UM3302

User manual

Getting started with STM32CubeU0 for STM32U0 series

Introduction

STM32Cube is an STMicroelectronics original initiative to improve designer productivity significantly by reducing development effort, time, and cost. STM32Cube covers the whole STM32 portfolio.

STM32Cube includes:

- A set of user-friendly software development tools to cover project development from conception to realization, among which are:
  - STM32CubeMX, a graphical software configuration tool that allows the automatic generation of C initialization code using graphical wizards
  - STM32CubeIDE, an all-in-one development tool with peripheral configuration, code generation, code compilation, and debug features
  - STM32CubeCLT, an all-in-one command-line development toolset with code compilation, board programming, and debug features
  - STM32CubeProgrammer (STM32CubeProg), a programming tool available in graphical and command-line versions
  - STM32CubeMonitor (STM32CubeMonitor, STM32CubeMonPwr, STM32CubeMonRF, STM32CubeMonUCPD), powerful monitoring tools to fine-tune the behavior and performance of STM32 applications in real time
- STM32Cube MCU and MPU Packages, comprehensive embedded-software platforms specific to each microcontroller and microprocessor series (such as STM32CubeU0 for the STM32U0 series), which include:
  - STM32Cube hardware abstraction layer (HAL), ensuring maximized portability across the STM32 portfolio
  - STM32Cube low-layer APIs, ensuring the best performance and footprints with a high degree of user control over hardware
  - A consistent set of middleware components such as ThreadX, FileX / LevelX, USBX, touch library, mbed-crypto, MCUboot, and OpenBL
  - All embedded software utilities with full sets of peripheral and applicative examples
- STM32Cube Expansion Packages, which contain embedded software components that complement the functionalities of the STM32Cube MCU and MPU Packages with:
  - Middleware extensions and applicative layers
  - Examples running on some specific STMicroelectronics development boards

This user manual describes how to get started with the STM32CubeU0 MCU Package.

Section 2 describes the main features of the STM32CubeU0 MCU Package.

Section 3 and Section 4 provide an overview of the STM32CubeU0 architecture and MCU Package structure.

1 General information

The STM32CubeU0 application runs on the STM32U0 series 32-bit microcontrollers based on the Arm® Cortex®-M processor.

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

2 STM32CubeU0 main features

STM32CubeU0 gathers, in a single package, all the generic embedded software components required to develop an application for the STM32U0 series microcontrollers. In line with the STM32Cube initiative, this set of components is highly portable, not only within the STM32U0 series microcontrollers but also to other STM32 series.

STM32CubeU0 is fully compatible with the STM32CubeMX code generator for generating 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 open-source BSD license for user convenience.

The STM32CubeU0 MCU Package also contains a comprehensive middleware component constructed around Microsoft® Azure® RTOS middleware and other in-house and open source stacks, with the corresponding examples.

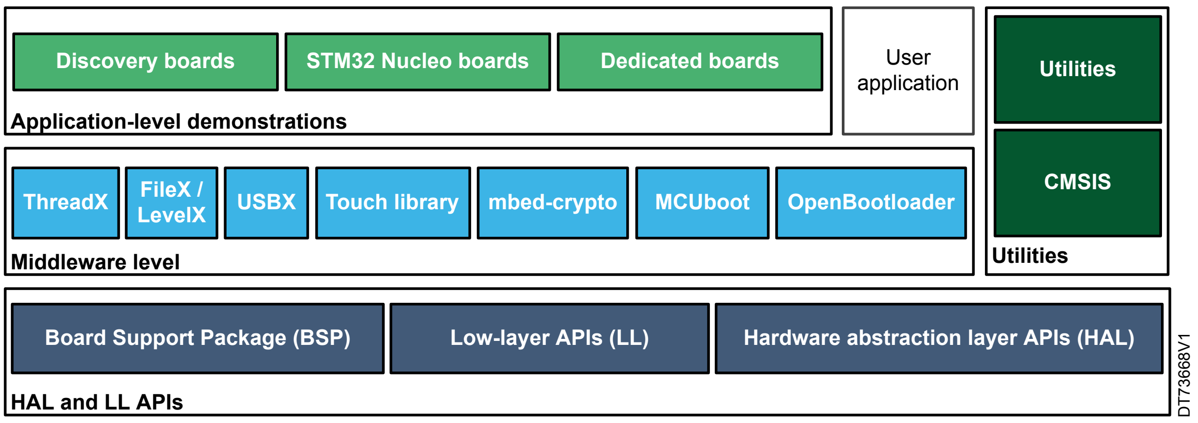
They come with free user-friendly license terms:

- Integrated and full featured RTOS: ThreadX
- CMSIS-RTOS implementation with FreeRTOS™ open-source solution
- CMSIS-RTOS implementation with ThreadX
- USB Device stacks coming with many classes: USBX
- Advanced file system and flash memory translation layer: FileX/LevelX
- OpenBootloader (OpenBL)
- MCUboot
- mbed-crypto libraries
- STM32_Touch library

Several applications and demonstrations implementing all these middleware components are also provided in the STM32CubeU0 MCU Package.

Figure 1 illustrates the STM32CubeU0 MCU Package component layout.

