AN5044
Application note

STM32 standard peripheral library to STM32Cube low-layer migration

Introduction

Part of the STM32 value proposition is the availability of complete and full firmware libraries, providing developers with an initial framework to build their embedded application.

The central brick of this framework is the well-known Standard Peripheral Library (SPL), which is used by all the middleware components to access STM32 peripherals.

Over years, the STM32 portfolio has continued to grow and to offer to developers a variety of solutions for balancing cost, power and performance, notably the STM32Cube Low Layer (LL) drivers.

For designers of STM32 microcontroller applications, it is important to be able to easily upgrade the microcontroller type and/or a firmware solution to another. This migration is often needed, since the development for the SPL has stopped and when product requirements and specifications grow, putting extra demands on the variety of used peripherals.

This application note shows the steps to migrate from an existing STM32-based application developed under the STM32 SPL to any one of the other microcontroller types using the STM32Cube LL drivers.

This application note groups together all the most important information needed for a successful migration of STM32 SPL-based application to STM32Cube LL APIs usage.

It is composed of three sections:

- STM32 SPL vs. STM32Cube LL Architecture Overview: presenting a description of both solutions and a comparison between both.
- STM32 SPL to STM32Cube LL Manual Migration: detailing the steps to migrate manually an application based on standard peripheral library.
- STM32 SPL to STM32Cube LL Automatic Migration: presenting the SPL2LL-Converter migration tool as a solution for automatic migration of StdPeriph-based application.

This document applies to the STM32 SPL-supported microcontrollers listed in Table 1.

Table 1. Applicable products

Type	Product series
Microcontrollers	STM32F0, STM32F1, STM32F2, STM32F3, STM32F4, STM32F7, STM32L0, STM32L1 and STM32L4 Series.

Contents

1 STM32 SPL vs. STM32Cube LL architecture overview 5

1.1 STM32 SPL 5

1.1.1 Overview 5

1.1.2 Inclusion Model 5

1.2 STM32Cube Low Layer drivers 7

1.2.1 Overview 7

1.2.2 Inclusion model 9

1.2.3 APIs definition levels and classification 11

1.3 Summary 12

2 STM32 SPL to STM32Cube LL manual migration 14

2.1 STM32 SPL and STM32Cube LL APIs equivalence 14

2.1.1 NVIC interrupt configuration 14

2.1.2 Peripheral drivers 16

2.1.3 Migration cases 16

2.2 Project creation 18

3 STM32 SPL to STM32Cube LL automatic migration 20

3.1 SPL2LL-Converter migration tool specifications 20

3.1.1 Overview and features 20

3.1.2 SPL2LL-Converter migration tool block diagram 22

3.2 SPL2LL-Converter migration tool usage guidelines 23

3.2.1 Migration tool package architecture 23

3.2.2 SPL2LL-Converter migration tool 24

3.2.3 User application migration steps with initial application environment . . 27

3.2.4 User application migration steps using available LL templates 28

3.2.5 SPL2LL-Converter migration tool limitations 28

3.2.6 GUI application 29

4 Revision history 31

List of tables

Table 1. Applicable products . . . 1
Table 2. STM32 SPL application's files description . . . 7
Table 3. LL-supported peripherals . . . 8
Table 4. STM32Cube LL application's files description . . . 11
Table 5. STM32 SPL vs. STM32Cube comparison summary . . . 12
Table 6. Cortex-Mx equivalences between STM32 SPL and CMSIS core driver . . . 15
Table 7. Document revision history . . . 31

List of figures

Figure 1. Inclusion Model for STM32 SPL application . . . 6
Figure 2. Inclusion model for STM32Cube application . . . 10
Figure 3. STM32 SPL vs. STM32Cube LL API classification . . . 13
Figure 4. Analogy of user application before and after manual migration . . . 19
Figure 5. Automatic migration scenarios . . . 21
Figure 6. Block diagram of the migration tool . . . 22
Figure 7. Migration tool package tree . . . 23
Figure 8. Migration tool startup . . . 24
Figure 9. Tool display for a successful migration . . . 25
Figure 10. Tool display for an unsuccessful migration . . . 26
Figure 11. Analogy of user project before and after automatic migration . . . 26
Figure 12. SPL2LL-Converter migration tool graphic interface . . . 29
Figure 13. Logging after successful migration . . . 30

1 STM32 SPL vs. STM32Cube LL architecture overview

This section describes the architecture of both STM32 Standard Peripheral Library (SPL) and STM32Cube LL with a summary to comparing the main differences between the two solutions.

1.1 STM32 SPL

1.1.1 Overview

CMSIS

The STM32 SPL's CMSIS is composed of two layers: a Core Peripheral Access layer that contains name definitions, address definitions and helper APIs to access core Cortex-Mx registers and peripherals. And an STM32 Peripheral Access layer which provides definitions for all the peripheral registers, bit fields and memory mapping for the device.

STM32 SPL drivers

The library is built around a modular programming approach ensuring the independencies between the several components building the main application. It allows an easy porting on a large product range and evaluation boards with a minimum changes on the code of the common parts.

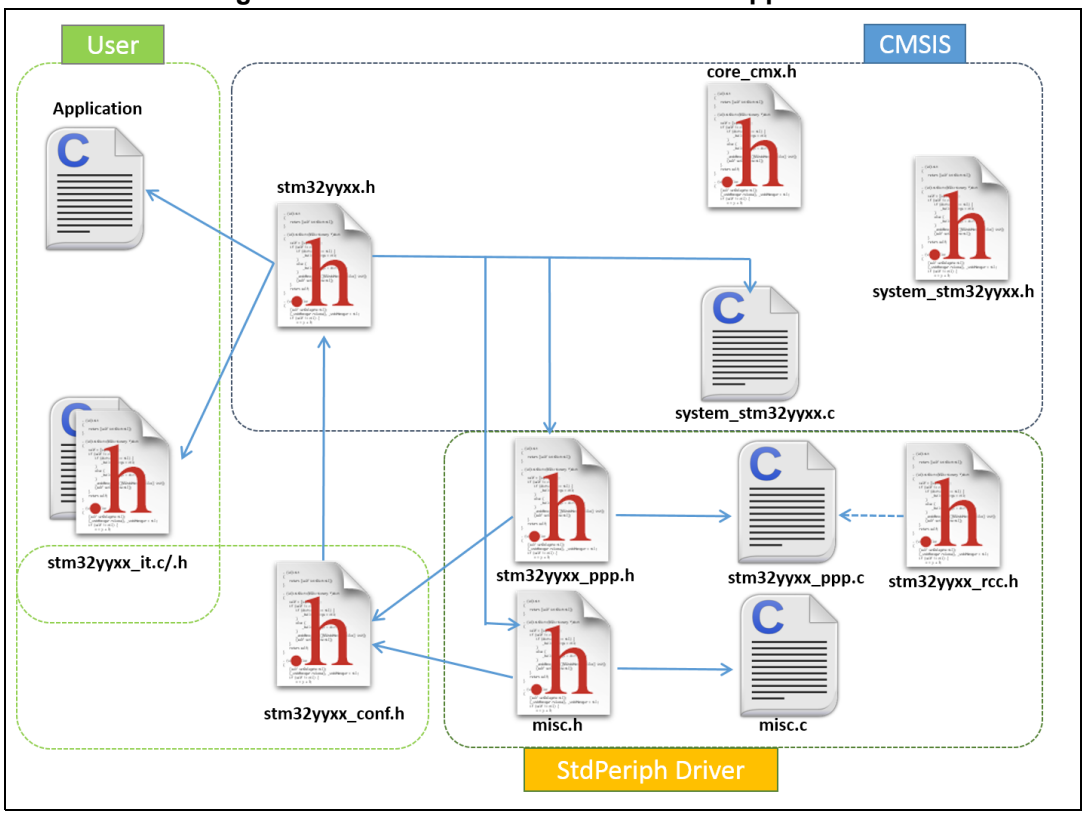
The STM32 SPL drivers provide drivers and header files for each peripheral. Each driver consists of a set of APIs covering all the peripheral functionalities.

It also implements run-time failure detection by checking the input values for all library functions. Such dynamic checking contributes towards enhancing the robustness of the software. Thus, it is suitable for user application development and debugging.

1.1.2 Inclusion Model

The files inclusion model for a default user application based on STM32 SPL is shown in *Figure 1*.

Figure 1. Inclusion Model for STM32 SPL application



Each STM32 embedded peripheral has a source code file stm32yyxx_ppp.c and a header file stm32yyxx_ppp.h. The stm32yyxx_ppp.c file contains all the firmware APIs required to use the PPP peripheral.

The stm32yyxx_conf.h file is considered a user file that is intended to be customized and tailored according to the used peripherals in the application. It is used to specify the set of parameters to interface with the library drivers before running any application.

