# UM1754
# User manual

## Getting started with STM32CubeL0 firmware package for STM32L0 Series

## Introduction

The STMCube™ initiative was originated by STMicroelectronics to ease developers life by reducing development efforts, time and cost. STM32Cube covers the STM32 portfolio.

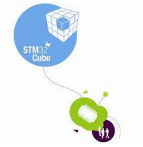
STM32Cube Version 1.x includes:

- The STM32CubeMX, a graphical software configuration tool that allows to generate C initialization code using graphical wizards
- A comprehensive embedded software platform, delivered per series (such as STM32CubeL0 for STM32L0 Series)
  - The STM32Cube HAL, an STM32 abstraction layer embedded software, ensuring maximized portability across the STM32 portfolio
  - 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, STMTouch and FatFS
  - All embedded software utilities coming with a full set of examples.

This user manual describes how to get started with the STM32CubeL0 firmware package.

*Section 1* describes the main features of STM32CubeL0 firmware, part of the STMCube™ initiative.

*Section 2* and *Section 3* provide an overview of the STM32CubeL0 architecture and firmware package structure.



# Contents


**1 STM32CubeL0 main features . . . . . 5**

**2 STM32CubeL0 architecture overview . . . . . 7**

2.1 Level 0 . . . . . 7

2.1.1 Board Support Package (BSP) . . . . . 8

2.1.2 Hardware Abstraction Layer (HAL) and Low Layer (LL) . . . . . 8

2.1.3 Basic peripheral usage examples . . . . . 9

2.2 Level 1 . . . . . 9

2.2.1 Middleware components . . . . . 9

2.2.2 Examples based on the middleware components . . . . . 10

2.3 Level 2 . . . . . 10

**3 STM32CubeL0 firmware package overview . . . . . 11**

3.1 Supported STM32L0 devices and hardware . . . . . 11

3.2 Firmware package overview . . . . . 13

**4 Getting started with STM32CubeL0 . . . . . 15**

4.1 Running custom example . . . . . 15

4.2 Developing custom application . . . . . 16

4.3 Using STM32CubeMX to generate the initialization C code . . . . . 19

4.4 Getting STM32CubeL0 release updates . . . . . 19

4.4.1 Installing and running the STM32CubeUpdater program . . . . . 19

**5 FAQs . . . . . 20**

**6 Revision history . . . . . 22**

# List of tables

Table 1. Macros for STM32L0 Series . . . . . 11
Table 2. STM32 boards for STM32L0 series . . . . . 12
Table 3. Document revision history . . . . . 22

# List of figures

Figure 1. STM32CubeL0 firmware components . . . . . 6
Figure 2. STM32CubeL0 firmware architecture . . . . . 7
Figure 3. STM32CubeL0 firmware package structure . . . . . 13
Figure 4. STM32CubeL0 example overview . . . . . 14

# 1 STM32CubeL0 main features

STM32CubeL0 gathers together, in a single package, all the generic embedded software components required to develop an application on STM32L0 microcontrollers. In line with the STMCube™ initiative, this set of components is highly portable, not only within the STM32L0 series but also to other STM32 series.

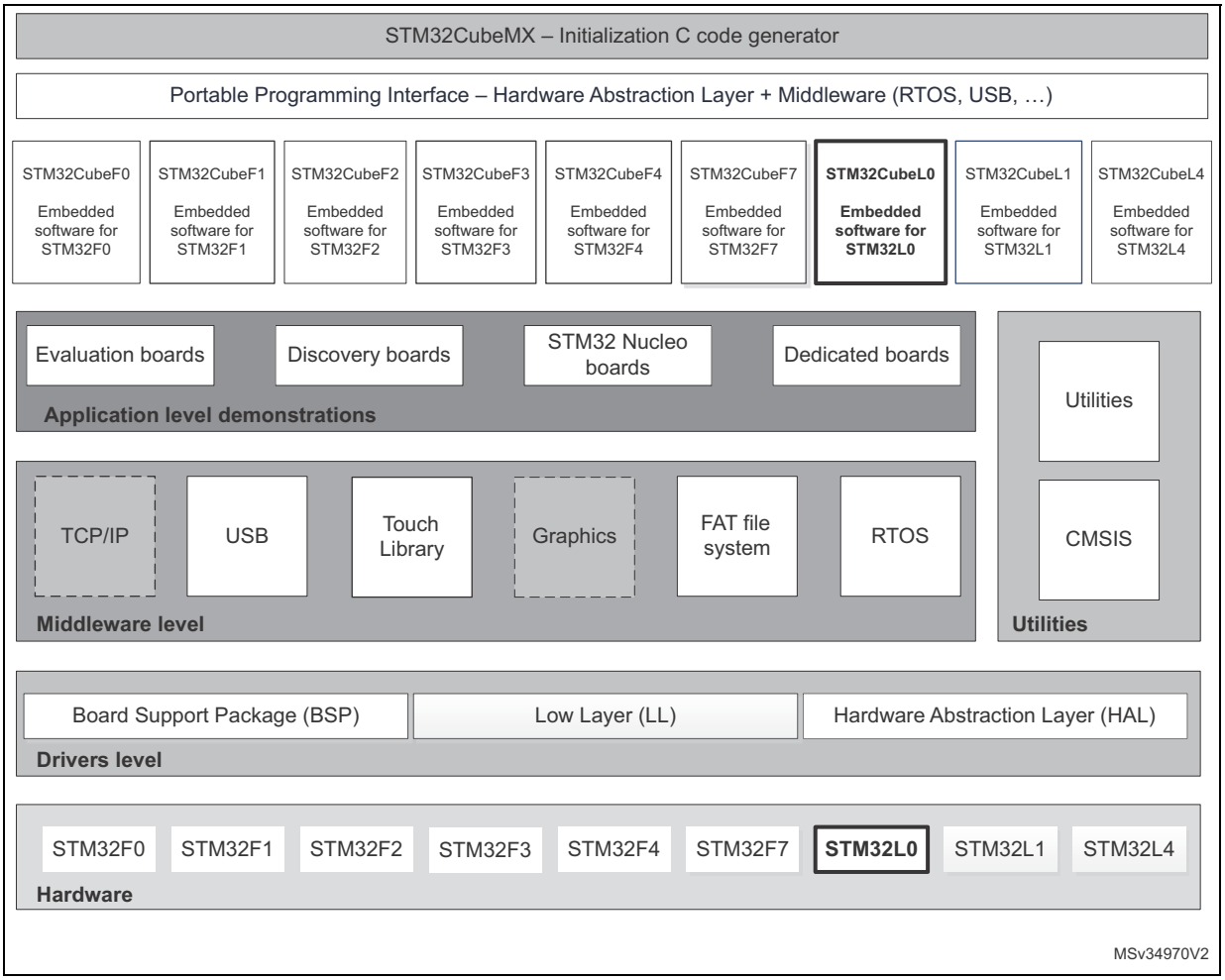
STM32CubeL0 is fully compatible with STM32CubeMX code generator that allows the user to generate initialization code. The package includes Low Layer (LL) and hardware abstraction layer (HAL) APIs that cover the microcontroller hardware, together with an extensive set of examples running on STMicroelectronics boards. The HAL and LL APIs are available in an open-source BSD license for user convenience.