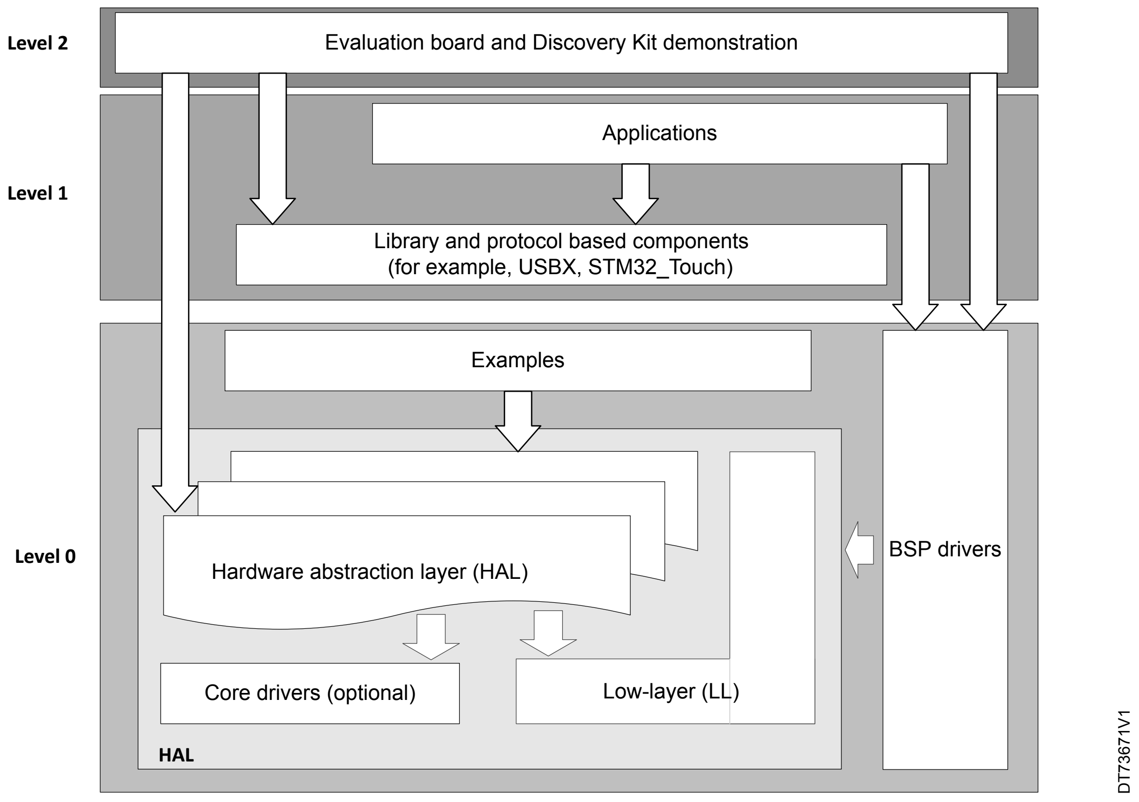
Figure 1. **STM32CubeU0 MCU Package components**



3 STM32CubeU0 architecture overview

The STM32CubeU0 MCU Package solution is built around three independent levels that easily interact, as described in Figure 2.

Figure 2. STM32CubeU0 MCU Package architecture


Level 2
Evaluation board and Discovery Kit demonstration
Applications
Level 1
Library and protocol based components
(for example, USBX, STM32_Touch)
Examples
Level 0
Hardware abstraction layer (HAL)
BSP drivers
Core drivers (optional)
Low-layer (LL)
HAL
DT73671V1

3.1 Level 0

This level is divided into three sublayers:

- Board support package (BSP)
- Hardware abstraction layer (HAL)
  - HAL peripheral drivers
  - Low-layer drivers
- Basic peripheral usage examples

3.1.1 Board support package (BSP)

This layer offers a set of APIs relative to the hardware components in the hardware boards (such as LCD, joystick, and temperature sensor). It is composed of two parts:

- Component
  This is the driver relative to the external device on the board and not the STM32 device. The component driver provides specific APIs to the BSP driver external components and cab be portable onto any other board.
- BSP driver
  It allows linking the component drivers to a specific board and provides a set of user-friendly APIs. The API naming rule is `BSP_FUNCT_Action()`.
  Example: `BSP_LED_Init()`, `BSP_LED_On()`

The BSP is based on a modular architecture allowing for easy porting on any type of hardware by implementing the low-level routines.

3.1.2 Hardware abstraction layer (HAL) and low-layer (LL)

The STM32CubeU0 HAL and LL are complementary and cover a wide range of application requirements:

- The HAL drivers offer high-level function-oriented highly portable APIs. They hide the MCU and peripheral complexity from the end user.
  The HAL drivers provide generic multi-instance feature-oriented APIs that simplify user application implementation by providing ready-to-use processes. For example, for the communication peripherals (I2S, UART, and others), it provides APIs allowing the initialization and configuration of the peripheral, managing data transfer based on polling, interrupt, or DMA process, and handling communication errors that may arise during communication. The HAL driver APIs are split in two categories:
  – Generic APIs that provide common and generic functions to all STM32 series.
  – Extension APIs that provide 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 the MCU and peripheral specifications.
  The LL drivers are designed to offer a fast light-weight expert-oriented layer that 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 a complex upper-level stack.
  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.
  – A 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 the HAL and capability to be used in standalone mode (without HAL drivers).
  – Full coverage of the supported peripheral features.

