Figure 190. Block remapping - example 1

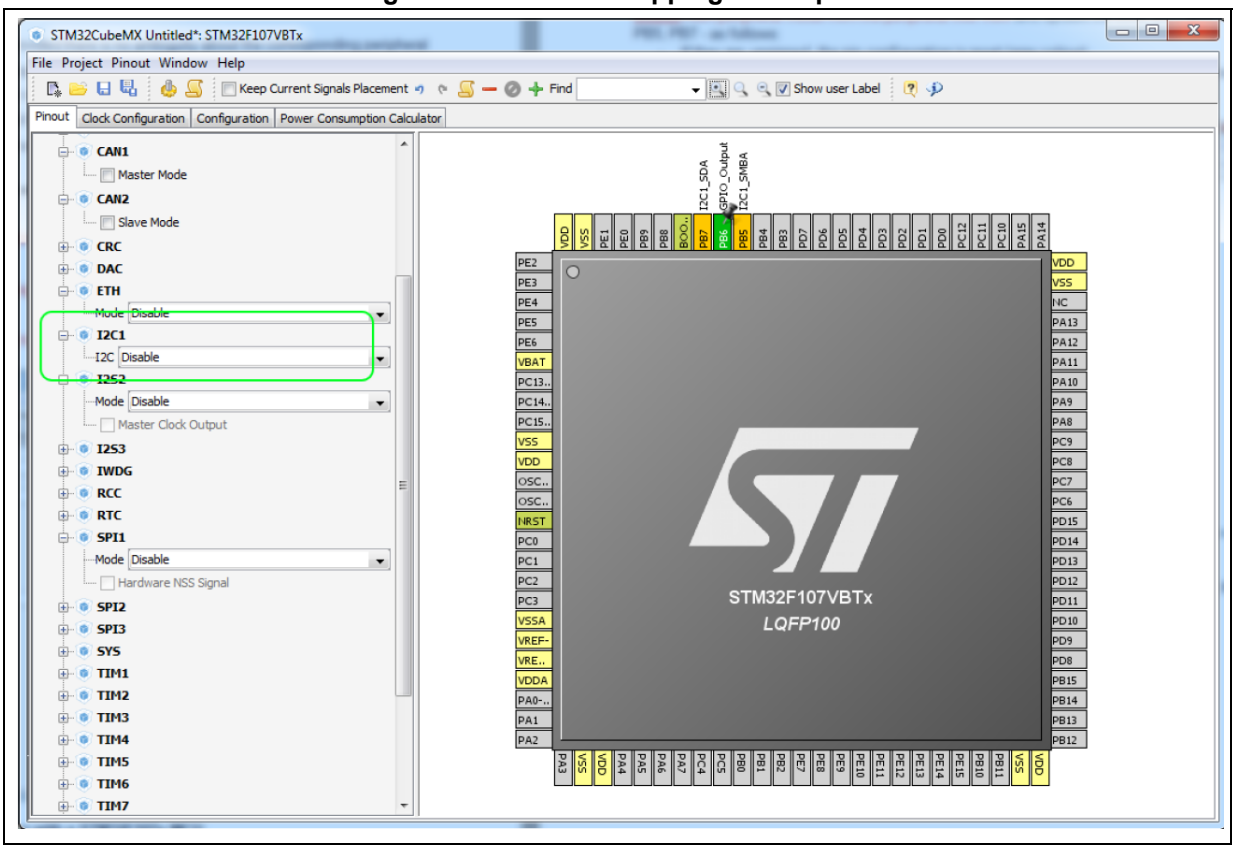
