (except for internal SRAM/FLASH memory which do not require any initialization). Refer to the product reference manual for details on the DMA request corresponding to each peripheral.

For a given channel, HAL_DMA_Init() API allows programming the required configuration through the following parameters:

- Transfer Direction
- Source and Destination data formats
- Normal or Circular mode
- Channel Priority level
- Source and Destination Increment mode
- Hardware request connected to the peripheral

Two operating modes are available:

- Polling mode I/O operation
 - a. Use HAL_DMA_Start() to start DMA transfer when the source and destination addresses and the Length of data to be transferred have been configured.
 - b. Use HAL_DMA_PollForTransfer() to poll for the end of current transfer. In this case a fixed timeout can be configured depending on the user application.
- Interrupt mode I/O operation
 - a. Configure the DMA interrupt priority using HAL_NVIC_SetPriority()
 - b. Enable the DMA IRQ handler using HAL_NVIC_EnableIRQ()
 - c. Use HAL_DMA_Start_IT() to start DMA transfer when the source and destination addresses and the length of data to be transferred have been configured. In this case the DMA interrupt is configured.
 - d. Use HAL_DMA_IRQHandler() called under DMA_IRQHandler() Interrupt subroutine
 - e. When data transfer is complete, HAL_DMA_IRQHandler() function is executed and a user function can be called by customizing XferCpltCallback and XferErrorCallback function pointer (i.e. a member of DMA handle structure).

Additional functions and macros are available to ensure efficient DMA management:

- Use HAL_DMA_GetState() function to return the DMA state and HAL_DMA_GetError() in case of error detection.
- Use HAL_DMA_Abort() function to abort the current transfer

The most used DMA HAL driver macros are the following:

- __HAL_DMA_ENABLE: enables the specified DMA channel.
- __HAL_DMA_DISABLE: disables the specified DMA channel.
- __HAL_DMA_GET_FLAG: gets the DMA channel pending flags.
- __HAL_DMA_CLEAR_FLAG: clears the DMA channel pending flags.
- __HAL_DMA_ENABLE_IT: enables the specified DMA channel interrupts.
- __HAL_DMA_DISABLE_IT: disables the specified DMA channel interrupts.
- __HAL_DMA_GET_IT_SOURCE: checks whether the specified DMA channel interrupt has been enabled or not.



When a peripheral is used in DMA mode, the DMA initialization should be done in the HAL_PPP_MspInit() callback. In addition, the user application should associate the DMA handle to the PPP handle (refer to section “HAL IO operation

functions”).



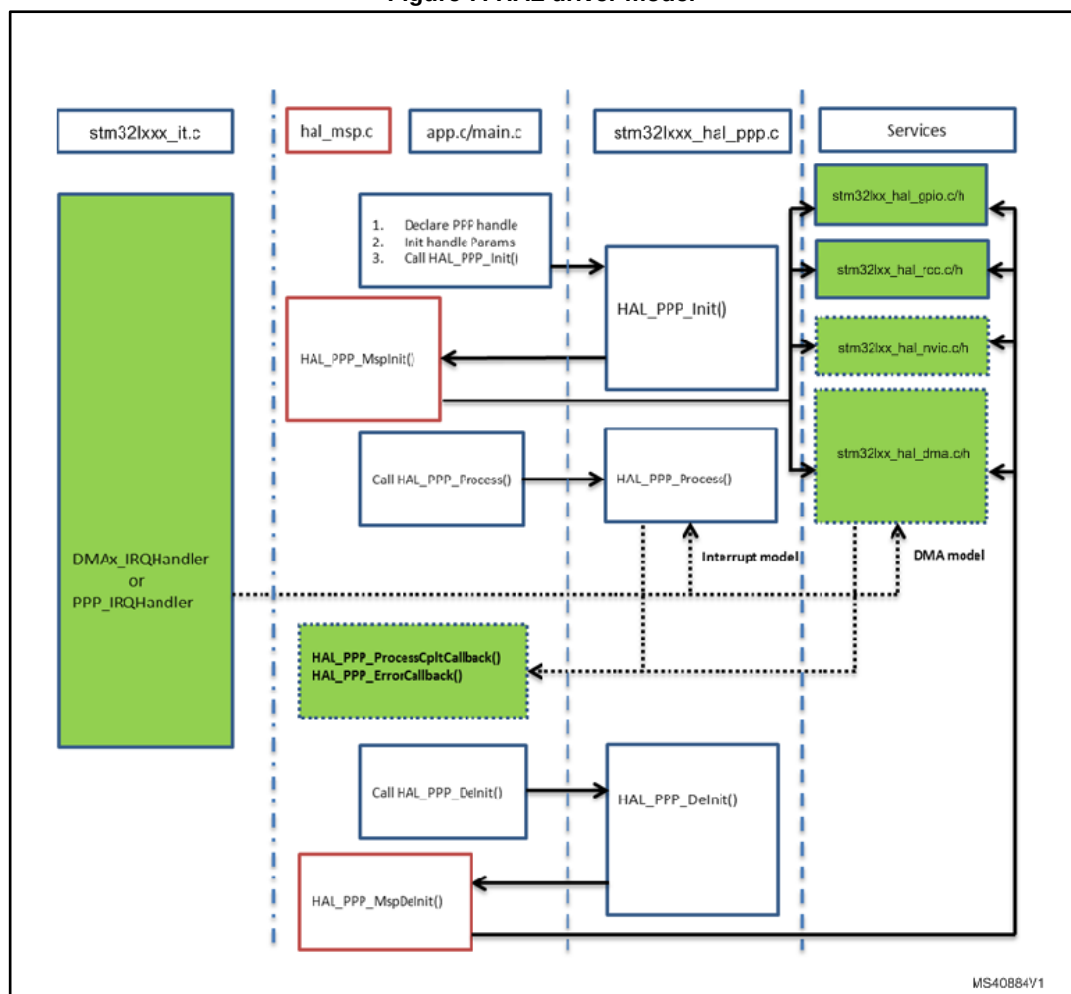
DMA channel callbacks need to be initialized by the user application only in case of memory-to-memory transfer. However when peripheral-to-memory transfers are used, these callbacks are automatically initialized by calling a process API function that uses the DMA.

2.12 How to use HAL drivers

2.12.1 HAL usage models

The following figure shows the typical use of the HAL driver and the interaction between the application user, the HAL driver and the interrupts.