STM32CubeL0 package also contains a set of middleware components with the corresponding examples. They come with very permissive license terms:

- Full USB Device stack supporting many classes
  - Device Classes: HID, MSC, CDC, DFU
- CMSIS-RTOS implementation with FreeRTOS open source solution
- FAT File system based on open source FatFS solution
- STMTouch touch sensing solutions

A demonstration implementing all these middleware components is also provided in the STM32CubeL0 package.

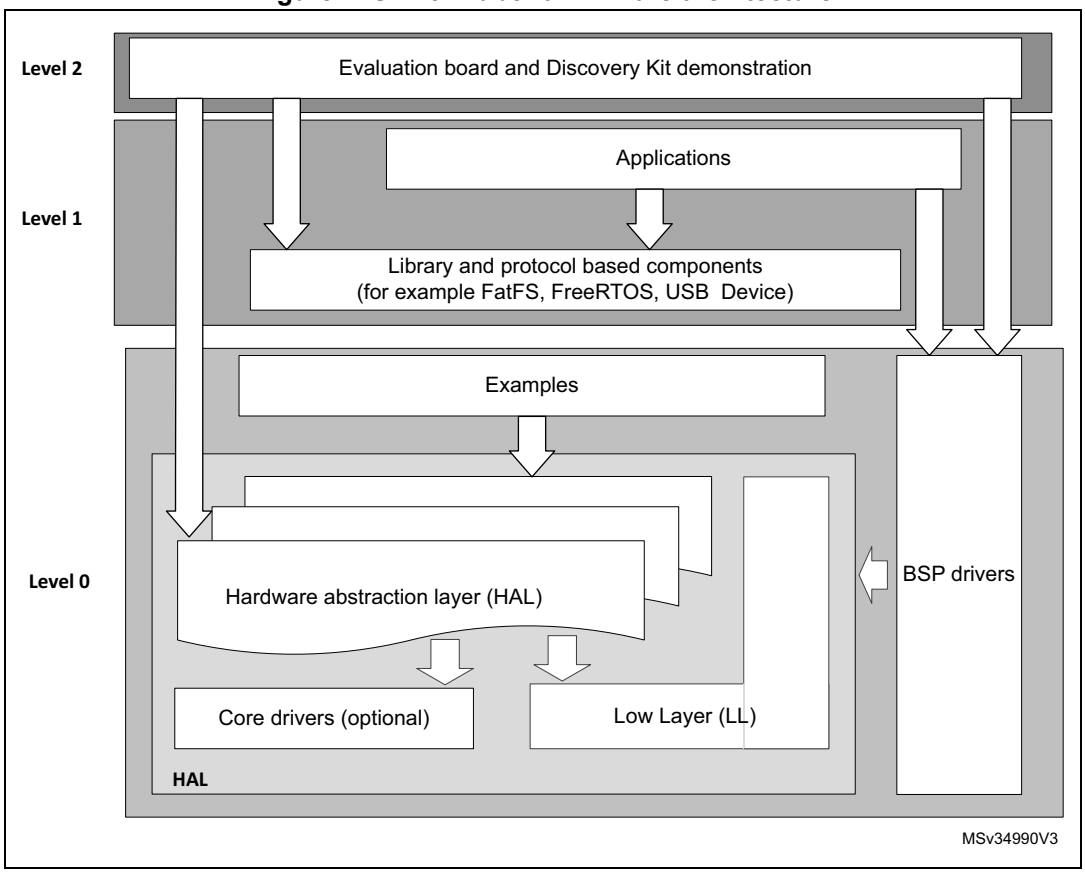
**Figure 1. STM32CubeL0 firmware components**