Table 2 regroups and describes the above files referred by the user's STM32 SPL based application:

Table 2. STM32 SPL application's files description

-	File name	Description
STM32 SPL	stm32yyxx_conf.h	Peripheral's drivers configuration file. The user can enable or disable the peripheral header file inclusion by using the template. This file can also be used to enable or disable the Library run-time failure before compiling the firmware library drivers, through the pre-processor define USE_FULL_ASSERT
	stm32yyxx_ppp.h	Header file of PPP peripheral.
	stm32yyxx_ppp.c	Driver source code file PPP peripheral written in C language.
	stm32yyxx_it.h	Header file including all interrupt handlers prototypes.
	stm32yyxx_it.c	Template source file containing the interrupt service routine (ISR) for Cortex-Mx exceptions. User can add additional ISRs for the used peripherals (for the available peripheral interrupt handler's name, please refer to the startup_stm32yyxx.s).
CMISIS	stm32yyxx.h	CMSIS Cortex-Mx STM32yyxx device peripheral access layer header file. This file is the unique include that the application programmer is using in the source code. This file contains: – Configuration section that allows to select: –The device used in the target application –To use or not the peripheral's drivers in application code (i.e. code will be based on direct access to peripheral's registers rather than drivers API), this option is controlled by the #define USE_STDPERIPH_DRIVER –To change few application-specific parameters such as the HSE crystal frequency – Data structures and address mapping for all peripherals – Peripheral's registers declarations and bits definition – Macros to access peripheral's registers hardware
	system_stm32yyxx.h	CMSIS Cortex-Mx STM32yyxx devices peripheral access layer system header file
	system_stm32yyxx.c	CMSIS Cortex-Mx STM32yyxx devices peripheral access layer system source file

1.2 STM32Cube Low Layer drivers

1.2.1 Overview

The Low Layer (LL) drivers are a part of the STM32Cube firmware HAL and 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.

The Low Layer (LL) drivers are designed to offer:

- 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 in the STM32 peripherals. *Table 3* lists the STM32 embedded peripherals covered by the Low Layer scope:

Table 3. LL-supported peripherals

Peripherals		STM32Cube LL support
System	FLASH	No
	EXTI	Yes
	GPIO	Yes
	DMAXs	Yes
	PWR	Yes
	RCC	Yes
	Cortex	Yes
	SYSCFG	Yes
	NVIC(1)	No (already covered by CMSIS)
Analog	ADC	Yes
	DAC	Yes
	COMP	Yes
	OPAMP	Yes
	DFSDM	No
Timers	RTC	Yes
	TIM	Yes
	LPTIM	Yes
	HRTIM	Yes
	WWDG	Yes
Cryptography	CRC	Yes
	CRYP	No
	HASH	No
	RNG	Yes

Table 3. LL-supported peripherals (continued)

Peripherals		STM32Cube LL support
Basic connectivity	I2C	Yes
	UART/USART/LPUART	Yes
	SWPMI	Yes
	SPI/I2S	Yes
	SDMMC(SDIO)	No

1. NVIC is covered in STM32 SPL by the misc.h/.c driver

The LL APIs reflect exactly the hardware capabilities and provide one-shot operations. The operations must be called following the programming model described in the microcontroller line reference manual.

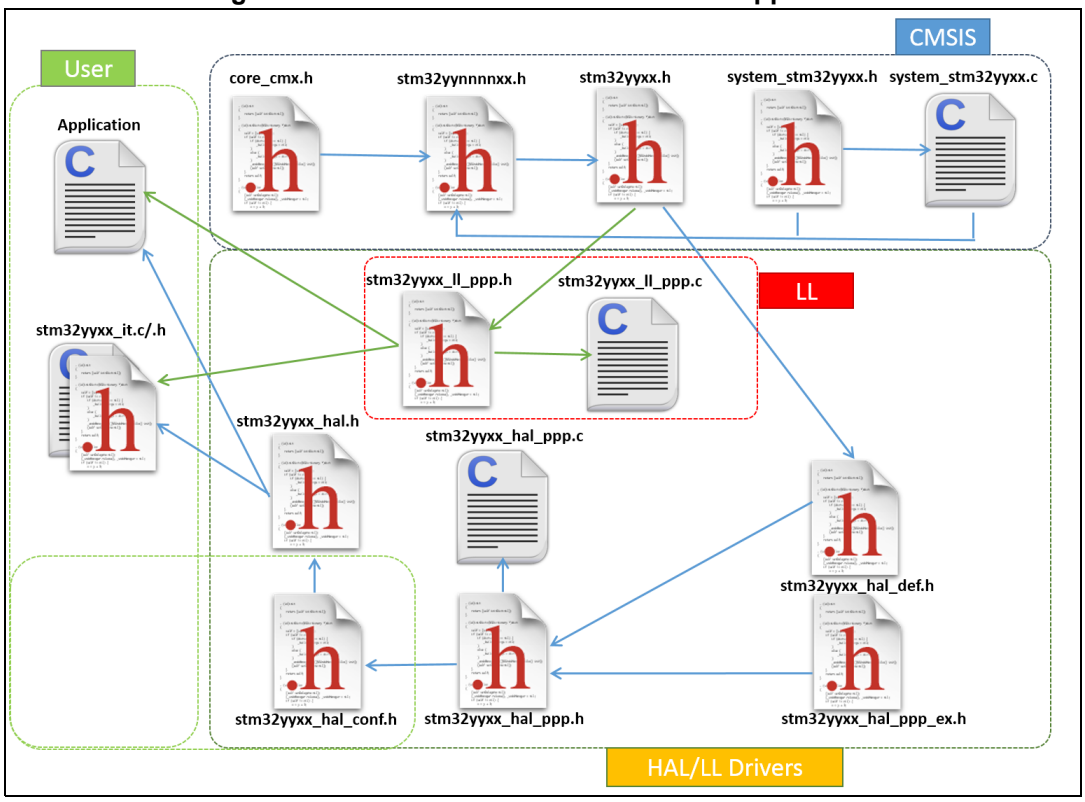
As a result, the LL services do not implement any processing and do not require any additional memory resources to save their states, counter or data pointers: all the operations are performed by changing the associated peripheral registers content.

All the Low Layer drivers are called through their physical instances (Peripheral registers structures mapped on the peripherals base registers) and are given in one module for each physical peripheral located in a separate header file.

1.2.2 Inclusion model

As an STM32Cube application, the user is exposed to different files inclusion models according the adopted drivers. *Figure 2* shows the inclusions via either STM32Cube HAL or Low layer drivers.

Figure 2. Inclusion model for STM32Cube application



Through what the STM32Cube package provides, there are three categories for an STM32Cube application:

- HAL application: the application is based solely on HAL drivers and following an inclusion as highlighted by the blue arrows.
- LL application: Since the LL drivers are standalone. The user can develop his application using only LL drivers, referring to them only from his source files. Note that LL drivers cannot include each other but they have to include only the CMSIS device file. Thus, there is no need for a configuration file, and the user must include the used drivers in the entry point file on his application.
- MIX application: the application in which the user calls on HAL and LL drivers and uses both of their APIs to develop his code. Drivers are independent from each other and there is no interference in their inclusions approaches.

Table 4. STM32Cube LL application's files description

File name		Description
HAL	stm32yyxx_hal.c	This is the common part of the HAL initialization
	stm32yyxx_hal.h	This is the header file of the common part of the HAL initialization
	stm32yyxx_hal_ppp.c	This is the c-source file of the PPP driver. The PPP driver is a standalone module, to be used in a project the user should enable the correspondent define USE_HAL_PPP_MODULE in the configuration file
	stm32yyxx_hal_ppp.h	This is the header file of the PPP driver.
	stm32yyxx_hal_ppp_ex.h/.c	These files are an extension of the standard set of APIs within the driver.
	stm32yyxx_def.h	Common HAL resources such as common define statements, enumerations, structures and macros.
LL	stm32yyxx_ll_ppp.h/c	This is the h-source file of the PPP low layer driver. The low layer PPP driver is a standalone module. To be used, the application should include the stm32yyxx_ll_ppp.h
CMSIS	stm32yyxx.h	CMSIS Cortex-Mx STM32yyxx series peripheral access layer header file.
	stm32yynnnnxx.h	CMSIS Cortex-Mx STM32yyxx device peripheral access layer header file.
	system_stm32yyxx.h/.c	This file contains APIs which are called at startup just after reset and before branching to the main program.

Contrary to the HAL drivers, the low level ones are not built on process model but rather in simple access operations on registers. Thus, the low level layer has no configuration file.

1.2.3 APIs definition levels and classification

The Low Layer drivers' purpose is to provide an abstraction APIs level that covers the STM32 snippets APIs and Standard Peripheral drivers functionalities.

The low layer drivers provide a complementary set of basic APIs allowing the customization or the replacement of high level processes.