Figure 7: HAL driver model



The functions implemented in the HAL driver are shown in green, the functions called from interrupt handlers in dotted lines, and the msp functions implemented in the user application in red. Non-dotted lines represent the interactions between the user application functions.

Basically, the HAL driver APIs are called from user files and optionally from interrupt handlers file when the APIs based on the DMA or the PPP peripheral dedicated interrupts are used.

When DMA or PPP peripheral interrupts are used, the PPP process complete callbacks are called to inform the user about the process completion in real-time event mode (interrupts). Note that the same process completion callbacks are used for DMA in interrupt mode.

2.12.2 HAL initialization

2.12.2.1 HAL global initialization

In addition to the peripheral initialization and de-initialization functions, a set of APIs are provided to initialize the HAL core implemented in file `stm32l4xx_hal.c`.

- `HAL_Init()`: this function must be called at application startup to
 - initialize data/instruction cache and pre-fetch queue
 - set SysTick timer to generate an interrupt each 1ms (based on HSI clock) with the lowest priority
 - call `HAL_MspInit()` user callback function to perform system level initializations (Clock, GPIOs, DMA, interrupts). `HAL_MspInit()` is defined as “weak” empty function in the HAL drivers.
- `HAL_DeInit()`
 - resets all peripherals
 - calls function `HAL_MspDeInit()` which is a user callback function to do system level De-Initializations.
- `HAL_GetTick()`: this function gets current SysTick counter value (incremented in SysTick interrupt) used by peripherals drivers to handle timeouts.
- `HAL_Delay()`: this function implements a delay (expressed in milliseconds) using the SysTick timer.

Care must be taken when using `HAL_Delay()` since this function provides an accurate delay (expressed in milliseconds) based on a variable incremented in SysTick ISR. This means that if `HAL_Delay()` is called from a peripheral ISR, then the SysTick interrupt must have highest priority (numerically lower) than the peripheral interrupt, otherwise the caller ISR will be blocked.

2.12.2.2 System clock initialization

The clock configuration is done at the beginning of the user code. However the user can change the configuration of the clock in his own code. Please find below the typical Clock configuration sequence to reach the maximum 80 MHz clock frequency based on the HSE clock:

```
void SystemClock_Config(void)
{
    RCC_ClkInitTypeDef clkinitstruct = {0};
    RCC_OscInitTypeDef oscinitstruct = {0};

    /* Configure PLLs-----*/
    /* PLL configuration: PLLCLK = (HSE/PLLM * PLLN) / PLLR = (16/1 * 20) / 2 = 80
    MHz*/
    /* Enable HSE Oscillator and activate PLL with HSE as source */
    oscinitstruct.OscillatorType = RCC_OSCILLATORTYPE_HSE;
    oscinitstruct.HSEState = RCC_HSE_ON;
    oscinitstruct.PLL.PLLState = RCC_PLL_ON;
    oscinitstruct.PLL.PLLSource = RCC_PLLSOURCE_HSE;
    oscinitstruct.PLL.PLLM = 1;
    oscinitstruct.PLL.PLLN = 20;
    oscinitstruct.PLL.PLLR = 2;
    oscinitstruct.PLL.PLLQ = 7;
    oscinitstruct.PLL.PLLQ = 4;
    if (HAL_RCC_OscConfig(&oscinitstruct) != HAL_OK)
```

```

{
    /* Initialization Error */
    while(1);
}
/* Select PLL as system clock source and configure the HCLK, PCLK1 and PCLK2
clocks dividers */
clkinitstruct.ClockType = (RCC_CLOCKTYPE_SYSCLK I   RCC_CLOCKTYPE_HCLK I
RCC_CLOCKTYPE_PCLK1 I RCC_CLOCKTYPE_PCLK2);
clkinitstruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
clkinitstruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
clkinitstruct.APB2CLKDivider = RCC_HCLK_DIV1;
clkinitstruct.APB1CLKDivider = RCC_HCLK_DIV1;
if
    (HAL_RCC_ClockConfig(&clkinitstruct,FLASH_LATENCY_4) != HAL_OK)
{
    /* Initialization Error */
    while(1);
}
}

```

2.12.2.3 HAL MSP initialization process

The peripheral initialization is done through *HAL_PPP_Init()* while the hardware resources initialization used by a peripheral (PPP) is performed during this initialization by calling MSP callback function *HAL_PPP_MspInit()*.

The MspInit callback performs the low level initialization related to the different additional hardware resources: RCC, GPIO, NVIC and DMA.

All the HAL drivers with handles include two MSP callbacks for initialization and de-initialization:

```

/**
 * @brief Initializes the PPP MSP.
 * @param hppp: PPP handle
 * @retval None */
void __weak HAL_PPP_MspInit(PPP_HandleTypeDef *hppp) {
    /* NOTE: This function Should not be modified, when the callback is needed,
    the HAL_PPP_MspInit could be implemented in the user file */
}
/**
 * @brief DeInitializes PPP MSP.
 * @param hppp: PPP handle
 * @retval None */
void __weak HAL_PPP_MspDeInit(PPP_HandleTypeDef *hppp) {
    /* NOTE: This function Should not be modified, when the callback is needed,
    the HAL_PPP_MspDeInit could be implemented in the user file */
}

```

The MSP callbacks are declared empty as weak functions in each peripheral driver. The user can use them to set the low level initialization code or omit them and use his own initialization routine.

The HAL MSP callback is implemented inside the *stm32l4xx_hal_msp.c* file in the user folders. An *stm32l4xx_hal_msp.c* file template is located in the HAL folder and should be copied to the user folder. It can be generated automatically by STM32CubeMX tool and further modified. Note that all the routines are declared as weak functions and could be overwritten or removed to use user low level initialization code.

stm32l4xx_hal_msp.c file contains the following functions:

Table 15: MSP functions

Routine	Description
void HAL_MspInit()	Global MSP initialization routine
void HAL_MspDeInit()	Global MSP de-initialization routine

Routine	Description
void HAL_PPP_MspInit()	PPP MSP initialization routine
void HAL_PPP_MspDeInit()	PPP MSP de-initialization routine

By default, if no peripheral needs to be de-initialized during the program execution, the whole MSP initialization is done in *Hal_MspInit()* and MSP De-Initialization in the *Hal_MspDeInit()*. In this case the *HAL_PPP_MspInit()* and *HAL_PPP_MspDeInit()* are not implemented.