# 2 STM32CubeL0 architecture overview

The STM32CubeL0 firmware solution is built around three independent levels that can easily interact with each other as described in *Figure 2*:

**Figure 2. STM32CubeL0 firmware architecture**




## 2.1 Level 0

This level is divided into three sub-layers:

- Board Support package (BSP)
- Hardware Abstraction Layer (HAL)
  - HAL peripheral drivers
  - Low Layer drivers
- Basic peripheral usage examples

### 2.1.1 Board Support Package (BSP)

This layer offers a set of APIs related to the hardware components on the hardware boards (LCD drivers, Micro SD. etc…) and composed of two parts:

- Component

  This is the driver related to the external device on the board and not related to the STM32. The component driver provides specific APIs to the BSP driver's external components and can be ported to any board.

- BSP driver

  it enables the component driver to be linked to a specific board and provides a set of user-friendly APIs. The API naming rule is BSP_FUNCT_Action().

  Examples: BSP_LED_Init(), BSP_LED_On()

The BSP is based on a modular architecture that allows it to be ported easily to any hardware by just implementing the low level routines.

### 2.1.2 Hardware Abstraction Layer (HAL) and Low Layer (LL)

The STM32CubeL0 HAL and LL are complementary and cover a wide range of applications requirements:

- The HAL drivers offer high-level function-oriented highly-portable APIs. They hide the MCU and peripheral complexity to end user.

  The HAL drivers provide generic multi-instance feature-oriented APIs which simplify user application implementation by providing ready to use process. As example, for the communication peripherals (I2S, UART…), it provides APIs allowing initializing and configuring the peripheral, managing data transfer based on polling, interrupt or DMA process, and handling communication errors that may raise during communication.

  The HAL driver APIs are split in two categories:

  – Generic APIs which provides common and generic functions to all the STM32 series

  – Extension APIs which provides specific and customized functions for a specific family or a specific part number.

- The Low Layer APIs provide low-level APIs at register level, with better optimization but less portability. They require a deep knowledge of MCU and peripheral specifications. The 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 for those requiring heavy software configuration and/or complex upper-level stack (such as USB).

  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 corresponding to each field

  – Function for 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 and capability to be used in standalone mode (without HAL drivers)

  – Full coverage of the supported peripheral features.

### 2.1.3 Basic peripheral usage examples

This layer includes the examples build over the STM32 peripheral and using either the HAL or/and the Low Layer drivers APIs as well as the BSP resources.

## 2.2 Level 1

This level is divided into two sub-layers:

- Middleware components
- Examples based on the middleware components.

### 2.2.1 Middleware components

The middleware is a set of libraries covering USB Host and Device Libraries, STMTouch touch sensing, FreeRTOS and FatFS. Horizontal interactions between the components of this layer is done directly by calling the feature APIs while the vertical interaction with the low-level drivers is done through specific callbacks and static macros implemented in the library system call interface. For example, the FatFs implements the disk I/O driver to access microSD drive or the USB Mass Storage Class.

The main features of each middleware component are as follows:

- **USB Device libraries**
  - Several USB classes supported (Mass-Storage, HID, CDC, DFU, MSC).
  - Support of multi-packet transfer features that allows sending big amounts of data without splitting them into maximum packet size transfers.
  - Use of configuration files to change the core and the library configuration without changing the library code (Read Only).
  - RTOS and Standalone operation.
  - Link with low-level driver through an abstraction layer using the configuration file to avoid any dependency between the Library and the low-level drivers.
- **FreeRTOS**
  - Open source standard.
  - CMSIS compatibility layer.
  - Tickless operation during low-power mode.
  - Integration with all STM32Cube middleware modules.
- **FAT File system**
  - FATFS FAT open source library.
  - Long file name support.
  - Dynamic multi-drive support.
  - RTOS and standalone operation.
  - Examples with microSD.
- **STM32 Touch sensing library**

  Robust STMTouch capacitive touch sensing solution supporting proximity, touchkey, linear and rotary touch sensors. It is based a proven surface charge transfer acquisition principle.

### 2.2.2 Examples based on the middleware components

Each middleware component comes with one or more examples (called also Applications) showing how to use it. Integration examples that use several middleware components are provided as well.

## 2.3 Level 2

This level is composed of a single layer which consist in a global real-time and graphical demonstration based on the middleware service layer, the low-level abstraction layer and the basic peripheral usage applications for board based features.

# 3 STM32CubeL0 firmware package overview

## 3.1 Supported STM32L0 devices and hardware

STM32Cube offers a highly portable Hardware Abstraction Layer (HAL) built around a generic and modular architecture allowing the upper layers, Middleware and Application, to implement their functions without in-depth knowledge of the MCU being used. This improves the library code re-usability and guarantees an easy portability from one device to another.

The STM32CubeL0 offers full support for all STM32L0 Series devices. The user only has to define the right macro in stm32l0xx.h.

*Table 1* lists the macro to define depending on the STM32L0 device in use. Note that the macro must also be defined in the compiler preprocessor.