Each Low layer peripheral driver should cover the following three APIs levels:

- Level 1: The LL_PPP_WriteReg() / LL_PPP_ReadReg() (redirection of CMSIS registers operations).
- Level 2: One shot operations APIs (atomic) which are sorted as follow:
  - Peripherals activation/deactivation management: enable or disable a peripheral block, sub-block, or an associated feature.
    Example: LL_PPP_Disable(PPPx)
  - Peripheral functional operations management: start or launch a peripheral operation or set a peripheral in a functional state.
    Example: LL_PPP_Action()
  - Helper operations.
    Example: IS_PPP_State(PPPx)

– Specific Interrupt and Status flags management: handle status and register flags operations (get, Clear, Enable, Disable) for a single item.
Example: LL_PPP_ClearFlag_XX()

These APIs are provided in the “stm32yyxx_ll_PPP.h”.

- Level 3: Global configuration and initialization functions that cover full standalone operations on relative peripheral registers which are provided in “stm32yyxx_ll_PPP.c”.

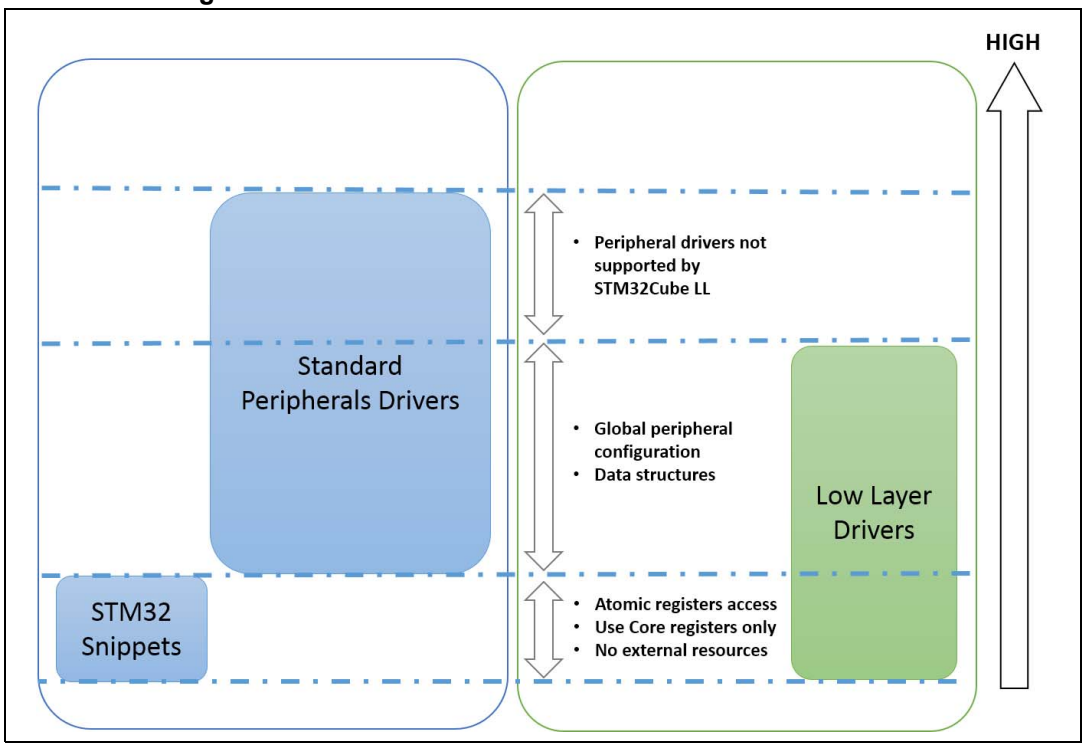
1.3 Summary

Table 5 shows an overall comparison between the STM32 SPL and STM32Cube CMSIS and drivers components:

Table 5. STM32 SPL vs. STM32Cube comparison summary

-	STM32 SPL	STM32Cube
CMSIS	– Single header file (registers and bits definition) common for all devices in a single series. The user needs to sort out peripherals present in the considered device. – Differences in Interrupt handlers’ names between products lines need to be managed by the user application. – system_stm32xx.c: configures the system clock before jumping to the main()	– Header file per device among a given series presenting the available features in the product and providing an organized layout for registers and bits definitions – Using aliases to abstract naming differences when migrating from STM32 SPL, such as the interrupt handlers’ names. – system_stm32xx.c: system clock configuration is no more implemented in this file. The user is required to perform it in the main file.
Peripheral drivers	– APIs are managing simple registers IO operation. Process and error management must be done by the user application. – Peripheral-oriented implementation. Compatibility loss in case of major updates on the peripheral features.	– LL APIs are functional-oriented with direct/atomic register access. – LL APIs mirroring the hardware capabilities of the STM32 series. – LL provides fully standalone operations on relative peripheral registers. – Compatibility vs. STM32 SPL: not compatible. New set of APIs defined for Low Layer.

Figure 3. STM32 SPL vs. STM32Cube LL API classification



The STM32Cube LL drivers offer a new set of inline functions allowing a direct and atomic register access substituting the code snippets and the standard peripheral driver. Its independency from the HAL drivers gives the user a standalone usage covering the supported peripherals and their features with minimal code footprint and memory resources. A full STM32 SPL to STM32Cube LL migration is therefore feasible.

2 STM32 SPL to STM32Cube LL manual migration

This section describes how to migrate an application developed based on an STM32 SPL to an equivalent application deploying STM32Cube LL that could be either on the same target device or a different STM32 series.

2.1 STM32 SPL and STM32Cube LL APIs equivalence

As seen in the previous chapter, the Low Layer drivers provide APIs and methods on different levels. They are ranging from elementary and atomic registers accesses which need no external resources to higher level configuration functions. The LL drivers can provide fully standalone operations and covers the STM32 SPL APIs.

In this section, an equivalence between STM32 SPL and STM32Cube LL will be brought in details.

2.1.1 NVIC interrupt configuration

The STM32xx_StdPeriph_Lib solution allows two different approaches to handle CMSIS Core features:

1. CMSIS Core Cortex-Mx driver through core_cmx.h file
2. STM32 Standard Peripheral driver through misc.h/c files.

However, the STM32Cube LL solution allows only the approach through CMSIS Core Cortex-Mx driver as the CMSIS core_cmx.h file provides a full set of macros covering almost for CMSIS core Cortex-Mx feature.

Two kinds of migration are possible:

- User familiarized to use CMSIS Core Cortex-Mx driver to manage interrupt, in this case there are no impact on user application.
- User familiarized to use misc.c/.h driver, the following steps should be followed for migrate:
  - Extract the configuration found in the NVIC_InitTypeDef initialization structure,
  - Set the priority groupings using NVIC_SetPriorityGrouping() API
  - Encode the priority using NVIC_EncodePriority() API
  - Set the new IRQn priority using NVIC_SetPriority() API
  - Enable or disable IRQn External interrupts.

Table 6 resuming the Cortex-Mx equivalences between STM32 SPL and CMSIS Core driver:

Table 6. Cortex-Mx equivalences between STM32 SPL and CMSIS core driver

STM32 SPL (misc.c/.h)	CMSIS (core_cmx.h)	Comment
void NVIC_PriorityGroupConfig(uint32_t NVIC_PriorityGroup);	__STATIC_INLINE void NVIC_SetPriorityGrouping(uint32_t PriorityGroup);	-

void NVIC_Init(NVIC_InitTypeDef* NVIC_InitStruct);	__STATIC_INLINE void NVIC_SetPriorityGrouping(uint32_t PriorityGroup);
	__STATIC_INLINE uint32_t NVIC_EncodePriority (uint32_t PriorityGroup, uint32_t PreemptPriority, uint32_t SubPriority);
	__STATIC_INLINE void NVIC_SetPriority(IRQn_Type IRQn, uint32_t priority)
	__STATIC_INLINE void NVIC_EnableIRQ(IRQn_Type IRQn);
	or
	__STATIC_INLINE void NVIC_DisableIRQ(IRQn_Type IRQn);
		NVIC_InitTypeDef structure isn't defined on STM32Cube driver,

void NVIC_SetVectorTable(uint32_t NVIC_VectTab, uint32_t Offset);	No equivalence	SCB->VTOR = NVIC_VectTab | (Offset & (uint32_t)0x1FFFFF80); 1- NVIC_VectTab: specifies if the vector table is in RAM or FLASH memory. This parameter can be one of the following values: - NVIC_VectTab_RAM: Vector Table in internal SRAM. - NVIC_VectTab_FLASH: Vector Table in internal FLASH. 2- Offset: Vector Table base offset field. This value must be a multiple of 0x200.

Table 6. Cortex-Mx equivalences between STM32 SPL and CMSIS core driver (continued)

STM32 SPL (misc.c/.h)	CMSIS (core_cmx.h)	Comment
void NVIC_SystemLPConfig(uint8_t LowPowerMode, FunctionalState NewState);	No equivalence	1- NewState = ENABLE SCB->SCR |= LowPowerMode; 2- NewState = DISABLE SCB->SCR &= (uint32_t)(~(uint32_t)LowPowerMode); LowPowerMode: Specifies the new mode for the system to enter low power mode. This parameter can be one of the following values: - NVIC_LP_SEVONPEND: Low Power SEV on Pend. - NVIC_LP_SLEEPDEEP: Low Power DEEPSLEEP request. - NVIC_LP_SLEEPONEXIT: Low Power Sleep on Exit.
void SysTick_CLKSourceConfig(uint32_t SysTick_CLKSource);	__STATIC_INLINE uint32_t SysTick_Config(uint32_t ticks);	-

2.1.2 Peripheral drivers

The Low Layer library supports most of system, analog, timers, cryptography and basic connectivity peripherals.