When one or more peripherals needs to be de-initialized in run time and the low level resources of a given peripheral need to be released and used by another peripheral, *HAL_PPP_MspDeInit()* and *HAL_PPP_MspInit()* are implemented for the concerned peripheral and other peripherals initialization and de-Initialization are kept in the global *HAL_MspInit()* and the *HAL_MspDeInit()*.

If there is nothing to be initialized by the global *HAL_MspInit()* and *HAL_MspDeInit()*, the two routines can simply be omitted.

2.12.3 HAL IO operation process

The HAL functions with internal data processing like transmit, receive, write and read are generally provided with three data processing modes as follows:

- Polling mode
- Interrupt mode
- DMA mode

2.12.3.1 Polling mode

In Polling mode, the HAL functions return the process status when the data processing in blocking mode is complete. The operation is considered complete when the function returns the HAL_OK status, otherwise an error status is returned. The user can get more information through the *HAL_PPP_GetState()* function. The data processing is handled internally in a loop. A timeout (expressed in ms) is used to prevent process hanging.

The example below shows the typical Polling mode processing sequence:

```
HAL_StatusTypeDef HAL_PPP_Transmit ( PPP_HandleTypeDef * phandle, uint8_t pData,
int16_t Size, uint32_t Timeout)
{
    if((pData == NULL) || (Size == 0))
    {
        return HAL_ERROR;
    }
    (...) while (data processing is running)
    {
        if( timeout reached )
        {
            return HAL_TIMEOUT;
        }
    }
    (...)
    return HAL_OK; }

```

2.12.3.2 Interrupt mode

In interrupt mode, the HAL function returns the process status after starting the data processing and enabling the appropriate interruption. The end of the operation is indicated by a callback declared as a weak function. It can be customized by the user to be informed in real-time about the process completion. The user can also get the process status through the *HAL_PPP_GetState()* function.

In interrupt mode, four functions are declared in the driver:

- *HAL_PPP_Process_IT()*: launch the process
- *HAL_PPP_IRQHandler()*: the global PPP peripheral interruption
- *__weak HAL_PPP_ProcessCpltCallback ()*: the callback relative to the process completion.
- *__weak HAL_PPP_ProcessErrorCallback()*: the callback relative to the process Error.

To use a process in interrupt mode, *HAL_PPP_Process_IT()* is called in the user file and *HAL_PPP_IRQHandler* in *stm32l4xx_it.c*.

The *HAL_PPP_ProcessCpltCallback()* function is declared as weak function in the driver. This means that the user can declare it again in the application. The function in the driver is not modified.

An example of use is illustrated below:

main.c file:

```
UART_HandleTypeDef UartHandle;
int main(void)
{
    /* Set User Parameters */
    UartHandle.Init.BaudRate = 9600;
    UartHandle.Init.WordLength = UART_DATABITS_8;
    UartHandle.Init.StopBits = UART_STOPBITS_1;
    UartHandle.Init.Parity = UART_PARITY_NONE;
    UartHandle.Init.HwFlowCtl = UART_HWCONTROL_NONE;
    UartHandle.Init.Mode = UART_MODE_TX_RX;
    UartHandle.Init.Instance = USART1;
    HAL_UART_Init(&UartHandle);
    HAL_UART_SendIT(&UartHandle, TxBuffer, sizeof(TxBuffer));
    while (1);
}
void HAL_UART_TxCpltCallback(UART_HandleTypeDef *huart)
{
}
void HAL_UART_ErrorCallback(UART_HandleTypeDef *huart)
{
}
```

stm32l4xx_it.c file:

```
extern UART_HandleTypeDef UartHandle;
void USART1_IRQHandler(void)
{
    HAL_UART_IRQHandler(&UartHandle);
}
```

2.12.3.3 DMA mode

In DMA mode, the HAL function returns the process status after starting the data processing through the DMA and after enabling the appropriate DMA interruption. The end of the operation is indicated by a callback declared as a weak function and can be customized by the user to be informed in real-time about the process completion. The user can also get the process status through the `HAL_PPP_GetState()` function. For the DMA mode, three functions are declared in the driver:

- `HAL_PPP_Process_DMA()`: launch the process
- `HAL_PPP_DMA_IRQHandler()`: the DMA interruption used by the PPP peripheral
- `__weak HAL_PPP_ProcessCpltCallback()`: the callback relative to the process completion.
- `__weak HAL_PPP_ErrorCpltCallback()`: the callback relative to the process Error.

To use a process in DMA mode, `HAL_PPP_Process_DMA()` is called in the user file and the `HAL_PPP_DMA_IRQHandler()` is placed in the `stm32l4xx_it.c`. When DMA mode is used, the DMA initialization is done in the `HAL_PPP_MspInit()` callback. The user should also associate the DMA handle to the PPP handle. For this purpose, the handles of all the peripheral drivers that use the DMA must be declared as follows:

```
typedef struct
{
  PPP_TypeDef *Instance; /* Register base address */
  PPP_InitTypeDef Init; /* PPP communication parameters */
  HAL_StateTypeDef State; /* PPP communication state */
  (...)
  DMA_HandleTypeDef *hdma; /* associated DMA handle */
} PPP_HandleTypeDef;
```

The initialization is done as follows (UART example):

```
int main(void)
{
  /* Set User Parameters */
  UartHandle.Init.BaudRate = 9600;
  UartHandle.Init.WordLength = UART_DATABITS 8;
  UartHandle.Init.StopBits = UART_STOPBITS 1;
  UartHandle.Init.Parity = UART_PARITY NONE;
  UartHandle.Init.HwFlowCtl = UART_HWCONTROL_NONE;
  UartHandle.Init.Mode = UART_MODE_TX_RX;
  UartHandle.Init.Instance = UART1;
  HAL_UART_Init(&UartHandle);
  (...)
}

void HAL_USART_MspInit (UART_HandleTypeDef * huart)
{
  static DMA_HandleTypeDef hdma_tx;
  static DMA_HandleTypeDef hdma_rx;
  (...)
  HAL_LINKDMA(UartHandle, DMA_Handle_tx, hdma_tx);
  HAL_LINKDMA(UartHandle, DMA_Handle_rx, hdma_rx);
  (...)
}
```

The `HAL_PPP_ProcessCpltCallback()` function is declared as weak function in the driver that means, the user can declare it again in the application code. The function in the driver should not be modified.