3.1.3 Basic peripheral usage examples

This layer encloses the examples built over the STM32 peripherals using only the HAL and BSP resources.

3.2 Level 1

This level is divided into two sublayers:

- Middleware components
- Examples based on the middleware components

3.2.1 Middleware components

The middleware is a set of libraries constructed around Microsoft® Azure® RTOS middleware and other in-house (such as OpenBL) and open-source (such as mbed-crypto) libraries. All are integrated and customized for STM32 MCU devices and enriched with corresponding application examples based on promotional boards. Horizontal interactions between the components of this layer are done by calling the feature APIs while the vertical interaction with the low-layer drivers is done through specific callbacks and static macros implemented in the library system call interface.

The main features of each middleware component are as follows:

- ThreadX:
  A real-time operating system (RTOS), designed for embedded systems, with two functional modes:
  - Common mode: common RTOS functionalities, such as thread management and synchronization, memory pool management, messaging, and event handling.
  - Module mode: an advanced usage mode that allows the on-the-fly loading and unloading of prelinked ThreadX modules through a module manager.
- FileX / LevelX:
  - Advanced flash file system (FS) / flash translation layer (FTL): fully featured to support NAND/NOR flash memories.
- USBX:
  - USB Device stacks coming with many classes (USB Type-C®).
- OpenBootloader:
  This middleware component provides an open-source bootloader with exactly the same features and tools as the STM32 system bootloader.
- STM32 Touch Sensing Library:
  A software library used to support touch sensors with the Touch Sensing Controller peripheral (TSC).
- MCUboot.
- mbed-crypto:
  An open-source cryptography library that supports a wide range of cryptographic operations, including:
  - Key management.
  - Hashing.
  - Symmetric cryptography.
  - Asymmetric cryptography.
  - Message authentication (MAC).
  - Key generation and derivation.
  - Authenticated encryption with associated data (AEAD).

3.2.2 Examples based on the middleware components

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

3.3 Level 2

This level is composed of a single layer that consists of 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.

4 STM32CubeU0 MCU Package overview

4.1 Supported STM32CubeU0 series devices and hardware

STM32Cube offers a highly portable hardware abstraction layer (HAL) built around a generic architecture. It allows the build-upon-layers principle, such as using the middleware layer to implement their functions without knowing what MCU is used. This improves the reusability of the library code and ensures easy portability to other devices.

In addition, owing to its layered architecture, STM32CubeU0 offers full support of all STM32U0 series devices. The user only has to define the right macro in `stm32u0xx.h`.

Table 1 shows which macro to define, depending on the STM32U0 series device used. This macro must also be defined in the compiler preprocessor.

Table 1. Macros for STM32CubeU0

Macro defined in `stm32u0xx.h`	STM32U0 devices
STM32U031x4	STM32U031F4, STM32U031K4
STM32U031x6	STM32U031F6, STM32U031K6, STM32U031C6, STM32U031R6, STM32U031G6
STM32U031x8	STM32U031F8, STM32U031K8, STM32U031C8, STM32U031R8, STM32U031G8
STM32U073x8	STM32U073K8, STM32U073H8, STM32U073C8, STM32U073R8, STM32U073M8
STM32U073xB	STM32U073KB, STM32U073HB, STM32U073CB, STM32U073RB, STM32U073MB
STM32U073xC	STM32U073KC, STM32U073HC, STM32U073CC, STM32U073RC, STM32U073MC
STM32U083xC	STM32U083KC, STM32U083HC, STM32U083CC, STM32U083RC, STM32U083MC

STM32CubeU0 features a rich set of examples and applications at all levels, making it easy to understand and use any HAL driver and/or middleware components. These examples run on the STMicroelectronics boards listed in Table 2.