The equivalence can be a simple API matching another, or can be a call of several LL APIs in case of a complex processing. The user may be required to thoughtfully inspect the manipulated registers within a given API to find out the corresponding LL APIs. The Low Layer drivers provide also sets of direct structure and peripheral initialization and configuration functions matching with the existing in STM32 SPL, sparing the effort of a consecutive call for LL APIs.

The SPL2LL-Converter migration tool, available at ST website, comes with tabulated databases for STM32 peripherals supported by the STM32Cube LL drivers. These tables provide with the direct equivalent in LL and mention the availability of some features among STM32 series.

These databases are present within the "SPL2LL_User_Manual.chm" file that we find in SPL2LL-Converter package.

2.1.3 Migration cases

During the code migration from STM32 SPL to STM32Cube LL drivers, the user may cross different approaches to write an equivalent LL code. These have been sorted into most common API cases listed as below:

- Change is only in function name, parameters order is kept with the equivalent LL defines.

GPIO_PinLockConfig(GPIOA, GPIO_Pin_0) ↔ LL_GPIO_LockPin(GPIOA, LL_GPIO_PIN_0)

- Updated function name depending on one of the parameters values, order is preserved with the equivalent LL defines:
  GPIO_WriteBit(GPIOA, GPIO_Pin_0, Bit_SET) ↔ LL_GPIO_SetOutputPin(GPIOA, LL_GPIO_PIN_0)
- Updated function name, parameters order may change in the equivalent LL function:
  TIM_ETRClockMode1Config(TIM1, TIM_ExtTRGPSC_DIV2, TIM_ExtTRGPolarity_Inverted, 0);
  ↔
  LL_TIM_ConfigETR(TIM1, LL_TIM_ETR_POLARITY_INVERTED, LL_TIM_ETR_PRESCALER_DIV2, LL_TIM_ETR_FILTER_FDIV1);
- Updated function name with the use of LL parameters having functional similarity and no API equivalence:
  ADC_SelectDifferentialMode(ADC1, ADC_Channel_1, ENABLE)
  ↔
  LL_ADC_SetChannelSingleDiff(ADC1, LL_ADC_CHANNEL_1, LL_ADC_DIFFERENTIAL_ENDED)
- Updated function name, adding new parameters:
  CRC_GetIDRegister() ↔ LL_CRC_Read_IDR(CRC)
- Many STM32 SPL functions migrated into one LL function, parameters may change following STM32 SPL drivers required parameters:
  SPI_NSSInternalSoftwareConfig(SPI1, SPI_NSSInternalSoft_Set);SPI_SSOutputCmd(SPI1, DISABLE);
  ↔
  LL_SPI_SetNSSMode(SPI1, LL_SPI_NSS_SOFT);
- One STM32 SPL function migrated to more than one LL function, parameters shared between the equivalent functions:
  ADC_RegularChannelConfig(ADC1, ADC_Channel_1, 1, ADC_SampleTime_2Cycles5);
  ↔
  LL_ADC_REG_SetSequencerRanks(ADC1, LL_ADC_REG_RANK_1, LL_ADC_CHANNEL_1);
  LL_ADC_SetChannelSamplingTime(ADC1, LL_ADC_CHANNEL_1, LL_ADC_SAMPLINGTIME_2CYCLES_5);
- More than one STM32 SPL function migrated into more than one LL functions, parameters may be added, reduced and shared within the equivalent functions:
  RTC_CalibOutputConfig(RTC_CalibOutput_512Hz);
  RTC_CalibOutputCmd(ENABLE);
  ↔
  LL_RTC_DisableWriteProtection(RTC);
  LL_RTC_CAL_SetOutputFreq(RTC, LL_RTC_CALIB_OUTPUT_512HZ);
  LL_RTC_EnableWriteProtection(RTC);
- Updated function name, parameters are populated from the STM32 SPL structures content. Values must be retrieved from the structures and assigned as parameters:
  RTC_SetTime(RTC_Format_BCD, &RTC_TimeStruct);
  ↔

LL_RTC_TIME_Config(RTC, LL_RTC_TIME_FORMAT_AM_OR_24, RTC_TimeStruct.Hours, RTC_TimeStruct.Minutes, RTC_TimeStruct.Seconds);

- No explicit API equivalent on LL. The user need to migrate using direct register accesses:
  NVIC_SetVectorTable(NVIC_VectTab_FLASH, 0x1000);
  ↔
  SCB->VTOR = 0x08000000U | (0x1000U & 0x1FFFFF80U);

2.2 Project creation

Before migrating to STM32Cube Low Layer drivers, make sure to have initially a properly working application to avoid unwanted issues.

To update application code to use the STM32Cube Low Layer drivers and to run on a given STM32 MCU series, the user have the choice to opt for one of two approaches, as detailed below:

- **Keeping old application environment**
  1. In application repository, create a directory in which the user add own target device's STM32Cube drivers. This should contains two folders:
     – CMSIS: presents the new CMSIS files coming with the STM32Cube.
     – STM32yyxx_HAL_Drivers: containing the required drivers.
  2. Update toolchain configuration parameters:
     – Linker configuration: Device connections and Flash memory loader. These files are provided with the latest version of toolchain that supports the target device. For more information, please refer to toolchain documentation.
     – Project files: Add the LL drivers' source files and CMSIS "system_stm32yyxx.c".

     Note: - There is no configuration header file in the Low Layer environment. The call for drivers is done in the entry point file of application. In case of migration needing APIs from STM32Cube HAL, the user can add "stm32yyxx_hal_conf.h" to refer to HAL drivers.

     - The default SPL system clock configuration already performed in system_stm32yyxx.c should be updated manually in top of migrated main.c file by adding private SystemClock_Config() API. This API is already existing in the STM32Cube LL_Template project for the targeted STM32.

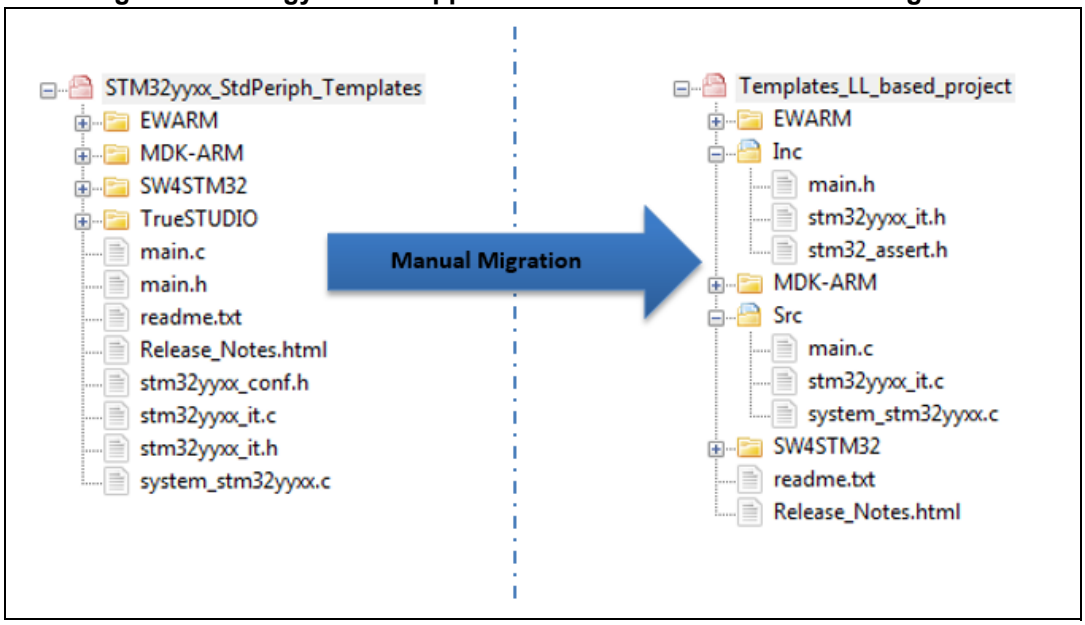
     – Project configuration parameters: Add the necessary include paths for LL drivers and CMSIS files and update the pre-processor symbols with the keywords required by the used drivers and target device.
  3. Rewrite the part of application code using the STM32Cube LL APIs. For migration, the user have to look-up for the STM32 SPL and STM32Cube LL equivalence provided within the SPL2LL-Converter migration tool.
- **Using the available Low Layer templates**
  The faster approach is start from a ready LL template project supporting the commonly used toolchains. It spares the user the time of going through the steps of toolchain configuration and eliminates the risk of erroneous updates. Thus, user can get straight into coding application with the LL APIs.
  The STM32Cube firmware package provides an LL template project tailored for each

target device within a given STM32 series. This gives user the freedom to even update the current hardware platform.