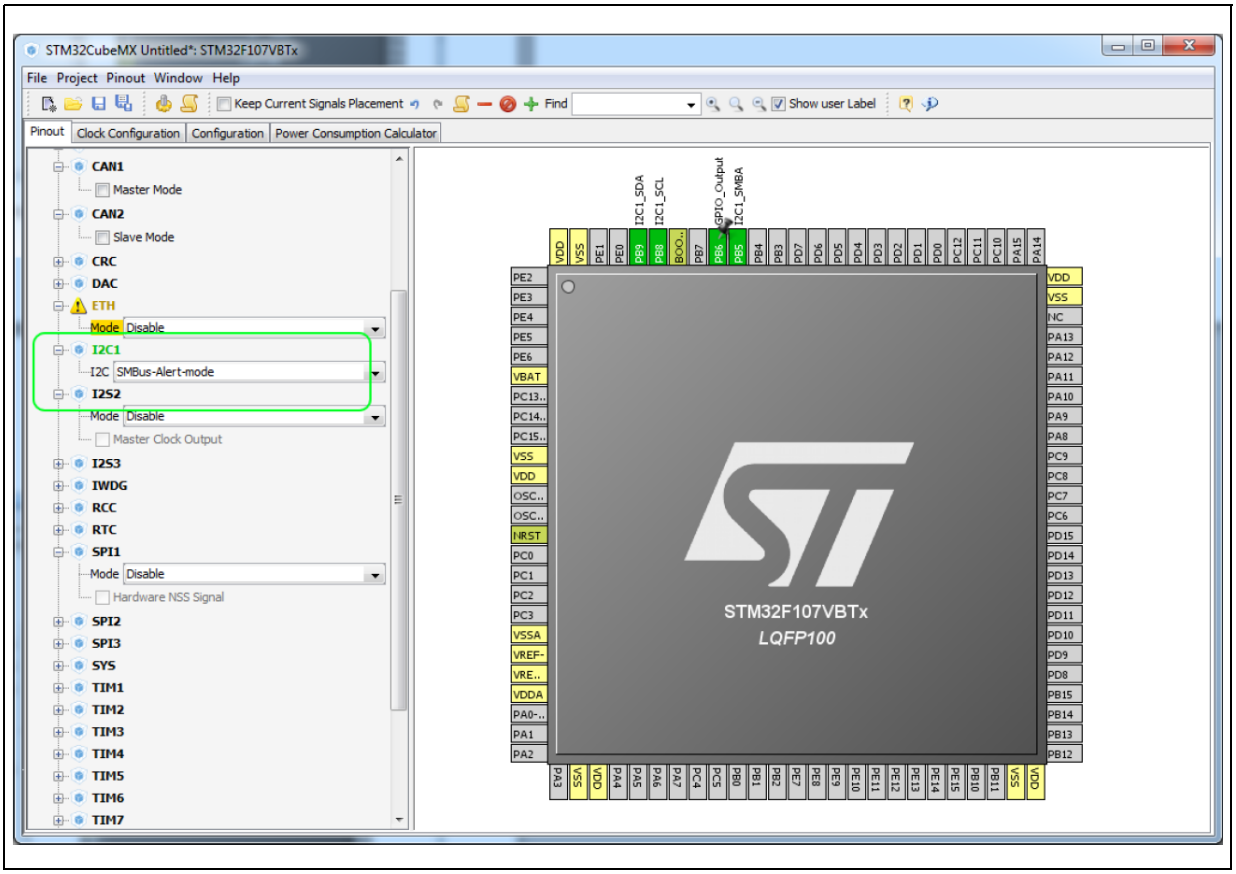