Table 2. Boards for STM32CubeU0 series

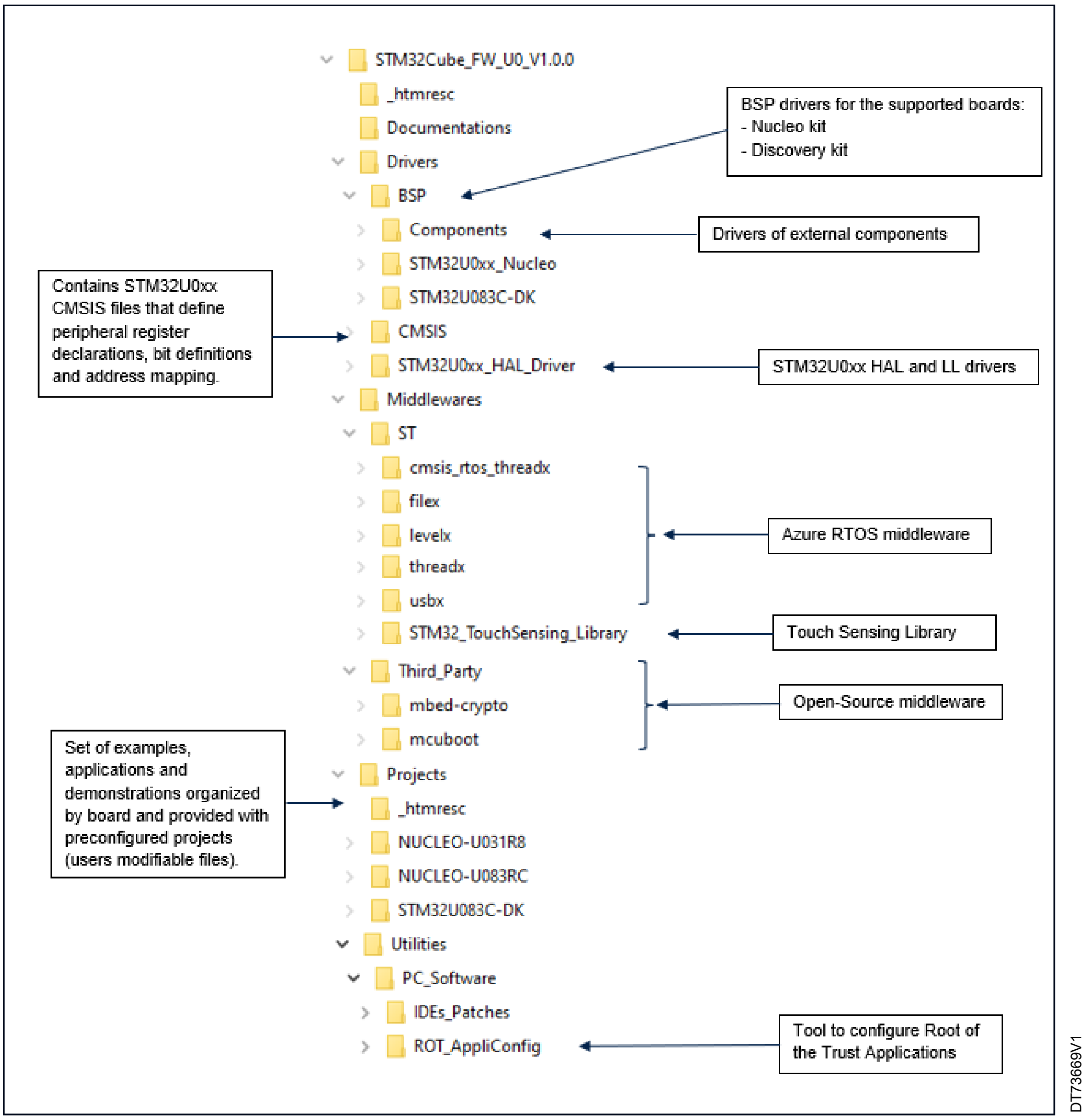
Board	Supported STM32CubeU0 devices
NUCLEO-U031R8	STM32U031xx
NUCLEO-U083RC	STM32U073xx, STM32U083xx
STM32U083C-DK	STM332U073xx, STM32U083xx

The STM32CubeU0 MCU Package is able to run on all compatible hardware. The user updates the BSP drivers to port the provided examples onto their own board, if the latter has the same hardware features (such as LED, LCD, and buttons).

4.2 MCU Package overview

The STM32CubeU0 MCU Package solution is provided in one single zip package, with the structure shown in Figure 3.

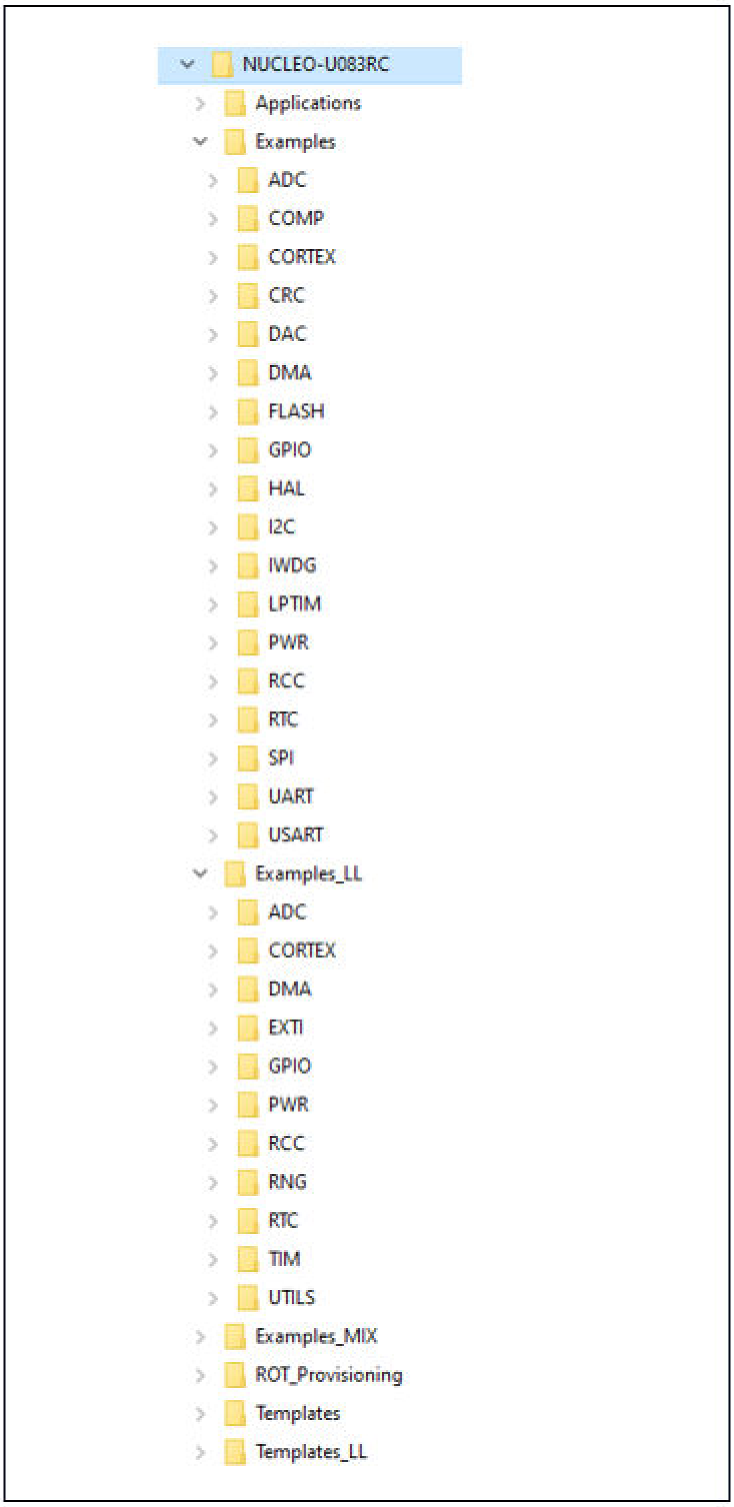
Figure 3. STM32CubeU0 MCU Package structure