Note: The STM32Cube comes with a rich set of examples (varying from 70 to 90 in total) using either LL drivers only or mixed between LL and HAL, demonstrating how to use the different peripherals. These examples commonly share the same application scenarios with those provided by STM32 SPL, which makes it easy for the developer to quickly compare both solutions and get started with STM32Cube LL.

Figure 4 illustrates the successful migration of an application based on an STM32 SPL Template project to an STM32Cube Template_LL project.

Figure 4. Analogy of user application before and after manual migration



3 STM32 SPL to STM32Cube LL automatic migration

This section presents the steps for an automatic migration using the SPL2LL-Converter migration tool. Throughout this section, the user will have an overview about the migration tool and its features, its series and peripherals coverage and the detailed guidelines to successfully run it.

3.1 SPL2LL-Converter migration tool specifications

3.1.1 Overview and features

The STM32 SPL2LL-Converter migration tool is a solution developed by ST using Perl programming language. The purpose of this tool is to upgrade an existing application developed via STM32 SPL to an equivalent application based on the STM32Cube LL drivers. The process through this tool covers the migration of the user's whole source files.

It starts initially from the user source files, scans each to find and extract STM32 SPL APIs and then looks up for equivalence in a given database and creates equivalent user source files based on STM32Cube LL drivers.

The STM32 SPL2LL-Converter migration tool features:

- Supports all STM32 series supported by STM32 SPL and STM32Cube LL
  - From STM32 SPL drivers: STM32F0, STM32F10, STM32F2, STM32F30, STM32F37, STM32F4 and STM32L1 series.
  - To STM32Cube LL drivers: STM32F0, STM32F1, STM32F2, STM32F30, STM32F37, STM32F4, STM32F7, STM32L0, STM32L1 and STM32L4 series.
- Supports all peripherals covered by LL framework
  - System peripherals (RCC, PWR, FLASH, GPIO, EXTI, DMA and NVIC)
  - Analog peripherals (ADC, DAC, COMP, OPAMP and CRS)
  - Timers (RTC, TIM, LPTIM, HRTIM, IWDG and WWDG)
  - Communication peripherals (I2C, USART, LPUART and SPI/I2S)
  - Cryptography peripherals (CRC and RNG)
- Compatible with Windows®, Linux® and MacOS®

The tool is able to handle the differences between STM32 series caused by the variations in the available peripherals versions. This sets the need for two migration cases to be performed as presented in *Figure 5*.

Figure 5. Automatic migration scenarios

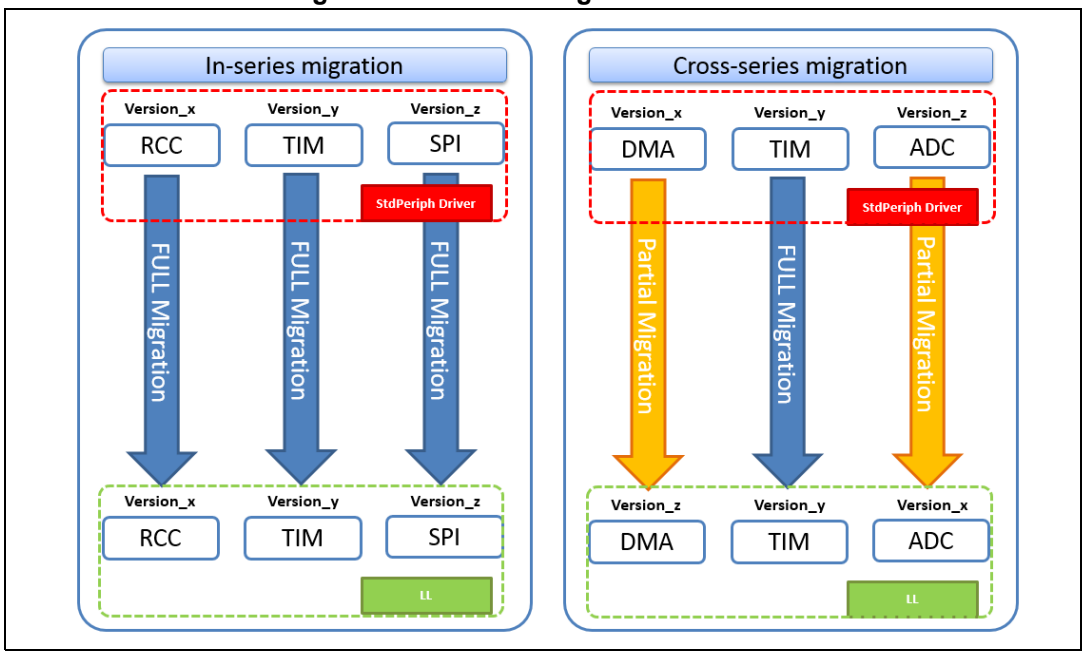


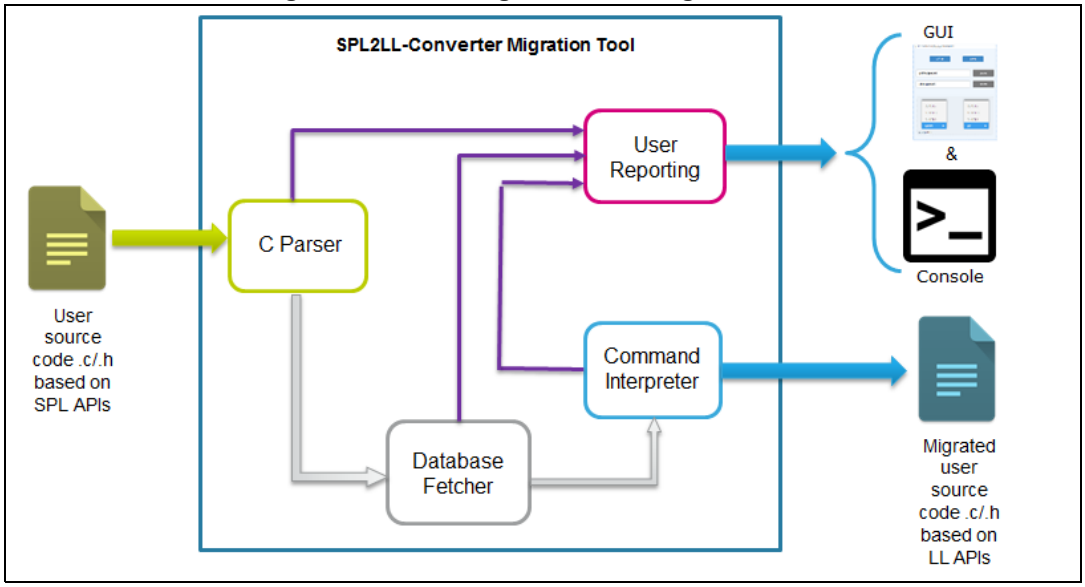
Figure 5 gives an insight about the two migration scenarios:

- In-Series migration: Migration performed within the same series (e.g. from STM32 SPL STM32F4 to STM32Cube LL STM32F4), in which each peripheral's APIs are fully migrated to LL.
- Cross-Series migration: Migration performed across STM32 series (e.g. from STM32 SPL STM32F4 to STM32Cube LL STM32L4). In this scenario, we can either have:
  - Partial-code migration: The tool partially converts the source code due to potential absent features from a peripheral version to another. Migration considers only common features.
  - Full-code migration: All STM32 SPL APIs are fully converted for a given peripheral if the version is the same in the target device.

3.1.2 SPL2LL-Converter migration tool block diagram

The tool is developed in Perl programming language following a modular architecture. The overall process is performed through four modules as presented in *Figure 6*.

Figure 6. Block diagram of the migration tool



- **C Parser**
  Firstly, the tool takes the user's source files as input. Through this module, it allows to parse and scan them, line by line, and extracts source code.
  The tool has the ability to differentiate between a user's comment, resulting in copying it into the target files as it is, and the actual code.
  The STM32 SPL based code is sorted and interpreted under 4 types:
  - Literals: grouping the STM32 SPL defines and structures' naming.
  - Structures: grouping the different structures fields in STM32 SPL drives.
  - Functions: comprising all the functions provided by the STM32 SPL drivers.
  - Includes: identifying the files inclusions within the code.
- **Database fetcher**
  Within the package, the STM32 SPL2LL-Converter migration toll comes with a full set of XML files for the STM32 peripherals supported by the STM32Cube LL drivers. This set constitutes the XML database that serves as a source from which the migration tool extract the LL equivalence for the initially parsed code.
  The architecture of the database matches the way the previous C Parser module interprets data.
  The fetcher module parses all the database and extracts the matching LL API. It returns for structures and literals sections only the equivalent expression, and for features section the STM32Cube LL function naming with the corresponding commands directing the migration process.
- **Command Interpreter**
  Once a STM32 SPL API has been spotted in the database, the interpreter module executes the built-in commands provided with the STM32Cube LL equivalent API in

order to correctly migrate. These commands generally control parameters numbers, order and default values.