Figure 191. Block remapping - example 2



A.2 Block inter-dependency

On the **Chip** view, the same signal can appear as an alternate function for multiple pins. However it can be mapped only once.

As a consequence, for STM32F1 MCUs, two blocks of pins cannot be selected simultaneously for the same peripheral mode: when a block/signal from a block is selected, the alternate blocks are cleared.

Example of block remapping of SPI in full-duplex master mode with a STM32F107x MCU

If SPI1 full-duplex master mode is selected from the tree view, by default the corresponding SPI signals are assigned to PB3, PB4 and PB5 pins (see *Figure 192*).

If the user assigns to PA6 the SPI1_MISO function currently assigned to PB4, STM32CubeMX clears the PB4 pin from the SPI1_MISO function, as well as all the other pins configured for this block, and moves the corresponding SPI1 functions to the relevant pins in the same block as the PB4 pin (see *Figure 193*).

(by pressing CTRL and clicking PB4 to show PA6 alternate function in blue, then drag and drop the signal to pin PA6)

Figure 192. Block inter-dependency - SPI signals assigned to PB3/4/5

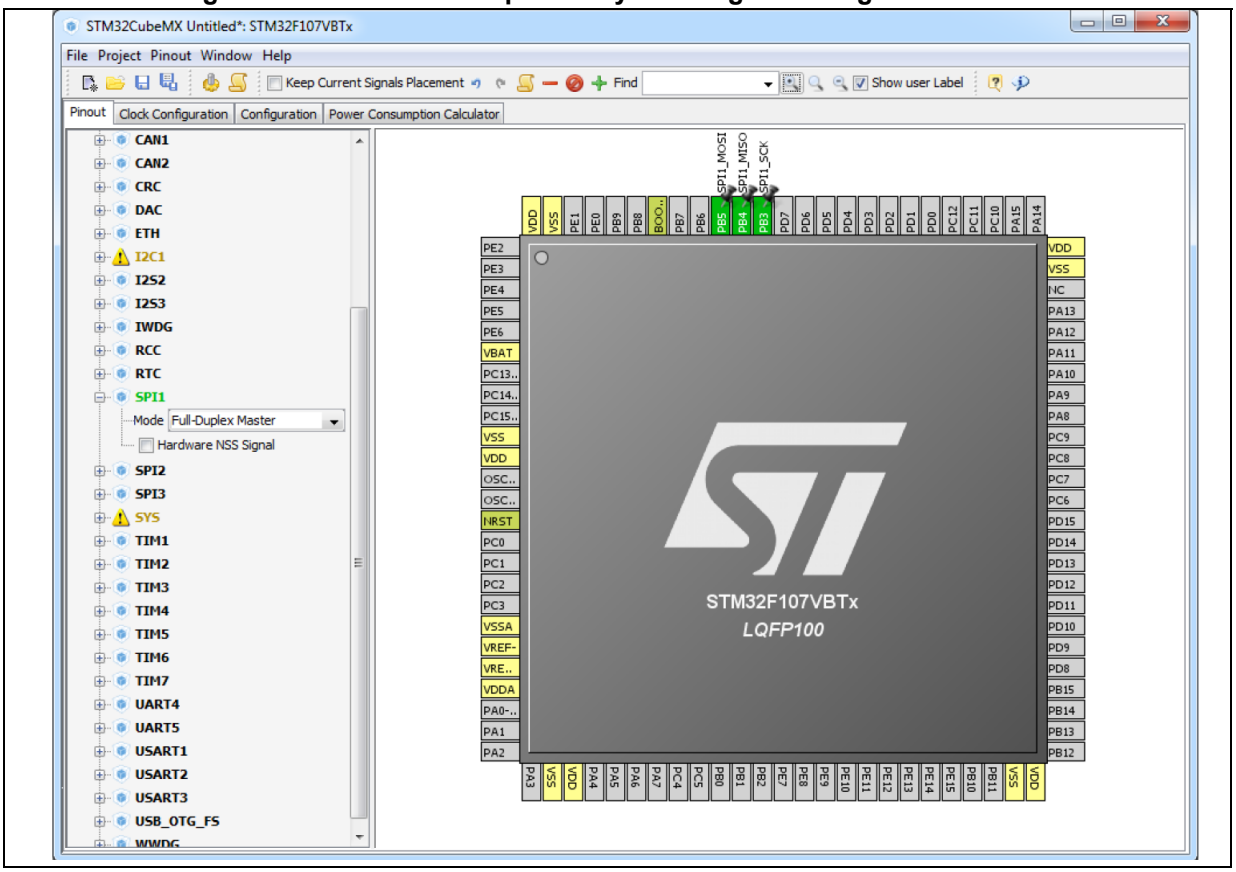
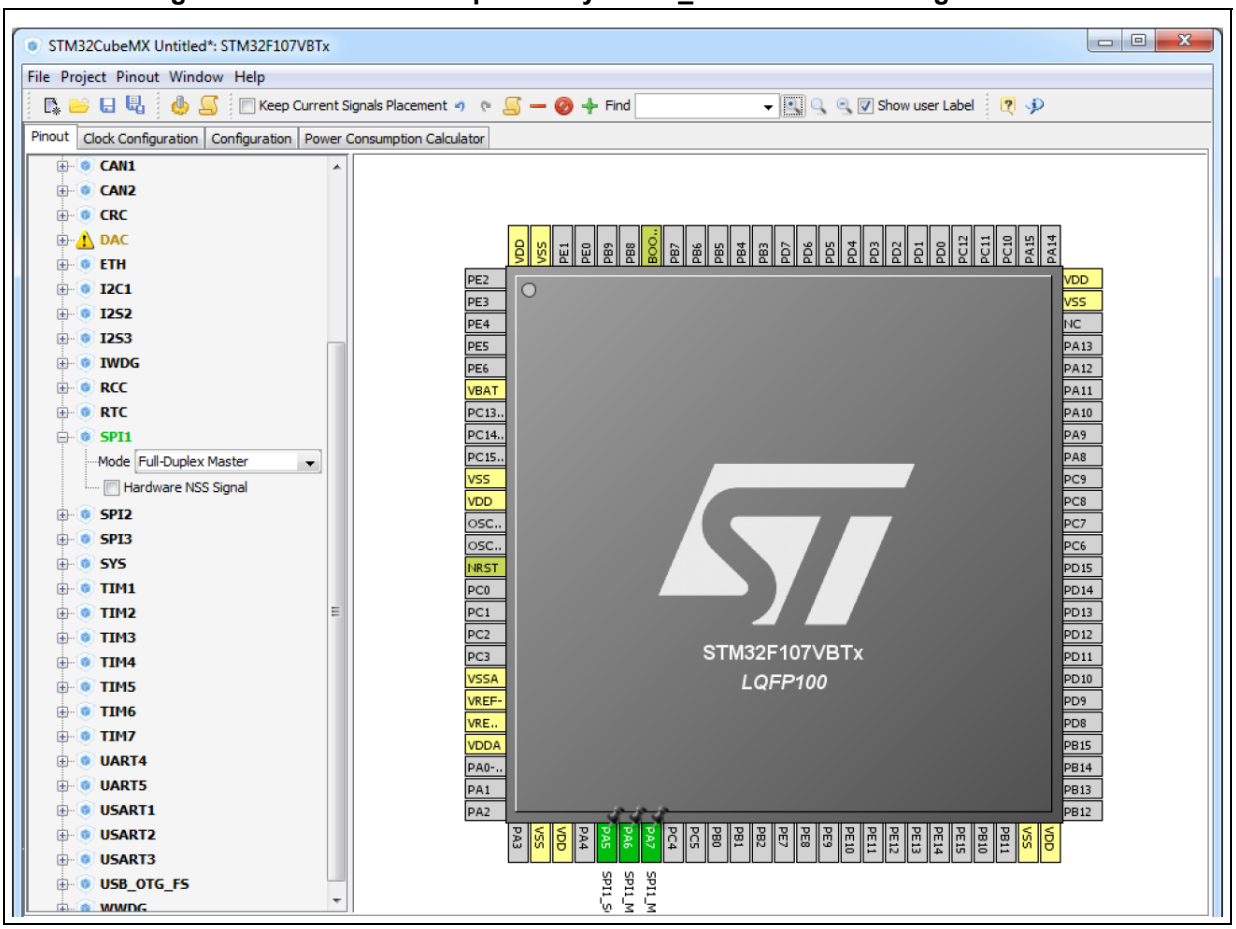