An example of use is illustrated below:

main.c file:

```
UART_HandleTypeDef UartHandle;
int main(void)
{
    /* Set User Paramaters */
    UartHandle.Init.BaudRate = 9600;
    UartHandle.Init.WordLength = UART_DATABITS_8;
    UartHandle.Init.StopBits = UART_STOPBITS_1;
    UartHandle.Init.Parity = UART_PARITY_NONE;
    UartHandle.Init.HwFlowCtl = UART_HWCONTROL_NONE;
    UartHandle.Init.Mode = UART_MODE_TX_RX; UartHandle.Init.Instance = USART1;
    HAL_UART_Init(&UartHandle);
    HAL_UART_Send_DMA(&UartHandle, TxBuffer, sizeof(TxBuffer));
    while (1);
}

void HAL_UART_TxCpltCallback(UART_HandleTypeDef *phuart)
{
}

void HAL_UART_TxErrorCallback(UART_HandleTypeDef *phuart)
{
}
```

stm32l4xx_it.c file:

```
extern UART_HandleTypeDef UartHandle;
void DMAx_IRQHandler(void)
{
    HAL_DMA_IRQHandler(&UartHandle.DMA_Handle_tx);
}
```

HAL_USART_TxCpltCallback() and *HAL_USART_ErrorCallback()* should be linked in the *HAL_PPP_Process_DMA()* function to the DMA transfer complete callback and the DMA transfer Error callback by using the following statement:

```
HAL_PPP_Process_DMA (PPP_HandleTypeDef *hppp, Params...)
{
    (...)
    hppp->DMA_Handle->XferCpltCallback = HAL_UART_TxCpltCallback ;
    hppp->DMA_Handle->XferErrorCallback = HAL_UART_ErrorCallback ;
    (...)
}
```

2.12.4 Timeout and error management

2.12.4.1 Timeout management

The timeout is often used for the APIs that operate in polling mode. It defines the delay during which a blocking process should wait till an error is returned. An example is provided below:

```
HAL_StatusTypeDef HAL_DMA_PollForTransfer(DMA_HandleTypeDef *hdma, uint32_t CompleteLevel, uint32_t Timeout)
```

The timeout possible value are the following:

Table 16: Timeout values

Timeout value	Description
0	No poll: Immediate process check and exit
1 ... (HAL_MAX_DELAY -1) ⁽¹⁾	Timeout in ms
HAL_MAX_DELAY	Infinite poll till process is successful

Notes:

⁽¹⁾HAL_MAX_DELAY is defined in the stm32l4xx_hal_def.h as 0xFFFFFFFF

However, in some cases, a fixed timeout is used for system peripherals or internal HAL driver processes. In these cases, the timeout has the same meaning and is used in the same way, except when it is defined locally in the drivers and cannot be modified or introduced as an argument in the user application.

Example of fixed timeout:

```
#define LOCAL_PROCESS_TIMEOUT 100
HAL_StatusTypeDef HAL_PPP_Process(PPP_HandleTypeDef)
{
    (...)
    timeout = HAL_GetTick() + LOCAL_PROCESS_TIMEOUT;
    (...)
    while(ProcessOngoing)
    {
        (...)
        if(HAL_GetTick() >= timeout)
        {
            /* Process unlocked */
            __HAL_UNLOCK(hppp);
            hppp->State= HAL_PPP_STATE_TIMEOUT;
            return HAL_PPP_STATE_TIMEOUT;
        }
    }
    (...)
}
```

The following example shows how to use the timeout inside the polling functions:

```
HAL_PPP_StateTypeDef HAL_PPP_Poll (PPP_HandleTypeDef *hppp, uint32_t Timeout)
{
    (...)
    timeout = HAL_GetTick() + Timeout;
    (...)
    while(ProcessOngoing)
    {
        (...)
        if(Timeout != HAL_MAX_DELAY)
        {
            if(HAL_GetTick() >= timeout)
```

```

{
  /* Process unlocked */
  HAL_UNLOCK(hppp);
  hppp->State= HAL_PPP_STATE_TIMEOUT;
  return hppp->State;
}
(...)
}

```

2.12.4.2 Error management

The HAL drivers implement a check on the following items:

- Valid parameters: for some process the used parameters should be valid and already defined, otherwise the system may crash or go into an undefined state. These critical parameters are checked before being used (see example below).

```

HAL StatusTypeDef HAL_PPP_Process(PPP_HandleTypeDef* hppp, uint32_t *pdata, uint32_t
Size)
{ if ((pdata == NULL) || (Size == 0))
{ return HAL_ERROR;
}
}

```

- Valid handle: the PPP peripheral handle is the most important argument since it keeps the PPP driver vital parameters. It is always checked in the beginning of the *HAL_PPP_Init()* function.

```

HAL StatusTypeDef HAL_PPP_Init(PPP_HandleTypeDef* hppp)
{ if (hppp == NULL) //the handle should be already allocated
{ return HAL_ERROR;
}
}

```

- Timeout error: the following statement is used when a timeout error occurs:

```

while (Process ongoing)
{ timeout = HAL_GetTick() + Timeout;
while (data processing is running)
{ if(timeout)
{ return HAL_TIMEOUT;
}
}
}

```

When an error occurs during a peripheral process, *HAL_PPP_Process()* returns with a *HAL_ERROR* status. The HAL PPP driver implements the *HAL_PPP_GetError()* to allow retrieving the origin of the error.

```

HAL_PPP_ErrorTypeDef HAL_PPP_GetError (PPP_HandleTypeDef *hppp);

```

In all peripheral handles, a *HAL_PPP_ErrorTypeDef* is defined and used to store the last error code.

```

typedef struct
{
  PPP_TypeDef * Instance; /* PPP registers base address */
  PPP_InitTypeDef Init; /* PPP initialization parameters */
  HAL_LockTypeDef Lock; /* PPP locking object */
  __IO HAL_PPP_StateTypeDef State; /* PPP state */
  __IO HAL_PPP_ErrorTypeDef ErrorCode; /* PPP Error code */
  (...)
  /* PPP specific parameters */
}
PPP_HandleTypeDef;

```

The error state and the peripheral global state are always updated before returning an error:

```
PPP->State = HAL_PPP_READY; /* Set the peripheral ready */
PP->ErrorCode = HAL_ERRORCODE; /* Set the error code */
HAL_UNLOCK(PPP); /* Unlock the PPP resources */
return HAL_ERROR; /*return with HAL error */
```

HAL_PPP_GetError () must be used in interrupt mode in the error callback:

```
void HAL_PPP_ProcessCpltCallback(PPP_HandleTypeDef *hspi)
{
  ErrorCode = HAL_PPP_GetError (hspi); /* retrieve error code */
}
```

2.12.4.3 Run-time checking

The HAL implements run-time failure detection by checking the input values of all HAL driver functions. The run-time checking is achieved by using an *assert_param* macro. This macro is used in all the HAL driver functions which have an input parameter. It allows verifying that the input value lies within the parameter allowed values.