STM32Cube_FW_U0_V1.0.0
_htmresc
Documentations
Drivers
BSP
Components
STM32U0xx_Nucleo
STM32U083C-DK
CMSIS
STM32U0xx_HAL_Driver
Middlewares
ST
cmsis_rtos_threadx
filex
levelx
threadx
usbx
STM32_TouchSensing_Library
Third_Party
mbed-crypto
mcuboot
Projects
_htmresc
NUCLEO-U031R8
NUCLEO-U083RC
STM32U083C-DK
Utilities
PC_Software
IDEs_Patches
ROT_AppliConfig
BSP drivers for the supported boards:
- Nucleo kit
- Discovery kit
Drivers of external components
Contains STM32U0xx CMSIS files that define peripheral register declarations, bit definitions and address mapping.
STM32U0xx HAL and LL drivers
Azure RTOS middleware
Touch Sensing Library
Open-Source middleware
Set of examples, applications and demonstrations organized by board and provided with preconfigured projects (users modifiable files).
Tool to configure Root of the Trust Applications
DT73669V1

For each board, a set of examples is provided with preconfigured projects for EWARM, MDK-ARM, and STM32CubeIDE toolchains.

Figure 4 shows the project structure for the STM32U0xx_Nucleo board.

Figure 4. STM32CubeU0 examples overview



```
NUCLEO-U083RC
  Applications
  Examples
    ADC
    COMP
    CORTEX
    CRC
    DAC
    DMA
    FLASH
    GPIO
    HAL
    I2C
    IWDG
    LPTIM
    PWR
    RCC
    RTC
    SPI
    UART
    USART
  Examples_LL
    ADC
    CORTEX
    DMA
    EXTI
    GPIO
    PWR
    RCC
    RNG
    RTC
    TIM
    UTILS
  Examples_MIX
  ROT_Provisioning
  Templates
  Templates_LL
```

DT73670V1

The examples are classified according to the STM32Cube level they apply to, and are named as explained below:

- Level 0 examples 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 components.
- Level 1 examples are called applications. They provide typical use cases of each middleware component.

Any firmware application for a given board can be built quickly using the template projects available in the `Templates` and `Templates_LL` directories.

All examples have the same structure:

- A `\Inc` folder, containing all header files.
- A `\Src` folder for the source code.
- `\EWARM`, `\MDK-ARM`, and `\STM32CubeIDE` folders, containing the pre-configured project for each toolchain.
- A `readme.txt` file describing the example behavior and environment requirements to make it work.
- An `*.ioc` file, allowing users to open most of the firmware examples within STM32CubeMX (starting from STM32CubeMX 6.11).

Table 3. **Number of examples for each board**

Level	NUCLEO-U031R8	NUCLEO-U083RC	STM32U083C-DK	Total
Applications	2	8	9	19
Demonstration	0	0	1	1
Examples	28	104	30	162
Examples_LL	3	78	1	82
Examples_MIX	0	14	0	14
Templates	1	1	1	3
Templates_LL	1	1	1	3
Total projects	35	206	43	284

As shown in Table 3 above, the STM32CubeU0 package contains 284 examples dispatched on 3 boards, 193 of which are unique examples.

5 Getting started with STM32CubeU0

5.1 Running a first example

This section explains how to run a first example on an STM32U0 series board, toggling an LED on the NUCLEO-U083RC board.

1. Download the STM32CubeU0 MCU Package and unzip it into a separate directory without modifying the package structure as shown in Figure 3.