**Table 1. Macros for STM32L0 Series**

| Macro defined in stm32l0xx.h | STM32L0 devices |
|---|---|
| STM32L011xx | STM32L011K4, STM32L011G4, STM32011F4, STM32L011E4, STM32L011D4 |
| STM32L021xx | STM32L021K4 |
| STM32L031xx | STM32L031C6, STM32L031K6, STM32L031G6, STM32L031E6, STM32L031F6, STM32L031C4, STM32L031K4, STM32L031G4, STM32L031E4, STM32L031F4 |
| STM32L041xx | STM32L041C6, STM32L041K6, STM32L041G6, STM32L041F6, STM32L041C4 |
| STM32L051xx | STM32L051K8, STM32L051C6, STM32L051C8, STM32L051R6, STM32L051R8 |
| STM32L052xx | STM32L052K6, STM32L052K8, STM32L052C6, STM32L052C8, STM32L052R6, STM32L052R8 |
| STM32L053xx | STM32L053C6, STM32L053C8, STM32L053R6, STM32L053R8 |
| STM32L062xx | STM32L062K8 |
| STM32L063xx | STM32L063C8, STM32L063R8 |
| STM32L071xx | STM32L071V8, STM32L071K8, STM32L071VB, STM32L071RB, STM32L071CB, STM32L071KB, STM32L071VZ, STM32L071RZ, STM32L071CZ, STM32L071KZ |
| STM32L072xx | STM32L072V8, STM32L072VB, STM32L072RB, STM32L072CB, STM32L072VZ, STM32L072RZ, STM32L072CZ |
| STM32L073xx | STM32L073V8, STM32L073VB, STM32L073RB, STM32L073VZ, STM32L073RZ |
| STM32L082xx | STM32L082KBU, STM32L082KZU |
| STM32L083xx | STM32L083V8, STM32L083VB, STM32L083RB, STM32L083VZ, STM32L083RZ |

STM32CubeL0 features a rich set of examples and applications making it easy to understand and use any HAL driver and/or Middleware components. These examples can be run on any of the STMicroelectronics board as listed in *Table 2*:

**Table 2. STM32 boards for STM32L0 series**

| Board | STM32L0 devices supported |
|---|---|
| NUCLEO-L053R8 | STM32L053x8 |
| STM32L053-DISCO | STM32L053C8 |
| NUCLEO-L073RZ | STM32L073RZ |
| STM32L073Z_EVAL | STM32L073VZ |
| NUCLEO-L031K6 | STM32L031K6 |
| NUCLEO-L011K4 | STM32L011K4 |

As for all other STM32 Nucleo boards, the NUCLEO-L053R8 and NUCLEO-L073RZ feature a reduced set of hardware components (one user Key button and one user LED). To enrich the middleware support offer for STM32CubeL0 firmware package, an LCD display Adafruit Arduino shield was chosen, which embeds in addition to the LCD a µSD connector and a Joystick. The NUCLEO-L031K6 and NUCLEO-L011K4 boards are Nucleo-32 boards, a new format for 32 pins component.

The STM32CubeL0 family supports now both Nucleo-32 and Nucleo-64 boards.

- Nucleo-64 boards support Adafruit LCD display Arduino™ UNO shields which embedded microSD connector and a joystick in addition to the LCD
- Nucleo-32 boards support Gravitech 7-segment display Arduino™ NANO shield which allows displaying up to four-digit numbers and characters

The Arduino™ shield drivers are provided within the BSP component. Their usage is illustrated by a demonstration firmware.

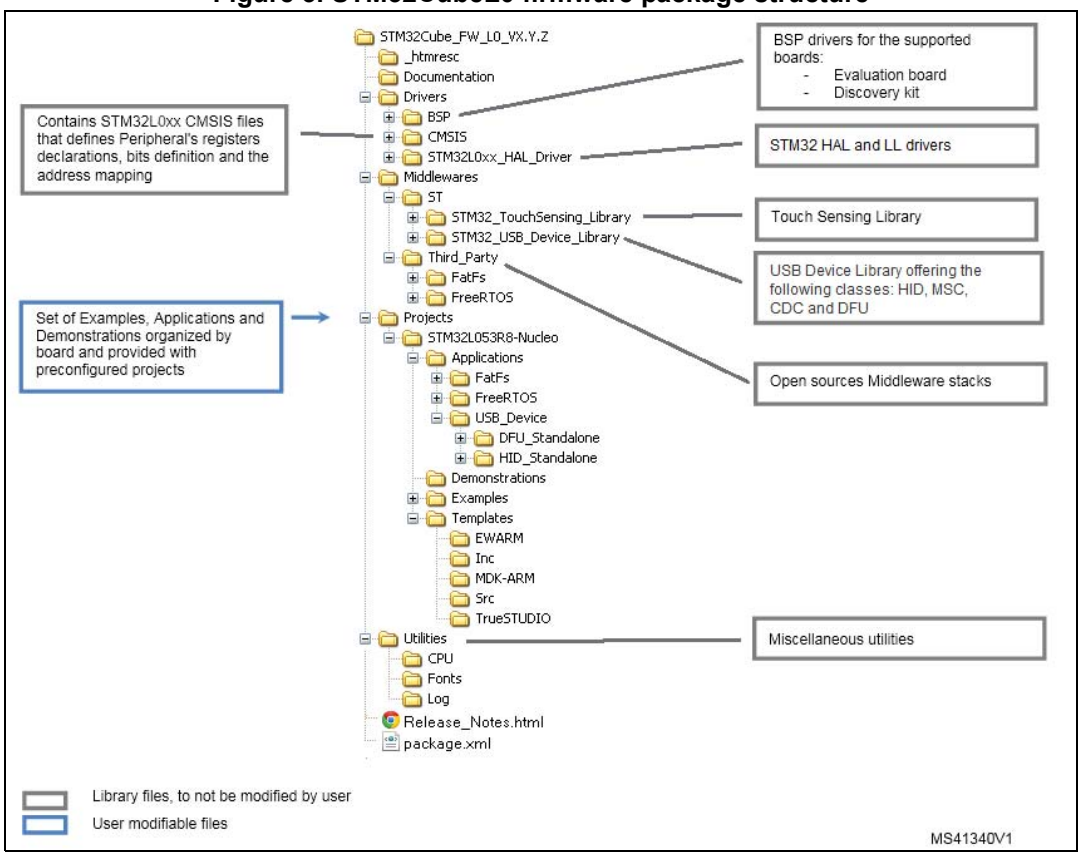
In the BSP component the dedicated drivers, for that Arduino shield are available and their use is illustrated through either the provided BSP example or in the Demonstration firmware, without forgetting the FatFS middleware application.