To enable the run-time checking, use the *assert_param* macro, and leave the define **USE_FULL_ASSERT** uncommented in *stm32l4xx_hal_conf.h* file.

```
void HAL_UART_Init(UART_HandleTypeDef *huart)
{
  (...) /* Check the parameters */
  assert_param(IS_UART_INSTANCE(huart->Instance));
  assert_param(IS_UART_BAUDRATE(huart->Init.BaudRate));
  assert_param(IS_UART_WORD_LENGTH(huart->Init.WordLength));
  assert_param(IS_UART_STOPBITS(huart->Init.StopBits));
  assert_param(IS_UART_PARITY(huart->Init.Parity));
  assert_param(IS_UART_MODE(huart->Init.Mode));
  assert_param(IS_UART_HARDWARE_FLOW_CONTROL(huart->Init.HwFlowCtl));
  (...)

  /** @defgroup UART_Word_Length *
  @{
  */
  #define UART_WORDLENGTH_8B ((uint32_t)0x00000000)
  #define UART_WORDLENGTH_9B ((uint32_t)USART_CR1_M)
  #define IS_UART_WORD_LENGTH(LENGTH) (((LENGTH) == UART_WORDLENGTH_8B) || \
    \ ((LENGTH) == UART_WORDLENGTH_9B))
```

If the expression passed to the *assert_param* macro is false, the *assert_failed* function is called and returns the name of the source file and the source line number of the call that failed. If the expression is true, no value is returned.

The *assert_param* macro is implemented in *stm32l4xx_hal_conf.h*:

```
/* Exported macro -----*/
#ifdef USE_FULL_ASSERT
/**
 * @brief The assert_param macro is used for function's parameters check.
 * @param expr: If expr is false, it calls assert failed function
 * which reports the name of the source file and the source
 * line number of the call that failed.
 * If expr is true, it returns no value.
 * @retval None */
#define assert_param(expr) ((expr)?(void)0:assert_failed((uint8_t *) FILE, \
  _LINE_))
/* Exported functions -----*/
void assert_failed(uint8_t * file, uint32_t line);
#else
#define assert_param(expr) ((void)0)
#endif /* USE_FULL_ASSERT */
```

The `assert_failed` function is implemented in the `main.c` file or in any other user C file:

```
#ifndef USE_FULL_ASSERT /**
 * @brief Reports the name of the source file and the source line number
 * where the assert param error has occurred.
 * @param file: pointer to the source file name
 * @param line: assert_param error line source number
 * @retval None */
void assert_failed(uint8_t* file, uint32_t line)
{
    /* User can add his own implementation to report the file name and line number,
    ex: printf("Wrong parameters value: file %s on line %d\r\n", file, line) */
    /* Infinite loop */
    while (1)
    {
    }
}
```



Because of the overhead run-time checking introduces, it is recommended to use it during application code development and debugging, and to remove it from the final application to improve code size and speed.

3 Overview of low-layer drivers

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

The LL drivers feature:

- A set of functions to initialize peripheral main features according to the parameters specified in data structures
- A set of functions used to fill initialization data structures with the reset values of each field
- Functions to perform peripheral de-initialization (peripheral registers restored to their default values)
- A set of inline functions for direct and atomic register access
- Full independence from HAL since LL drivers can be used either in standalone mode (without HAL drivers) or in mixed mode (with HAL drivers)
- Full coverage of the supported peripheral features.

The low-layer drivers provide hardware services based on the available features of the STM32 peripherals. These services reflect exactly the hardware capabilities and provide one-shot operations that must be called following the programming model described in the microcontroller line reference manual. As a result, the LL services do not implement any processing and do not require any additional memory resources to save their states, counter or data pointers: all the operations are performed by changing the associated peripheral registers content.

3.1 Low-layer files

The low-layer drivers are built around header/C files (one per each supported peripheral) plus five header files for some System and Cortex related features.

Table 17: LL driver files

File	Description
<i>stm32l4xx_ll_bus.h</i>	This is the h-source file for core bus control and peripheral clock activation and deactivation <i>Example: LL_AHB2_GRP1_EnableClock</i>
<i>stm32l4xx_ll_ppp.h/c</i>	stm32l4xx_ll_ppp.c provides peripheral initialization functions such as LL_PPP_Init(), LL_PPP_StructInit(), LL_PPP_DeInit(). All the other APIs are defined within stm32l4xx_ll_ppp.h file. The low-layer PPP driver is a standalone module. To use it, the application must include it in the xx_ll_ppp.h file.
<i>stm32l4xx_ll_cortex.h</i>	Cortex-M related register operation APIs including the SysTick, Low power (LL_SYSTICK_xxxxx, LL_LPM_xxxxx "Low Power Mode" ...)
<i>stm32l4xx_ll_utils.h/c</i>	This file covers the generic APIs: • Read of device unique ID and electronic signature • Timebase and delay management • System clock configuration.
<i>stm32l4xx_ll_system.h</i>	System related operations (LL_SYSCFG_xxx, LL_DBGMCU_xxx, LL_FLASH_xxx and LL_VREFBUF_xxx)