Note: copy the package as close as possible to the root volume (for example `C:\Eval` or `G:\Tests`) because some IDEs can encounter problems when the path length is too long.

2. Browse to `\Projects\NUCLEO-U083RC\Examples`.
3. Open `\GPIO`, then the `\GPIO_EXTI` folders.
4. Open the project with a preferred toolchain. A quick overview on how to open, build, and run an example with the supported toolchains is given below.
5. Rebuild all files and load the image into the target memory.
6. Run the example: each time the USER pushbutton is pressed, the LED1 toggles (for more details, refer to the example `readme` file).

To open, build and run an example with the supported toolchains, follow the steps below.

- EWARM:
  1. Open the `\EWARM` subfolder in the example folder.
  2. Launch the `Project.eww` workspace.

  Note: The workspace name may differ from one example to another.

  3. Rebuild all files: [**Project**]>[**Rebuild all**].
  4. Load project image: [**Project**]>[**Debug**].
  5. Run program: [**Debug**]>[**Go (F5)**].
- MDK-ARM:
  1. Open the `\MDK-ARM` subfolder in the example folder.
  2. Launch the `Project.uvprojx` workspace.

  Note: The workspace name may differ from one example to another.

  3. Rebuild all files: [**Project**]>[**Rebuild all target files**].
  4. Load project image: [**Project**]>[**Start/Stop Debug Session**].
  5. Run program: [**Debug**]>[**Run (F5)**].
- STM32CubeIDE
  1. Open the STM32CubeIDE toolchain.
  2. Click [**File**]>[**Switch Workspace**]>[**Other**] and browse to the STM32CubeIDE workspace directory.
  3. Click [**File**]>[**Import**], select [**General**]>[**Existing Projects Into Workspace**] and click [**Next**].
  4. Browse to the STM32CubeIDE workspace directory and select the project.
  5. Rebuild all project files: select the project in the Project explorer window, then click the [**Project**]>[**Build project**] menu.
  6. Run program: [**Run**]>[**Debug (F11)**].

5.2 Developing a custom application

5.2.1 Using STM32CubeMX to develop or update an application

In the STM32CubeU0 MCU Package, all example projects are generated with the STM32CubeMX tool to initialize the system, peripherals, and middleware.

Note: The direct use of an existing example project from the STM32CubeMX tool requires STM32CubeMX 6.11 or higher.

- After the installation of STM32CubeMX, open and, if necessary, update a proposed project. The quickest way to open an existing project is to double-click on the `*.ioc` file so STM32CubeMX automatically opens the project and its source files.

- The initialization source code of such projects is generated by STM32CubeMX; the main application source code is contained by the comments `USER CODE BEGIN` and `USER CODE END`. In case the IP selection and settings are modified, STM32CubeMX updates the initialization part of the code but preserves the main application source code.

To develop a custom project in STM32CubeMX, follow this step-by-step process:

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

For more information about STM32CubeMX, refer to the user manual *STM32CubeMX for STM32 configuration and initialization C code generation* (UM1718).

For a list of the available example projects for STM32CubeU0, refer to the application note *STM32Cube firmware examples for the STM32U0 series* (AN6063).

5.2.2 HAL application

This section describes the steps required to create a custom HAL application using STM32CubeU0.

1. Create a project
   To create a new project, either start from the `template` project, provided for each board in `\Projects\<STM32xxx_yyy>\Templates`, or from any available project in `\Projects\<STM32xxx_yyy>\Examples` or `\Projects\<STM32xxx_yyy>\Applications` (where `<STM32xxx_yyy>` refers to the board name, such as NUCLEO-U083RC).
   The template project provides only an empty main loop function, which is a good starting point for understanding theSTM32CubeU0 project settings. The template has the following characteristics:
   – It contains the HAL source code and CMSIS, and BSP drivers that form the minimum set of components required to develop code on a given board.
   – It contains the include paths for all firmware components.
   – It defines the supported STM32U0 series devices, allowing the configuration of the CMSIS and HAL drivers.
   – It provides ready-to-use user files that are preconfigured as shown below:
     ◦ HAL initialized with default time base with Arm® core SysTick.
     ◦ SysTick ISR implemented for `HAL_Delay()` purpose.

Note: When copying an existing project to another location, make sure all the include paths are updated.

2. **Add the necessary middleware to the project (optional)**
   The available middleware stacks are USBX library, Azure® RTOS, Touch Sensing. To identify the source files to be added to the project file list, refer to the documentation provided for each middleware component. Refer to the applications in `\Projects\STM32xxx_yyy\Applications\<MW_Stack>` (where `<MW_Stack>` refers to the middleware stack, such as USBX) to know which source files and include paths to add.
3. **Configure the firmware components**
   The HAL and middleware components offer a set of build-time configuration options, using macros (`#define`) declared in a header file. A template configuration file is provided within each component that has to be copied to the project folder (usually the configuration file is named `xxx_conf_template.h`, and the word *"_template"* needs to be removed when copying it to the project folder). The configuration file provides enough information to understand the impact of each configuration option. More detailed information is available in the documentation provided for each component.