The STM32CubeL0 firmware is able to run on any compatible hardware. The users can simply update the BSP drivers to port the provided examples to their own board, providing it has the same hardware functions (for example LED, pushbuttons).

## 3.2 Firmware package overview

The STM32CubeL0 firmware solution is provided in a single zip package with the structure shown in *Figure 3*.

**Figure 3. STM32CubeL0 firmware package structure**




For each board, a set of examples are provided with pre-configured projects for EWARM, MDK-ARM and either TrueSTUDIO or SW4STM32 toolchains.

*Figure 4* shows the project structure for the STM32L073RZ-Nucleo_EVAL board. The structure is identical for any other additional supported board.

The examples are classified depending on the STM32Cube level they apply to, and are named as follows:

- Examples in level 0 are called Examples, Examples_LL and Examples_MIX. They use respectively HAL drivers, LL drivers and a mix of HAL and LL drivers without any middleware component
- Examples in level 1 are called Applications, that provide typical use cases of each Middleware component
- Examples in level 2 are called Demonstration, that implement all the HAL, BSP and Middleware components

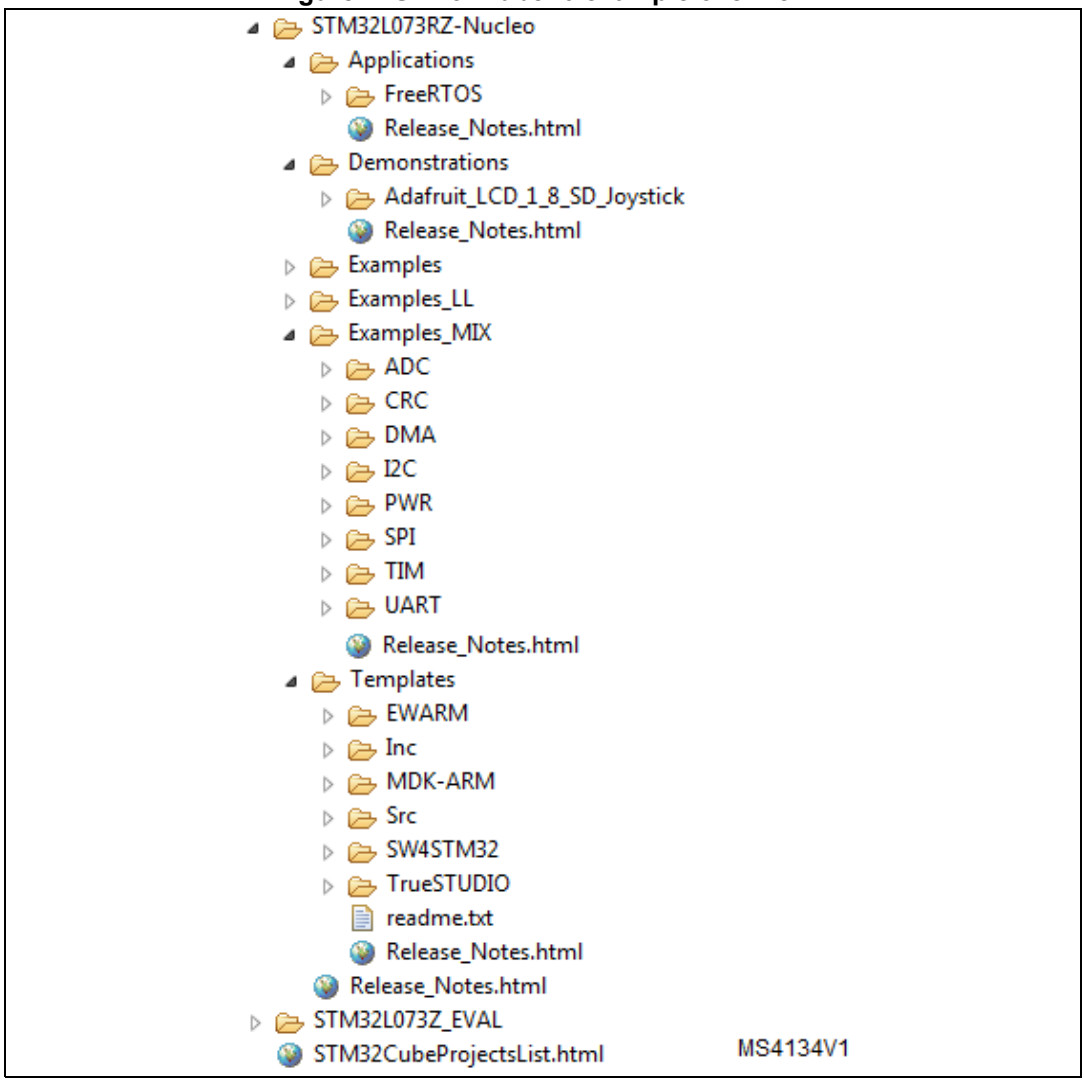
A template project is provided to users for the quick build of any firmware application on a given board.