Figure 193. Block inter-dependency - SPI1_MISO function assigned to PA6



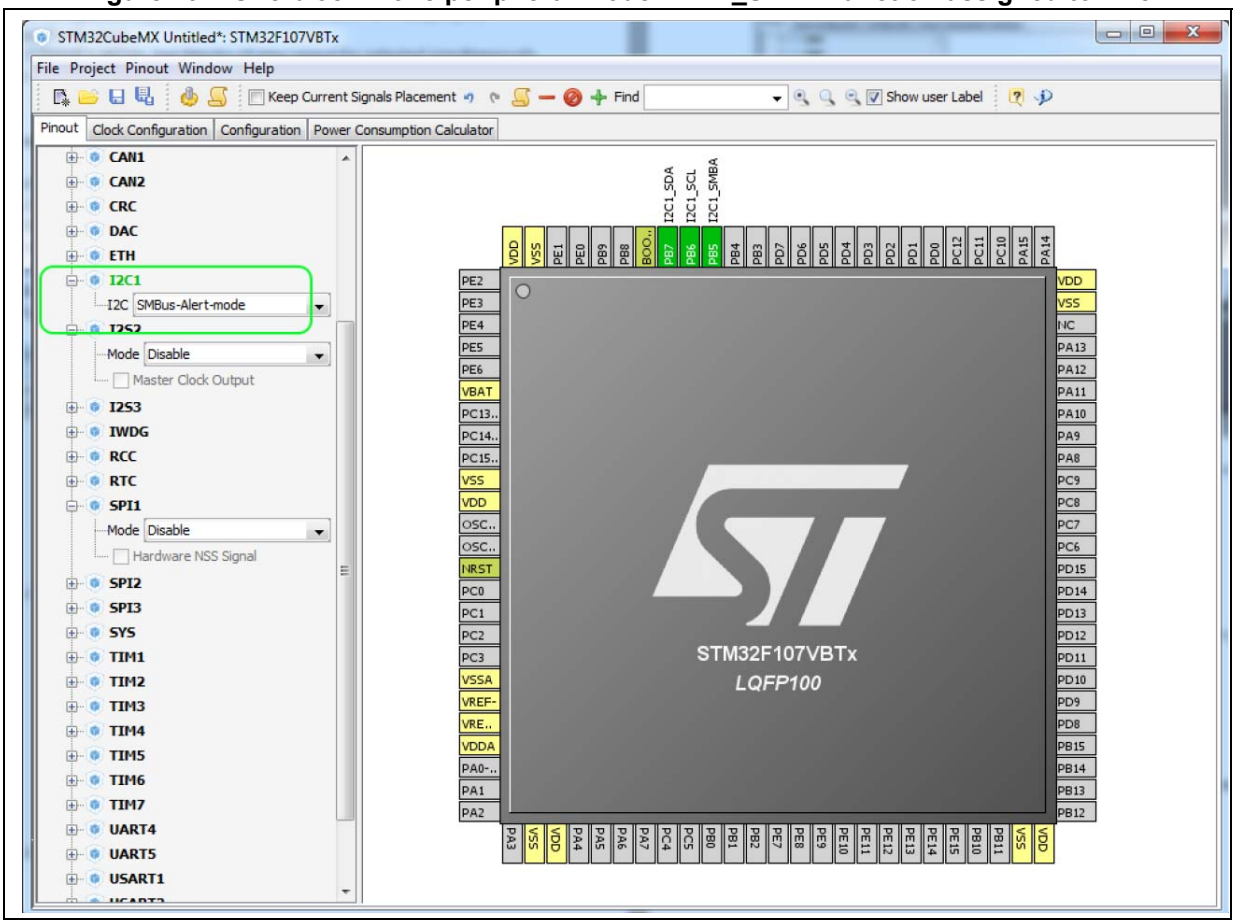
A.3 One block = one peripheral mode

When a block of pins is fully configured in the **Chip** view (shown in green), the related peripheral mode is automatically set in the Peripherals tree.

Example of STM32F107x MCU

Assigning the I2C1_SMBA function to PB5 automatically configures I2C1 peripheral in SMBus-Alert mode (see Peripheral tree in *Figure 194*).

Figure 194. One block = one peripheral mode - I2C1_SMBA function assigned to PB5



A.4 Block remapping (STM32F10x only)