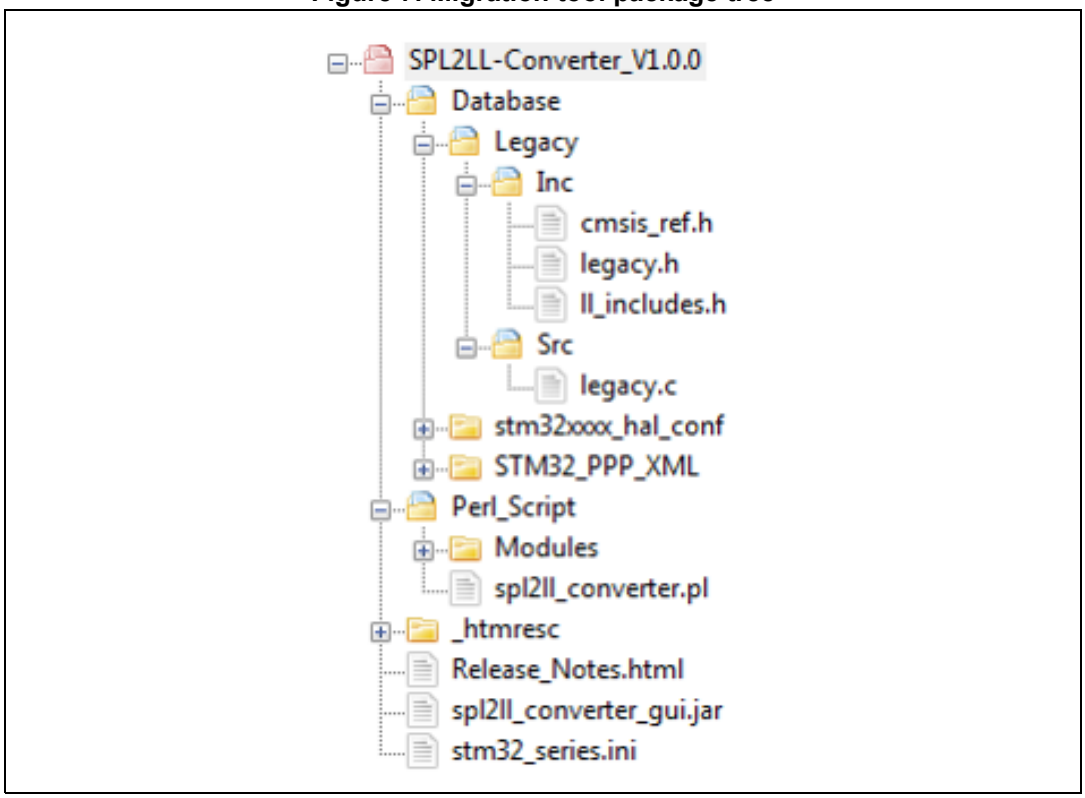
- **User Reporting**
  Along with the migration process, information sharing can be required. For this, the user reporting module provides all information regarding the migration advancement. The reporting is about the files migration statuses, warnings, errors and statistics. The reporting is done as a console output during the execution of the tool and also as a generation of a log file that is available in the target project folder.

3.2 SPL2LL-Converter migration tool usage guidelines

3.2.1 Migration tool package architecture

The SPL2LL-Converter migration tool comes in a package organized as presented in *Figure 7*.

Figure 7. Migration tool package tree



The migration package is composed of the following items:

- Database folder: where the database is located, containing:
  a) STM32_PPP_XML folder: contains the set of XML files; one file for each supported peripheral.
  b) stm32xxxx_hal_conf folder: contains the STM32xxxx HAL configuration files available within the latest STM32Cube_FW_YY packages. They are required to

replace the STM32 SPL stm32yyxx_conf.h in order to maintain compatibility with the STM32Cube firmware.

c) Legacy folder: contains the STM32 SPL APIs where no direct equivalence in LL is available, the same STM32 SPL APIs implementation are copied inside.

- Perl Script folder: in which to find the "spl2ll_converter.pl" Perl file and its required components.
- _htmresc folder: containing the STM32 SPL-STM32Cube LL APIs equivalence presented in HTML files for each supported peripheral.
- Release_Notes file: giving a brief about the migration tool and overall guidelines.
- spl2ll_converter_gui.jar: executable for a GUI application.

3.2.2 SPL2LL-Converter migration tool

Before running the STM32 SPL2LL-Converter Migration tool, it is mandatory that Perl is installed on the host. The use of ActiveState® Perl version 5.24.1 or more recent is recommended. ActivePerl® can be downloaded from ActiveState® website at https://www.activestate.com.

To run the migration tool, the user have to type the command described as following:

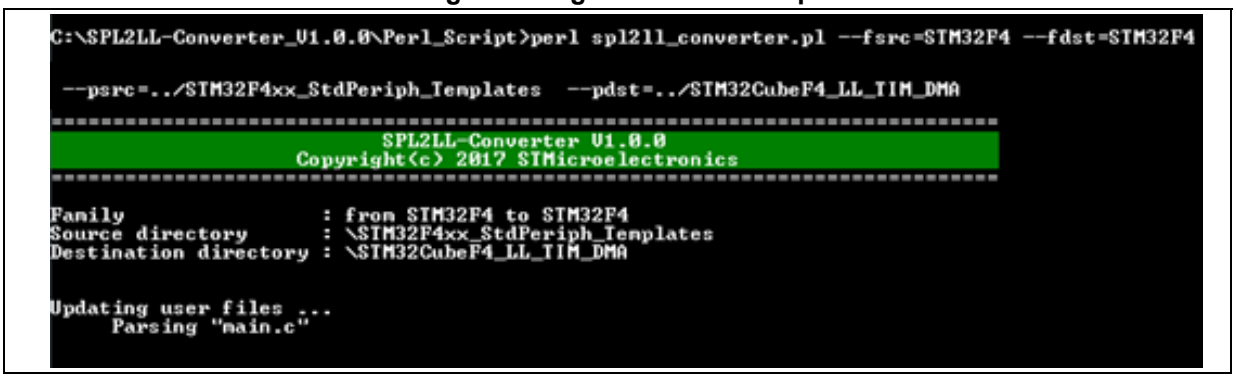
```
perl spl2ll_converter.pl
--fsrc=<STM32_Source_Series>
--fdst=<STM32_Destination_Series>
--psrc=<STM32_Source_Directory>
--pdst=<STM32_Destination_Directory>
```

The arguments represent the sources and destination series, as well as the required paths:

```
--fsrc: STM32 series/device source
--fdst: STM32 series/device target destination
--psrc: User directory path for STM32 SPL-based source code
--pdst: User directory path for output code
```

Figure 8 is a terminal screenshot showing the tool startup display:

Figure 8. Migration tool startup



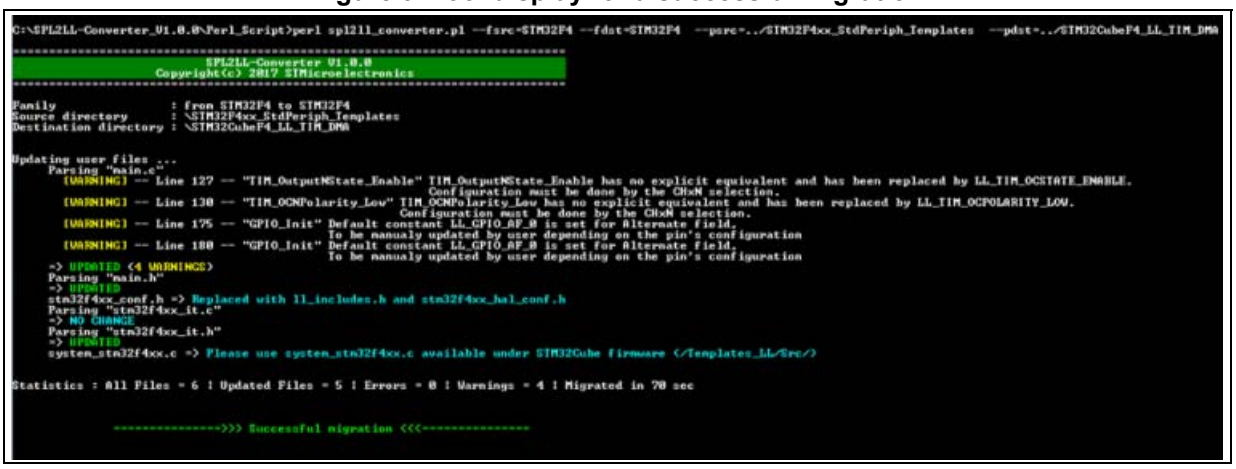
Once execution is done, the console output is printed on the terminal. It represents the overall details of the tool, the migration project, details about the user files and the reporting during the process.