All examples have the same structure,

- *\Inc* folder that contains all header files
- *\Src* folder for the sources code
- *\EWARM*, *\MDK-ARM* and either \SW4STM32 or *\TrueSTUDIO* folders containing the pre-configured project for each toolchain
- *readme.txt* describing the example behavior and the required environment to make it work

The list of examples provided in the STM32CubeL0 package is available in the STM32Cube_FW_L0_VX.Y.Z/Projects/STM32CubeProjectsList.html file.

**Figure 4. STM32CubeL0 example overview**



# 4 Getting started with STM32CubeL0

## 4.1 Running custom example

This section explains how to run a first example with STM32CubeL0, using as illustration the generation of a simple LED toggle running on the STM32L053R8 Nucleo board:

1. Download the STM32CubeL0 firmware package. Unzip the package into a directory. Make sure not to modify the package structure shown in *Figure 3*
2. Browse to `\Projects\STM32L053R8-Nucleo\Examples`
3. Open the *\GPIO* folder, then open the *\GPIO_EXTI* folder
4. Open the project by selecting preferred toolchain
5. Rebuild all files and load the genrerated image into the target memory
6. Run the example: each time the Key push button is pressed, the LED2 toggles (for more details, refer to the example readme file)

Below is a quick overview on how to open, build and run an example with the supported toolchains.

- EWARM
  a) Under the *example* folder, open the *\EWARM* sub folder
  b) Open the *Project.eww* workspace[a]
  c) Rebuild all files: **Project**->**Rebuild all**
  d) Load the project image: **Project**->**Debug**
  e) Run the program: **Debug**->**Go** (F5)
- MDK-ARM
  a) Under the example folder, open the *\MDK-ARM* sub folder
  b) Open the *Project.uvproj* workspace[a]
  c) Rebuild all files: **Project**->**Rebuild all target files**
  d) Load the project image: **Debug**->**Start/Stop Debug Session**
  e) Run the program: **Debug**->**Run** (F5)
- SW4STM32
  a) Open the \SW4STM32 toolchain
  b) Click **File->Switch Workspace->Other** and browse to the SW4STM32 workspace directory
  c) Click **File->Import**, select **General->Existing Projects into Workspace** and then click **Next**
  d) Browse to the SW4STM32 workspace directory and select the **project**
  e) Rebuild all project files: select the project in the "**Project explorer"** window then click the **Project->build project** menu
  f) Run program: **Run->Debug** (F11)
- TrueSTUDIO
  a) Open the TrueSTUDIO toolchain
  b) Select on **File**->**Switch Workspace**->**Other** and browse to the TrueSTUDIO workspace directory
  c) Click on **File**->**Import**, select **General**->'**Existing Projects into Workspace**' and then click "**Next**".
  d) Browse to the TrueSTUDIO workspace directory, select the **project**
  e) Rebuild all project files: Select the project in the "**Project explorer**" window then click on **Project**->**build project** menu
  f) Run the program: **Run**->**Debug** (F11)

## 4.2 Developing custom application

This section describes the successive steps to create a custom application using STM32CubeL0.

1. **Creating project:** to create a new project, it's possible to start from the **Template** project provided for each board under `\Projects\<STM32xx_xxx>\Templates` or

a. The workspace name may change from one example to another

from any available project under `\Projects\<STM32xx_xxx>\Examples` or `\Projects\STM32xx_xxx\Applications` (<STM32xx_xxx> refers to the board name, for example STM32L053R8).

The Template project provides an empty main loop function. It is a good starting point to get familiar with the project settings for STM32CubeL0. It has the following characteristics:

a) It contains sources of the HAL, CMSIS and BSP drivers which are the minimum required components to develop code for a given board.

b) It contains the include paths for all the firmware components.

c) It defines the STM32L0 device supported, allowing to set the configuration for the CMSIS and HAL drivers respectively.

d) It provides ready-to-use user files pre-configured as follows:

- HAL is initialized

- SysTick ISR implemented for HAL_Delay() purpose

- System clock is configured with the minimum frequency of the device (MSI) and though for an optimum power consumption.

*Note: When an existing project is copied from a location to another location make sure to update the include paths.*

2. **Adding the necessary Middleware to the project (optional):** the available Middleware stacks are: USB Device Libraries, STMTouch touch library, FreeRTOS and FatFS. To find out which source files are needed to add to the project files list, refer to the documentation provided for each Middleware. It is possible also to look at the applications available under `\Projects\STM32xx_xxx\Applications\<MW_Stack>` (<MW_Stack> refers to the Middleware stack, for example USB_Device) to get a better idea of the source files to be added and the include paths.