4. **Start the HAL library**
   After jumping to the main program, the application code must call the `HAL_Init()` API to initialize the HAL library, which carries out the following tasks:
   a. Configuration of the flash memory prefetch and SysTick interrupt priority (through macros defined in `stm32u0xx_hal_conf.h`).
   b. Configuration of the SysTick to generate an interrupt every millisecond at the SysTick interrupt priority `TICK_INT_PRIORITY`, defined in `stm32u0xx_hal_conf.h`, which is clocked by the MSI (at this stage, the clock has not been configured yet and the system is running from the internal 16 MHz MSI).
   c. Setting the NVIC group priority to 0.
   d. Calling the `HAL_MspInit()` callback function defined in the `stm32u0xx_hal_msp.c` user file to perform global low-level hardware initializations.
5. **Configure the system clock**
   The system clock configuration is done by calling the two APIs described below:
   – `HAL_RCC_OscConfig()`: this API configures the internal and/or external oscillators, as well as the PLL source and factors. The user chooses to configure one or all oscillators. They can skip the PLL configuration if there is no need to run the system at a high frequency.
   – `HAL_RCC_ClockConfig()`: this API configures the system clock source, the flash memory latency, the AHB prescalers, and the APB prescalers.

   Initialize the peripheral
   a. First, write the peripheral `HAL_PPP_MspInit` function by proceeding as follows:
      i. Enable the peripheral clock.
      ii. Configure the peripheral GPIOs.
      iii. Configure the DMA channel and enable DMA interrupt (if needed).
      iv. Enable peripheral interrupt (if needed).
   b. Edit `stm32xxx_it.c` to call the required interrupt handlers (peripheral and DMA), if necessary.
   c. Write the process complete callback functions if peripheral interrupt or DMA is going to be used.
   d. In `main.c`, initialize the peripheral handle structure, then call the `HAL_PPP_Init()` function to initialize the peripheral.
6. **Develop an application**
   At this stage, the system is ready and the user application code development can start.
   a. The HAL provides intuitive and ready-to-use APIs to configure the peripheral. It supports polling, interrupts, and a DMA programming model, to accommodate any application requirements. For more details on how to use each peripheral, refer to the rich example set provided in the STM32CubeU0 MCU Package.
   b. If the application has real-time constraints, STM32CubeU0 provides a large set of examples showing how to use FreeRTOS™ and integrate it with all middleware stacks, making it a good starting point for developing an application.

Caution: In the default HAL implementation, a SysTick timer is used as the timebase; it generates interrupts at regular time intervals. If `HAL_Delay()` is called from the peripheral ISR process, make sure that the SysTick interrupt has a higher priority (numerically lower) than the peripheral interrupt. Otherwise, the caller ISR process is blocked. Functions affecting timebase configurations are declared as `__weak` to make an override possible in case of other implementations in the user file (using a general-purpose timer or other time source). For more details, refer to the `HAL_TimeBase` example.

5.2.3 LL application

This section describes the steps needed to create a custom LL application using STM32CubeU0.

1. **Create a project**
   To create a new project, either start from the *Templates_LL* project provided for each board in `\Projects\<STM32xxx_yyy>\Templates_LL` or from any available project in `\Projects\<STM32xxx_yyy>\Examples_LL` (`<STM32xxx_yyy>` refers to the board name, such as NUCLEO-U083RC).
   The template project provides an empty main loop function, which is a good starting point for understanding the STM32CubeU0 project settings. The template has the following characteristics:
   – It contains the source codes of the LL and CMSIS drivers that form the minimum set of components required to develop code on a given board.
   – It contains the include paths for all required firmware components.
   – It selects the supported STM32U0 series device and allows the correct configuration of the CMSIS and LL drivers.
   – It provides ready-to-use user files, which are preconfigured as follows:
     ◦ `main.h`: LED and USER_BUTTON definition abstraction layer.
     ◦ `main.c`: system clock configuration for maximum frequency.
2. **Port an existing project to another board**
   a. Start from the Templates_LL project provided for each board, available in the `\Projects\<STM32xxx_yyy>\Templates_LL` folder.
   b. Select an LL example.

Note: To find the board on which LL examples are deployed, refer to the list of LL examples in `STM32CubeProjectsList.html`.

3. **Port the LL example**
   – Copy/paste the `Templates_LL` folder to keep the initial source, or directly update an existing `Templates_LL` project.
   – Replace `Templates_LL` files with the `Examples_LL` targeted project files.
   – Keep all board-specific parts. For clarity reasons, board-specific parts have been flagged with the following specific tags:

```
/* ============== BOARD SPECIFIC CONFIGURATION CODE BEGIN ============== */
/* ============== BOARD SPECIFIC CONFIGURATION CODE END ============== */
```

   The main porting steps are the following:
   a. Replace the `stm32u0xx_it.h` file.
   b. Replace the `stm32u0xx_it.c` file.
   c. Replace the `main.h` file and update it. Keep the LED and user button definition from the LL template under the `"BOARD SPECIFIC CONFIGURATION"` tags.
   d. Replace the `main.c` file and update it:
      • Keep the clock configuration of the `SystemClock_Config()` LL template function under the `"BOARD SPECIFIC CONFIGURATION"` tags.
      • Depending on the LED definition, replace each LEDx occurrence with another LEDy available in the `main.h` file.