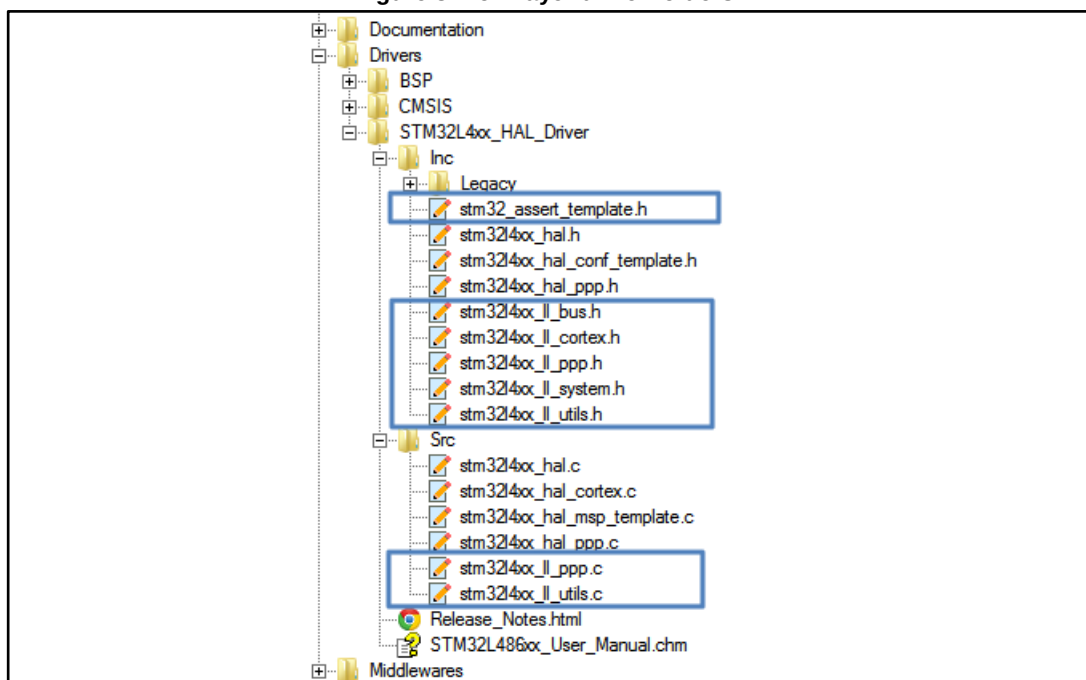
File	Description
stm32_assert_template.h	Template file allowing to define the assert_param macro, that is used when run-time checking is enabled. This file is required only when the LL drivers are used in standalone mode (without calling the HAL APIs). It should be copied to the application folder and renamed to stm32_assert.h.



There is no configuration file for the LL drivers.

The low-layer files are located in the same HAL driver folder.

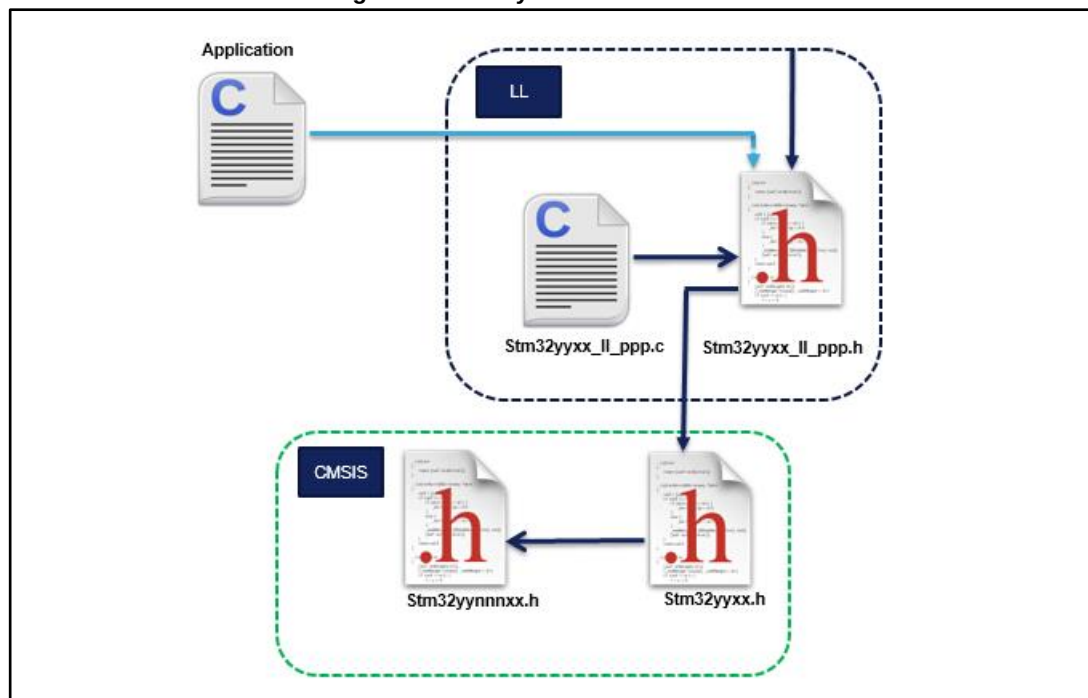
Figure 8: Low-layer driver folders



In general, low-layer drivers include only the STM32 CMSIS device file.

```
#include "stm32yyxx.h"
```

Figure 9: Low-layer driver CMSIS files



Application files have to include only the used low-layer driver header files.

3.2 Overview of low-layer APIs and naming rules

3.2.1 Peripheral initialization functions

The LL drivers offer three set of initialization functions. They are defined in `stm32l4xx_ll_ppp.c` file:

- Functions to initialize peripheral main features according to the parameters specified in data structures
- A set of functions used to fill initialization data structures with the reset values of each field
- Function for peripheral de-initialization (peripheral registers restored to their default values)

The definition of these LL initialization functions and associated resources (structure, literals and prototypes) is conditioned by a compilation switch: `USE_FULL_LL_DRIVER`. To use these functions, this switch must be added in the toolchain compiler preprocessor or to any generic header file which is processed before the LL drivers.

The below table shows the list of the common functions provided for all the supported peripherals:

Table 18: Common peripheral initialization functions

Functions	Return Type	Parameters	Description
LL_PPP_Init	<i>ErrorStatus</i>	• <i>PPP_TypeDef* PPPx</i> • <i>LL_PPP_InitTypeDef* PPP_InitStruct</i>	Initializes the peripheral main features according to the parameters specified in PPP_InitStruct. Example: LL_USART_Init(USART_TypeDef *USARTx, LL_USART_InitTypeDef *USART_InitStruct)
LL_PPP_StructInit	<i>void</i>	• <i>LL_PPP_InitTypeDef* PPP_InitStruct</i>	Fills each PPP_InitStruct member with its default value. Example. LL_USART_StructInit(LL_USART_InitTypeDef *USART_InitStruct)
LL_PPP_DeInit	<i>ErrorStatus</i>	• <i>PPP_TypeDef* PPPx</i>	De-initializes the peripheral registers, that is restore them to their default reset values. Example. LL_USART_DeInit(USART_TypeDef *USARTx)

Additional functions are available for some peripherals (refer to [Table 19: "Optional peripheral initialization functions"](#)).