3. **Configuring the firmware components:** the HAL and Middleware components offer a set of build time configuration options using macros declared with “#define” in a header file. A template configuration file is provided within each component, which has to be copied into to the project folder (usually the configuration file is named *xxx_conf_template.h*. Make sure to remove the word “_template” when copying the file to the project folder). The configuration file provides enough information to know the effect of each configuration option. More detailed information is available in the documentation provided for each component.

4. **Starting the HAL Library:** after jumping to the main program, the application code needs to call *HAL_Init()* API to initialize the HAL Library and do the following:

   a) Configure the Flash prefetch, instruction and data caches (user-configurable by macros defined in *stm32l0xx_hal_conf.h*)

   b) Configure the Systick to generate an interrupt every 1 msec, which is clocked by the MSI, this the default configuration after reset (at this stage, the clock is not yet configured and thus the system is running from the internal 4 MHz MSI).

   c) Call the *HAL_MspInit()* callback function defined in the user file *stm32l0xx_hal_msp.c* to do the global low level hardware initialization

5. **Configuring the system clock:** the system clock configuration is set by calling the two following APIs

   a) *HAL_RCC_OscConfig()*: configures the internal and/or external oscillators, PLL source and factors. The user can choose to configure one oscillator or all

oscillators. The user can also skip the PLL configuration if there is no need to run the system at high frequency

b) *HAL_RCC_ClockConfig()*: configures the system clock source, Flash latency and AHB and APB prescalers

*Note: Prior to configuring the system clock, it is recommended to enable the power controller clock, and to configure the appropriate voltage scaling, and therefore to optimize the power consumption when the system is clocked below the maximum allowed frequency.*

**6. Peripheral initialization**

a) Start by writing the peripheral HAL_PPP_MspInit function. For this function, please proceed as follows:

i. Enable the peripheral clock

ii. Configure the peripheral GPIOs

iii. Configure DMA channel and enable DMA interrupt (if needed).

iv. Enable peripheral interrupt (if needed)

b) Edit the stm32xxx_it.c to call required interrupt handlers (peripheral and DMA), if needed

c) Write process complete callback functions if peripheral interrupt or DMA has to be used

d) In the main.c file, initialize the peripheral handle structure then call the function HAL_PPP_Init() to initialize the peripheral

**7. Develop custom application**

At this stage, the system is ready and development of an application code can started.

a) The HAL provides intuitive and ready-to-use APIs to configure the peripheral, and supports polling, interrupt and DMA programming models, to accommodate any application requirements. For more details on how to use each peripheral, refer to the extensive set of examples provided.

b) If the application has some real-time constraints, a large set of examples are available showing how to use FreeRTOS and how integrate it with all Middleware stacks provided in STM32CubeL0. This can be a good starting point fordevelopment.

**Caution:** In the default HAL implementation, Systick timer is the source of time base. It is used to generate interrupts at regular time intervals. Take care if *HAL_Delay()* is called from peripheral ISR process. The SysTick interrupt must have higher priority (numerically lower) than the peripheral interrupt. Otherwise, the caller ISR process is blocked.
Functions affecting time base configurations are declared as __Weak to make override possible in case of other implementations in user file (using a general purpose timer for example or other time source), for more details please refer to HAL_TimeBase example.

## 4.3 Using STM32CubeMX to generate the initialization C code

An alternative to steps 1 to 6 described in *Section 4.2* consists in using the STM32CubeMX tool to generate code for the initialization of the system, the peripherals and middleware (Steps 1 to 5 above) through a step-by-step process:

- Select the STMicroelectronics STM32 microcontroller that matches the required set of peripherals.
- Configure each required embedded software thanks to a pinout-conflict solver, a clock-tree setting helper, a power consumption calculator, and the utility performing MCU peripheral configuration (for example GPIO, USART) and middleware stacks (for example USB).
- Generate the initialization C code based on the configuration selected. This code is ready to use within several development environments. The user code is kept at the next code generation.

For more information, please refer to UM1718.

## 4.4 Getting STM32CubeL0 release updates

The STM32CubeL0 firmware package comes with an updater utility: STM32CubeUpdater, also available as a menu within STM32CubeMX code generation tool.

The updater solution detects new firmware releases and patches available from *www.st.com* and proposes to download them to the user's computer.

### 4.4.1 Installing and running the STM32CubeUpdater program

- Double-click on the *SetupSTM32CubeUpdater.exe* file to launch the installation.
- Accept the license terms and follow the different installation steps.

Upon successful installation, STM32CubeUpdater becomes available as an STMicroelectronics program under Program Files and is automatically launched.

The STM32CubeUpdater icon appears in the system tray:

- Right-click the updater icon and select **Updater Settings** to configure the Updater connection and whether to perform manual or automatic checks (see STM32CubeMX User guide - UM1718 section 3 - for more details on Updater configuration).

# 5 FAQs

### What is the license scheme for the STM32CubeL0 firmware?

The HAL is distributed under a non-restrictive BSD (Berkeley Software Distribution) license.

The Middleware stacks made by ST (USB Device Libraries, STMTouch touch library) come with a licensing model allowing easy reuse, provided it runs on an ST device.

The Middleware based on well-known open-source solutions (FreeRTOS and FatFs) have user-friendly license terms. For more details, refer to the license agreement of each Middleware.