The following screenshots present two separate migration scenarios of the previous project:

- The first one is an In-series migration from STM32F4 to STM32F4
- The second is a Cross-series migration from STM32F4 to STM32F30.

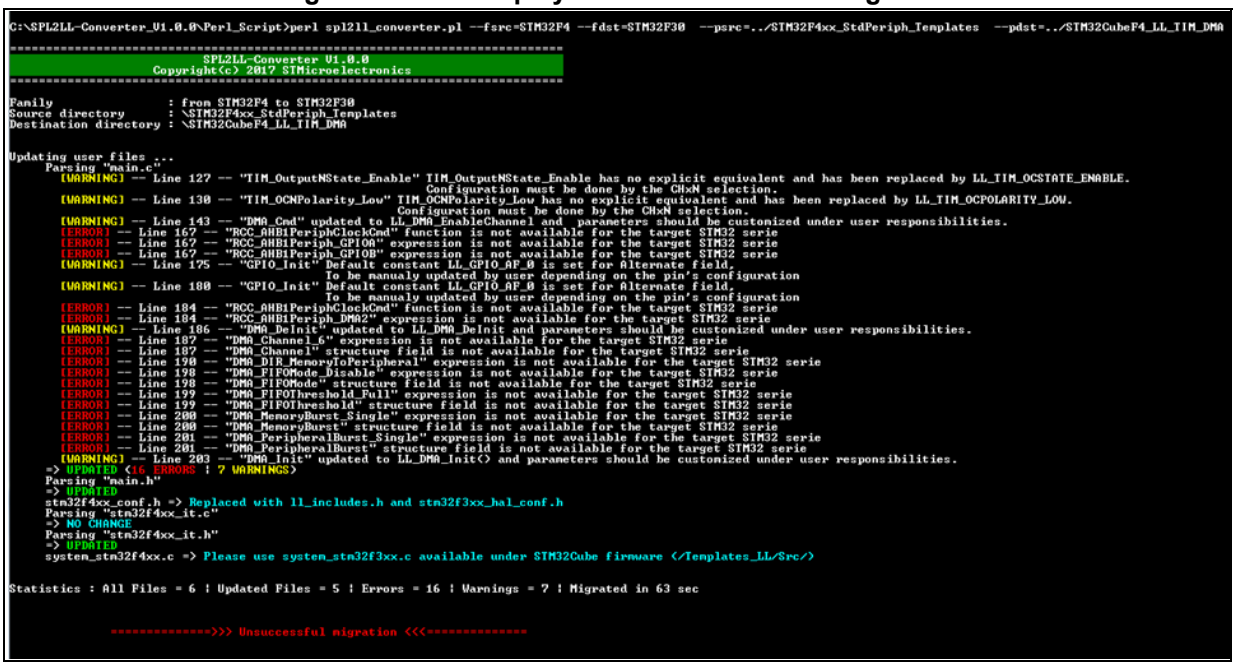
The screenshots highlight the different parts of the user display during the migration:

Figure 9. Tool display for a successful migration



1. tool header: showing the tool title and current version
2. Migration parameters: source and target series and the source and destination paths
3. Status of the user files: the status can be "UPDATED", "NO CHANGE" or a specific user message for certain files

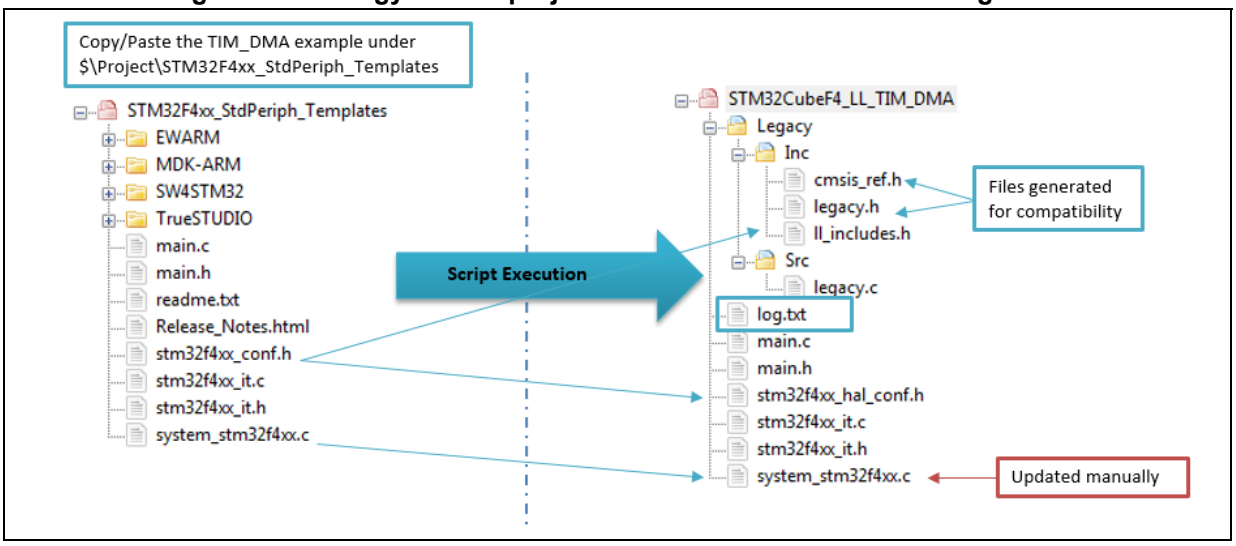
Figure 10. Tool display for an unsuccessful migration



4. tool header: showing the tool title and current version
5. Migration parameters: source and target series and the source and destination paths
6. Status of the user files: the status can be "UPDATED", "NO CHANGE" or a specific user message for certain files

Figure 11 is an example of a migration process applied on an STM32F4 StdPeriph_Lib TIM_DMA example:

Figure 11. Analogy of user project before and after automatic migration



After execution, the tool keeps the same overall hierarchy as the one of the initial project and applies some updates. *Figure 11* highlights these updates:

- Only source files (.h/.c) are migrated into the new LL project.
- Creation of a "Legacy" folder containing source files required for the migrated APIs not supported by LL. The "legacy.h/.c" present the STM32 SPL code, "ll_includes.h" is a part of the "stm32yyxx_conf.h" equivalence after migration and "cmsis_ref.h" contains the CMSIS includes for all the STM32 series and is required by "legacy.h".
- The migration of "stm32yyxx_conf.h" results its substitution by the files "stm32yyxx_hal_conf.h" and "ll_includes" for STM32Cube compliancy.
- Creation of a "log.txt" file presenting the global reporting during the migration process.

In case of need for a manual intervention, the user can check the generated log file "log.txt" in which he can find the printed warnings and use them as guidelines to adjust the LL source files.

However, when migration is not successful due to unavailable features in the target device, the user must then intervene by correcting the reported errors.

The STM32 SPL-based application must therefore be aligned to be compatible with the target present features.

3.2.3 User application migration steps with initial application environment

This section describes the automatic migration for the first approach detailed in *Section 2: STM32 SPL to STM32Cube LL manual migration*.

Prerequisites

The user asked to download the latest version of STM32Cube package of target series where to find the LL and CMSIS drivers.

The initial STM32 SPL based project should be error-free and fully functional, to avoid unwanted migration troubles and to make it easier for manual updates in case of partial migration.

Migration guide lines

Since the tool migrates only the source files, it makes it independent from the toolchain used. Therefore, the user are asked to:

- **Tool execution (automatic steps)**
  After setting up the environment for migration, the process starts as following:

1. Launch terminal and point to the tool location.
2. Run the SPL2LL-Converter migration tool.

- **User projects customization (manual steps)**

1. Update the old CMSIS folder by the latest version available under STM32Cube_FW_YY packages.
2. Add STM32xx_HAL_Driver folder containing the LL/HAL source/header files available within the same package.
3. Create new target project including the required LL/CMSIS drivers.
4. Setup configuration (pre-processor, start up file, MCU selection, Debugger).
5. Update the "system_stm32yyxx.c" bringing it from "Templates_LL" folder from own target STM32Cube firmware package.

Note: User should ensure to reproduce the system clock configuration already performed in system_stm32yyxx.c by implementing manually the SystemClock_Config() API in top of migrated main.c file.

6. Add user source files in the project.
7. Add "legacy.c" file available under the generated $Legacy/Src folder.

3.2.4 User application migration steps using available LL templates

This section describes the automatic migration for the second approach detailed in *Section 2: STM32 SPL to STM32Cube LL manual migration*.

The user can opt for using the STM32Cube LL templates as a target project. With this approach, a template project is ready to use.

There is no need to update CMSIS files or add the Low Layer drivers, as the template project refer to them by default.