Table 19: Optional peripheral initialization functions

Functions	Return Type	Parameters	Examples
LL_PPP{ _CATEGORY }_Init	Error Status	• <i>PPP_TypeDef* PPPx</i> • <i>LL_PPP{ _CATEGORY }_InitTypeDef* PPP{ _CATEGORY }_InitStruct</i>	Initializes peripheral features according to the parameters specified in PPP_InitStruct. Example: <pre>LL_ADC_INJ_Init(ADC_TypeDef *ADCx, LL_ADC_INJ_InitTypeDef *ADC_INJ_InitStruct)</pre> <pre>LL_RTC_TIME_Init(RTC_TypeDef *RTCx, uint32_t RTC_Format, LL_RTC_TimeTypeDef *RTC_TimeStruct)</pre> <pre>LL_RTC_DATE_Init(RTC_TypeDef *RTCx, uint32_t RTC_Format, LL_RTC_DateTypeDef *RTC_DateStruct)</pre> <pre>LL_TIM_IC_Init(TIM_TypeDef* TIMx, uint32_t Channel, LL_TIM_IC_InitTypeDef* TIM_IC_InitStruct)</pre> <pre>LL_TIM_ENCODER_Init(TIM_TypeDef* TIMx, LL_TIM_ENCODER_InitTypeDef* TIM_EncoderInitStruct)</pre>
LL_PPP{ _CATEGORY }_StructInit	void	<i>LL_PPP{ _CATEGORY }_InitTypeDef* PPP{ _CATEGORY }_InitStruct</i>	Fills each <i>PPP{ _CATEGORY }_InitStruct</i> member with its default value. Example: <pre>LL_ADC_INJ_StructInit(LL_ADC_INJ_InitTypeDef *ADC_INJ_InitStruct)</pre>
LL_PPP_CommonInit	Error Status	• <i>PPP_TypeDef* PPPx</i> • <i>LL_PPP_CommonInitTypeDef* PPP_CommonInitStruct</i>	Initializes the common features shared between different instances of the same peripheral. Example: <pre>LL_ADC_CommonInit(ADC_Common_TypeDef *ADCxy_COMMON, LL_ADC_CommonInitTypeDef *ADC_CommonInitStruct)</pre>

Functions	Return Type	Parameters	Examples
LL_PPP_CommonStructInit	void	LL_PPP_CommonInitTypeDef* PPP_CommonInitStruct	Fills each PPP_CommonInitStruct member with its default value Example: LL_ADC_CommonStructInit(LL_ADC_CommonInitTypeDef *ADC_CommonInitStruct)
LL_PPP_ClockInit	ErrorStatus	- PPP_TypeDef* PPPx - LL_PPP_ClockInitTypeDef* PPP_ClockInitStruct	Initializes the peripheral clock configuration in synchronous mode. Example: LL_USART_ClockInit(USART_TypeDef *USARTx, LL_USART_ClockInitTypeDef *USART_ClockInitStruct)
LL_PPP_ClockStructInit	void	LL_PPP_ClockInitTypeDef* PPP_ClockInitStruct	Fills each PPP_ClockInitStruct member with its default value Example: LL_USART_ClockStructInit(LL_USART_ClockInitTypeDef *USART_ClockInitStruct)

3.2.1.1 Run-time checking

Like HAL drivers, LL initialization functions implement run-time failure detection by checking the input values of all LL driver functions. For more details please refer to [Section 2.12.4.3: "Run-time checking"](#).

When using the LL drivers in standalone mode (without calling HAL functions), the following actions are required to use run-time checking:

1. Copy stm32_assert_template.h to the application folder and rename it to stm32_assert.h. This file defines the assert_param macro which is used when run-time checking is enabled.
2. Include stm32_assert.h file within the application main header file.
3. Add the USE_FULL_ASSERT compilation switch in the toolchain compiler preprocessor or in any generic header file which is processed before the stm32_assert.h driver.



Run-time checking is not available for LL inline functions.

3.2.2 Peripheral register-level configuration functions

On top of the peripheral initialization functions, the LL drivers offer a set of inline functions for direct atomic register access. Their format is as follows:

```
__STATIC_INLINE return_type LL_PPP_Function (PPPx_TypeDef *PPPx, args)
```

The “Function” naming is defined depending to the action category:

- **Specific Interrupt, DMA request and status flags management:**
Set/Get/Clear/Enable/Disable flags on interrupt and status registers

Table 20: Specific Interrupt, DMA request and status flags management

Name	Examples
<code>LL_PPP_{_CATEGORY}_ActionItem_BITNAME</code> <code>LL_PPP{_CATEGORY}_IsItem_BITNAME_Action</code>	• <code>LL_RCC_IsActiveFlag_LSIRDY</code> • <code>LL_RCC_IsActiveFlag_FWRST()</code> • <code>LL_ADC_ClearFlag_EOC(ADC1)</code> • <code>LL_DMA_ClearFlag_TCx(DMA_TypeDef* DMAx)</code>

Table 21: Available function formats

Item	Action	Format
Flag	Get	<code>LL_PPP_IsActiveFlag_BITNAME</code>
	Clear	<code>LL_PPP_ClearFlag_BITNAME</code>
Interrupts	Enable	<code>LL_PPP_EnableIT_BITNAME</code>
	Disable	<code>LL_PPP_DisableIT_BITNAME</code>
	Get	<code>LL_PPP_IsEnabledIT_BITNAME</code>
DMA	Enable	<code>LL_PPP_EnableDMAReq_BITNAME</code>
	Disable	<code>LL_PPP_DisableDMAReq_BITNAME</code>
	Get	<code>LL_PPP_IsEnabledDMAReq_BITNAME</code>



BITNAME refers to the peripheral register bit name as described in the product line reference manual.