### What boards are supported by the STM32CubeL0 firmware package?

The STM32CubeL0 firmware package provides BSP drivers and ready-to-use examples for the NUCLEO-L053R8, NUCLEO-L073RZ, NUCLEO-L011K4, NUCLEO-L031K6, STM32L073Z_EVAL and STM32L053-DISCO boards.

For the up-to-date list of supported boards, please refer to the firmware package release notes.

### Are any examples provided with the ready-to-use toolset projects?

Yes. STM32CubeL0 provides a rich set of examples and applications. They come with the pre-configured projects of several toolsets: IAR, Keil and GCC.

### Is there any link with Standard Peripheral Libraries?

The STM32Cube HAL and LL drivers are the replacement of the standard peripheral library:

- The HAL drivers offer a higher abstraction level compared to the standard peripheral APIs. They focus on peripheral common features rather than hardware. Their higher abstraction level allows defining a set of user-friendly APIs that can be easily ported from one product to another.
- The LL drivers offer low-level APIs at registers level. They are organized in a simpler and clearer way than direct register accesses. LL drivers also include peripheral initialization APIs, which are more optimized compared to what is offered by the SPL, while being functionally similar. Compared to HAL drivers, these LL initialization APIs allows an easier migration from the SPL to the STM32Cube LL drivers, since each SPL API has its equivalent LL API(s).

### Does the HAL take benefit from interrupts or DMA? How can this be controlled?

Yes. The HAL supports three API programming models: polling, interrupt and DMA (with or without interrupt generation).

### How are the product/peripheral specific features managed?

The HAL offers extended APIs, i.e. specific functions as add-ons to the common API to support features available on some products/lines only.

### How can STM32CubeMX generate code based on embedded software?

STM32CubeMX has a built-in knowledge of STM32 microcontrollers, including their peripherals and software. This enables the tool to provide a graphical representation to the user and generate *.h/*.c files based on user configuration.

### How to get regular updates on the latest STM32CubeL0 firmware releases?

The STM32CubeL0 firmware package comes with an updater utility, STM32CubeUpdater, that can be configured for automatic or on-demand checks for new firmware package updates (new releases or/and patches).

STM32CubeUpdater is integrated as well within the STM32CubeMX tool. When using this tool for STM32L0 configuration and initialization C code generation, the user can benefit from STM32CubeMX self-updates as well as STM32CubeL0 firmware package updates.

For more details, refer to *Section 4.4*.

### When should I use HAL versus LL drivers?

HAL drivers offer high-level and function-oriented APIs, with a high level of portability. Product/IPs complexity is hidden for end users.

LL drivers offer low-level APIs at registers level, with a better optimization but less portability. They require a deep knowledge of product/IPs specifications.

### How can I include LL drivers in my environment? Is there any LL configuration file as for HAL?

There is no configuration file. Source code shall directly include the necessary stm32l0xx_ll_ppp.h file(s).

### Can I use HAL and LL drivers together? If yes, what are the constraints?

It is possible to use both HAL and LL drivers. One can handle the IP initialization phase with HAL and then manage the I/O operations with LL drivers.

The major difference between HAL and LL is that HAL drivers require to create and use handles for operation management while LL drivers operates directly on peripheral registers. Mixing HAL and LL is illustrated in Examples_MIX example.

### Is there any LL APIs which are not available with HAL?

Yes, there are.

A few Cortex® APIs have been added in stm32l0xx_ll_cortex.h e.g. for accessing SCB or SysTick registers.

### Why are SysTick interrupts not enabled on LL drivers?

When using LL drivers in standalone mode, SysTick interrupts has not to be enabled because they are not used in LL APIs, while HAL functions requires SysTick interrupts to manage timeouts.

### How are LL initialization APIs enabled?

The definition of LL initialization APIs and associated resources (structure, literals and prototypes) is conditioned by the USE_FULL_LL_DRIVER compilation switch.

# 6 Revision history

Table 3. Document revision history

| Date | Revision | Changes |
|---|---|---|
| 24-Apr-2014 | 1 | Initial release. |
| 27-Feb-2015 | 2 | Extended the applicability to STM32L073Z-EVAL and NUCLEO-L073RZ boards for STM32L073xx and STM32L083xx device families, respectively.<br>Updated:<br>– *Table 1: Macros for STM32L0 Series*,<br>– *Table 2: STM32 boards for STM32L0 series*<br>– the list of available STM32 Nucleo boards in *Section 3.1: Supported STM32L0 devices and hardware*,<br>– *Figure 3: STM32CubeL0 firmware package structure*,<br>– the sentence introducing *Figure 4*,<br>– *Figure 4: STM32CubeL0 example overview*,<br>– the sentence introducing *Table 3*. |
| 13-Nov-2015 | 3 | Introduction of the STM32L031xx and STM32L011xx devices. Addition of new boards: NUCLEO-L031K6, NUCLEO-L011K4.<br>Updated:<br>– *Table 1: Macros for STM32L0 Series*<br>– *Table 2: STM32 boards for STM32L0 series*<br>– *Figure 3: STM32CubeL0 firmware package structure* |
| 04-May-2016 | 4 | Added Low Layer (LL) drivers<br>– *Figure 1: STM32CubeL0 firmware components*<br>– *Figure 2: STM32CubeL0 firmware architecture*<br>– *Figure 3: STM32CubeL0 firmware package structure*<br>– *Figure 4: STM32CubeL0 example overview*<br>Removed table of available examples, applications and demonstrations<br>FAQ updated |