So, the user have to:

- **Tool execution (automatic steps)**

1. Launch own terminal and point to the tool location.
2. Run the SPL2LL-Converter migration tool.

- **User projects customization (manual steps)**

3. Organize the migrated source files between the Inc and Src folders.
4. Add the migrated source files in the project.
5. Add "legacy.c" file available under the generated $Legacy/Src folder.

3.2.5 SPL2LL-Converter migration tool limitations

Currently, the SPL2LL-Converter migration tool presents some of limitations that are dependent to the APIs implementation inside STM32 series and user's coding methods:

- The tool considers the LL APIs equivalence available in the superset devices.
- Multi-called APIs are not supported:

  Function_X (arg_0, arg_1, function_y(arg_a, arg_b), arg_2) ;
- Aliases are not supported when they are called as APIs parameters:

  #define ALIAS_0 STM32SPL_Literal_0

  Function_X(ALIAS_0) ;

  It is mandatory to call directly the defined STM32 SPL literals, otherwise the line will not be taken in consideration.

  Function_X(STM32SPL_literal_0);
- Multi literals call as parameter in not supported for some functions:
  Function_X(LITERAL_0| LITERAL1 | LITERAL2);
  it is mandatory to call only one defined literal by function call, otherwise the line will not be taken in consideration.
  Function_X(LITERAL_0);
  Function_X(LITERAL_1);
  Function_X(LITERAL_2);
- For cross-series migration, function is considered as specific once its prototype is not fully compliant between STM32 peripherals versions.

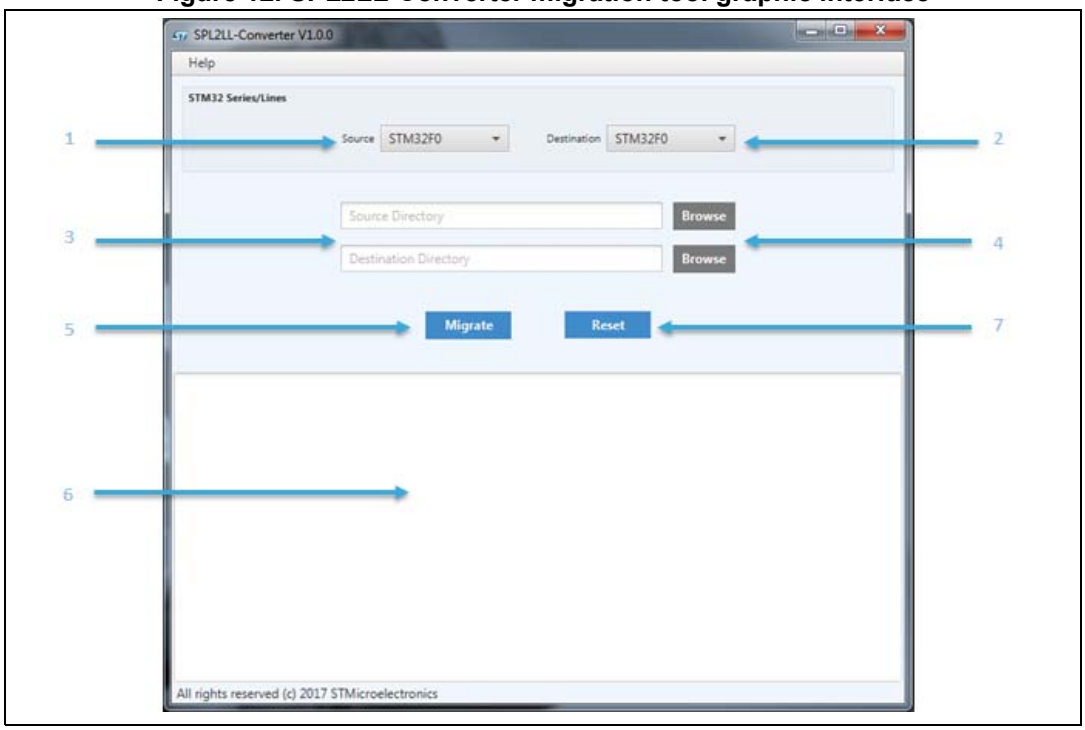
It is highly recommended to download the latest version of the tool for any potential bug fixes and behaviours enhancements.

3.2.6 GUI application

The automatic migration using the provided migration tool can be done also via a GUI application.

In the tool package, the user can find the executable for this application under the name “spl2ll_converter_gui.jar” as shown in *Figure 12*.

Figure 12. SPL2LL-Converter migration tool graphic interface



This application makes it faster and easier for users who are non-familiar with command lines to launch the migration process. It is compatible with Windows, Linux and macOS operating systems.

The interface is described as follows:

1. List of STM32 series/lines supporting STM32 SPL (source)
2. List of STM32 series/lines supporting STM32Cube LL (destination)
3. Source and Destination Projects paths
4. Browsing buttons for paths
5. Migration start button: calling on migration tool execution
6. Logging window: displaying the migration advancement and final status.
7. Reset button: re-initializing all GUI controls and fields

Before launching the application, you need to install Java RunTime Environment for 1.8.0 or later.

The user can get the latest version from the Java download web page.

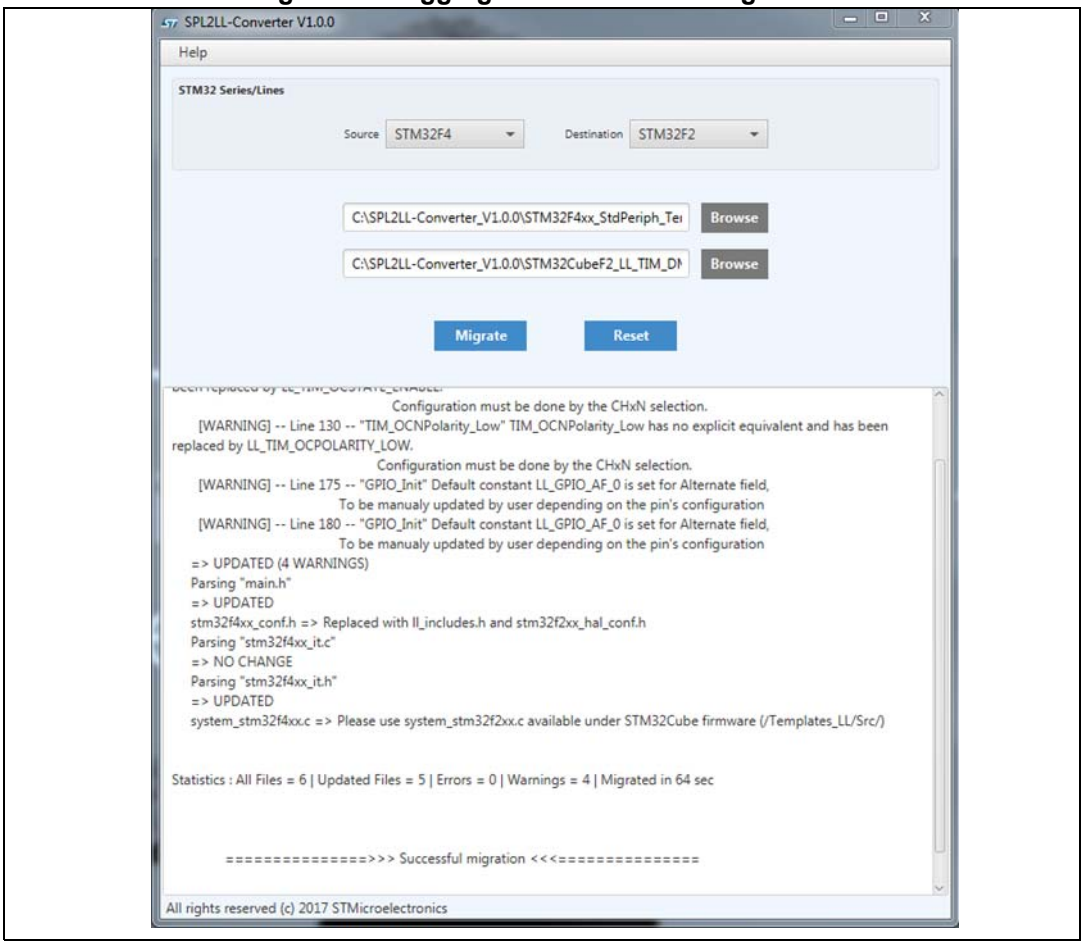
To use the application, you need to:

- Run the “spl2ll_converter_gui.jar”.
- Select the STM32 Source and destination series/lines that you need to migrate between.
- Select the source directory for your initial STM32 SPL-based application.
- Select the destination directory in which you want to find the migrated source files.
- Hit the “Migrate” button and wait for the migration process to be done.

All along the migration process, you can track the advancement on the logging window. Once finished, the overall status will be as well displayed and a log file is generated.

The figure below shows how the application looks like after a successful migration:

Figure 13. Logging after successful migration



4 Revision history

Table 7. Document revision history

Date	Revision	Changes
13-Jul-2017	1	Initial release.