- **Peripheral clock activation/deactivation management:** Enable/Disable/Reset a peripheral clock

Table 22: Peripheral clock activation/deactivation management

Name	Examples
<code>LL_BUS_GRPx_ActionClock{Mode}</code>	• <code>LL_AHB2_GRP1_EnableClock</code> (<code>LL_AHB2_GRP1_PERIPH_GPIOA LL_AHB2_GRP1_PERIPH_GPIOB</code>) • <code>by LL_APB1_GRP1_EnableClockSleep</code> (<code>LL_APB1_GRP1_PERIPH_DAC1</code>)



'x' corresponds to the group index and refers to the index of the modified register on a given bus.

- **Peripheral activation/deactivation management:** Enable/disable a peripheral or activate/deactivate specific peripheral features

Table 23: Peripheral activation/deactivation management

Name	Examples
<code>LL_PPP[_CATEGORY]_Action{Item}</code> <code>LL_PPP[_CATEGORY]_IsItemAction</code>	• <code>LL_ADC_Enable ()</code> • <code>LL_ADC_StartCalibration();</code> • <code>LL_ADC_IsCalibrationOnGoing;</code> • <code>LL_RCC_HSI_Enable ()</code> • <code>LL_RCC_HSI_IsReady()</code>

- **Peripheral configuration management:** Set/get a peripheral configuration settings

Table 24: Peripheral configuration management

Name	Examples
<code>LL_PPP[_CATEGORY]_Set{ or Get}ConfigItem</code>	<code>LL_USART_SetBaudRate (USART2, Clock, LL_USART_BAUDRATE_9600)</code>

- **Peripheral register management:** Write/read the content of a register/retrun DMA relative register address

Table 25: Peripheral register management

Name
<code>LL_PPP_WriteReg(__INSTANCE__, __REG__, __VALUE__)</code>
<code>LL_PPP_ReadReg(__INSTANCE__, __REG__)</code>
<code>LL_PPP_DMA_GetRegAddr (PPP_TypeDef *PPPx,{Sub Instance if any ex: Channel} , {uint32_t Propriety})</code>



The Propriety is a variable used to identify the DMA transfer direction or the data register type.

4 Cohabiting of HAL and LL

The low-layer APIs are designed to be used in standalone mode or combined with the HAL. They cannot be automatically used with the HAL for the same peripheral instance. If you use the LL APIs for a specific instance, you can still use the HAL APIs for other instances. Be careful that the low-layer APIs might overwrite some registers which content is mirrored in the HAL handles.

4.1 Low-layer driver used in standalone mode

The low-layer APIs can be used without calling the HAL driver services. This is done by simply including `stm32l4xx_ll_ppp.h` in the application files. The LL APIs for a given peripheral are called by executing the same sequence as the one recommended by the programming model in the corresponding product line reference manual. In this case the HAL drivers associated to the used peripheral can be removed from the workspace. However the STM32CubeL4 framework should be used in the same way as in the HAL drivers case which means that System file, startup file and CMSIS should always be used.



When the BSP drivers are included, the used HAL drivers associated with the BSP functions drivers should be included in the workspace, even if they are not used by the application layer.

4.2 Mixed use of low-layer APIs and HAL drivers

In this case the low-layer APIs are used in conjunction with the HAL drivers to achieve direct and register level based operations.

Mixed use is allowed, however some consideration should be taken into account:

- It is recommended to avoid using simultaneously the HAL APIs and the combination of low-layer APIs for a given peripheral instance. If this is the case, one or more private fields in the HAL PPP handle structure should be updated accordingly.
- For operations and processes that do not alter the handle fields including the initialization structure, the HAL driver APIs and the low-layer services can be used together for the same peripheral instance.
- The low-layer drivers can be used without any restriction with all the HAL drivers that are not based on handle objects (RCC, common HAL, flash and GPIO).

Several examples showing how to use HAL and LL in the same application are provided within `stm32l4` firmware package (refer to `Examples_MIX` projects).



1. When the HAL `Init/DeInit` APIs are not used and are replaced by the low-layer macros, the `InitMsp()` functions are not called and the MSP initialization should be done in the user application.
2. When process APIs are not used and the corresponding function is performed through the low-layer APIs, the callbacks are not called and post processing or error management should be done by the user application.
3. When the LL APIs is used for process operations, the IRQ handler HAL APIs cannot be called and the IRQ should be implemented by the user application. Each LL driver implements the macros needed to read and clear the associated interrupt flags.

5 HAL System Driver

5.1 HAL Firmware driver API description

5.1.1 How to use this driver

The common HAL driver contains a set of generic and common APIs that can be used by the PPP peripheral drivers and the user to start using the HAL.

The HAL contains two APIs' categories:

- Common HAL APIs
- Services HAL APIs

5.1.2 Initialization and de-initialization functions

This section provides functions allowing to:

- Initialize the Flash interface the NVIC allocation and initial time base clock configuration.
- De-initialize common part of the HAL.
- Configure the time base source to have 1ms time base with a dedicated Tick interrupt priority.
 - SysTick timer is used by default as source of time base, but user can eventually implement his proper time base source (a general purpose timer for example or other time source), keeping in mind that Time base duration should be kept 1ms since PPP_TIMEOUT_VALUES are defined and handled in milliseconds basis.
 - Time base configuration function (HAL_InitTick ()) is called automatically at the beginning of the program after reset by HAL_Init() or at any time when clock is configured, by HAL_RCC_ClockConfig().
 - Source of time base is configured to generate interrupts at regular time intervals. Care must be taken if HAL_Delay() is called from a peripheral ISR process, the Tick interrupt line must have higher priority (numerically lower) than the peripheral interrupt. Otherwise the caller ISR process will be blocked.
 - functions affecting time base configurations are declared as __weak to make override possible in case of other implementations in user file.

This section contains the following APIs:

- [*HAL_Init\(\)*](#)
- [*HAL_DeInit\(\)*](#)
- [*HAL_MspInit\(\)*](#)
- [*HAL_MspDeInit\(\)*](#)
- [*HAL_InitTick\(\)*](#)

5.1.3 HAL Control functions

This section provides functions allowing to:

- Provide a tick value in millisecond
- Provide a blocking delay in millisecond
- Suspend the time base source interrupt
- Resume the time base source interrupt
- Get the HAL API driver version
- Get the device identifier
- Get the device revision identifier