With these modifications, the example can now run on the targeted board.

5.3 Getting STM32CubeU0 release updates

The STM32CubeU0 MCU 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 and proposes to download them to the user's computer.

Installing and running the STM32CubeUpdater program

Follow the steps below to install and run STM32CubeUpdater.

1. Double-click the `SetupSTM32CubeUpdater.exe` file to launch the installation.

2. Accept the license terms and follow the different installation steps.
3. Upon successful installation, STM32CubeUpdater becomes available as an STMicroelectronics program in the *Program Files* folder and is launched automatically. The STM32CubeUpdater icon appears in the system tray.
4. Right-click the updater icon and select [**Updater Settings**] to configure the updater connection and whether to perform manual or automatic checks.

For more details on updater configuration, refer to section 3 of the user manual *STM32CubeMX for STM32 configuration and initialization C code generation* (UM1718).

6 FAQ

6.1 What is the license scheme for the STM32CubeU0 MCU Package?

The HAL is distributed under a nonrestrictive BSD (berkeley software distribution) license.

The middleware stacks made by STMicroelectronics (USB Device libraries, STM32_TouchSensing) come with a licensing model allowing easy reuse, provided it runs on an STMicroelectronics device.

The middleware components based on well-known open-source solutions (FreeRTOS™ and FatFS) have user-friendly license terms. For more details, refer to the relevant middleware license agreement.

6.2 What boards are supported by the STM32CubeU0 MCU Package?

The STM32CubeU0 MCU Package provides BSP drivers and ready-to-use examples for the following STM32CubeU0 series boards:

- NUCLEO-U031R8
- NUCLEO-U083RC
- STM32U083C-DK

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

Yes. STM32CubeU0 provides a rich set of examples and applications. They come with preconfigured projects for IAR Embedded Workbench®, Keil®, andSTM32CubeIDE.

6.4 Are there any links with standard peripheral libraries?

The STM32CubeU0 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 the features that are common to the peripherals rather than the hardware. A set of user-friendly APIs allow a higher abstraction level, rendering them easily portable from one product to another.
- The LL drivers offer low-layer register-level APIs. They are organized in a simpler and clearer way to avoid direct register access. LL drivers also include peripheral initialization APIs, which are more optimized compared to what the SPL offer, while being functionally similar. Compared to HAL drivers, these LL initialization APIs allow a straightforward migration from the SPL to the STM32CubeU0 LL drivers, since each SPL API has its equivalent LL API.

6.5 Does the HAL layer take advantage of interrupts or DMA? How can this be controlled?

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

6.6 How are the product-/peripheral-specific features managed?

The HAL drivers offer extended APIs, which are specific functions provided as add-ons to the common API to support features available on some products/lines only.

6.7 How can STM32CubeMX generate code based on embedded software?

STM32CubeMX has built-in knowledge of STM32 microcontrollers, including their peripherals and software. It provides a graphical representation to the user and can generate `*.h` and `*.c` files with the user configuration.

6.8 How to get regular updates on the latest STM32CubeU0 MCU Package releases?

The STM32CubeU0 MCU Package comes with an updater utility, STM32CubeUpdater, that is configurable for automatic or on-demand checks for new firmware package updates (new releases and patches).

STM32CubeUpdater is integrated within the STM32CubeMX tool. When using this tool for STM32U0 configuration and initialization C code generation, the user benefits from STM32CubeU0 auto-updates as well as STM32CubeU0 MCU Package updates.

For more details, refer to Section 5.3: Getting STM32CubeU0 release updates.

6.9 When to use the HAL versus LL drivers?

HAL drivers offer high-level and function-oriented APIs, with a high level of portability. The product/IP complexity is hidden from end users.

LL drivers offer low-layer register level APIs, with a better optimization but less portable. They require in-depth knowledge of product/IP specifications.

6.10 How to include LL drivers in an existing environment? Is there an LL configuration file, like for HAL?

There is no configuration file. Source code should directly include the necessary `stm32u0xx_ll_ppp.h` file(s).

6.11 Can HAL and LL drivers be used together? If so, what are the constraints?

It is possible to use both HAL and LL drivers. Use the HAL drivers for the IP initialization phase and manage the I/O operations with the LL drivers.

The major difference between HAL and LL is that HAL drivers require the creation and use of handles for operation management, while LL drivers operate directly on the peripheral registers. The mixing of HAL and LL is illustrated in the *Examples_MIX* example.

6.12 Are there any LL APIs that are not available with HAL?

Yes, there are. A few Cortex® APIs have been added in `stm32u0xx_ll_cortex.h`, for instance, to access the SCB or SysTick registers.

6.13 Why are SysTick interrupts not enabled on LL drivers?

When using LL drivers in standalone mode, there is no need to enable SysTick interrupts because they are not used in LL APIs, while HAL functions require SysTick interrupts to manage timeouts.

6.14 How are LL initialization APIs enabled?

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

To use LL initialization APIs, add this switch to the toolchain compiler preprocessor.

Revision history

Table 4. **Document revision history**

Date	Revision	Changes
31-Jan-2024	1	Initial release.

Contents

List of tables

List of figures