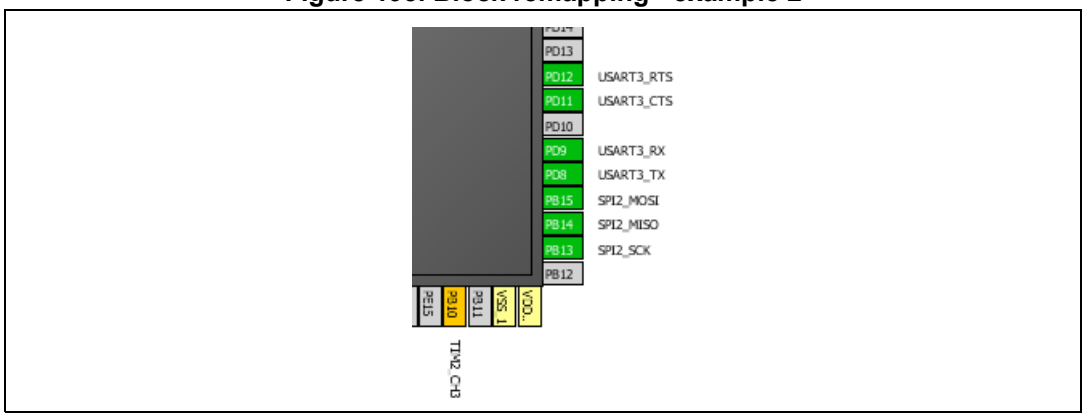
To configure a peripheral mode, STM32CubeMX selects a block of pins and assigns each mode signal to a pin in this block. In doing so, it looks for the first free block to which the mode can be mapped.

When setting a peripheral mode, if at least one pin in the default block is already used, STM32CubeMX tries to find an alternate block. If none can be found, it either selects the functions in a different sequence, or unchecks Keep Current Signals Placement, and remaps all the blocks to find a solution.

Example

STM32CubeMX remaps USART3 hardware-flow-control mode to the (PD8-PD9-PD11-PD12) block, because PB14 of USART3 default block is already allocated to the SPI2_MISO function (see *Figure 195*).

Figure 195. Block remapping - example 2



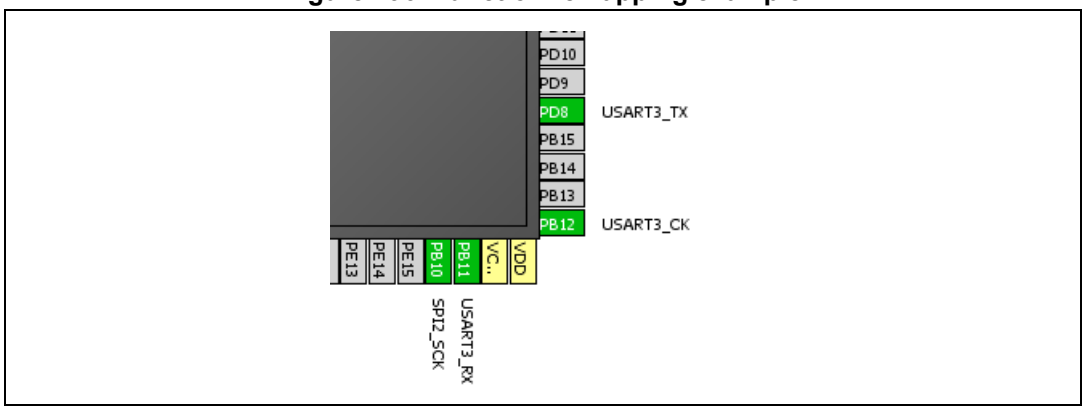
A.5 Function remapping

To configure a peripheral mode, STM32CubeMX assigns each signal of the mode to a pin. In doing so, it will look for the first free pin the signal can be mapped to.

Example using STM32F415x

When configuring USART3 for the Synchronous mode, STM32CubeMX discovered that the default PB10 pin for USART3_TX signal was already used by SPI. It thus remapped it to PD8 (see *Figure 196*).

Figure 196. Function remapping example



A.6 Block shifting (only for STM32F10x and when “Keep Current Signals placement” is unchecked)

If a block cannot be mapped and there are no free alternate solutions, STM32CubeMX tries to free the pins by remapping all the peripheral modes impacted by the shared pin.

Example

With the Keep current signal placement enabled, if USART3 synchronous mode is set first, the Asynchronous default block (PB10-PB11) is mapped and Ethernet becomes unavailable (shown in red) (see *Figure 197*).

Unchecking Keep Current Signals Placement allows STM32CubeMX shifting blocks around and freeing a block for the Ethernet MII mode. (see *Figure 198*).

Figure 197. Block shifting not applied

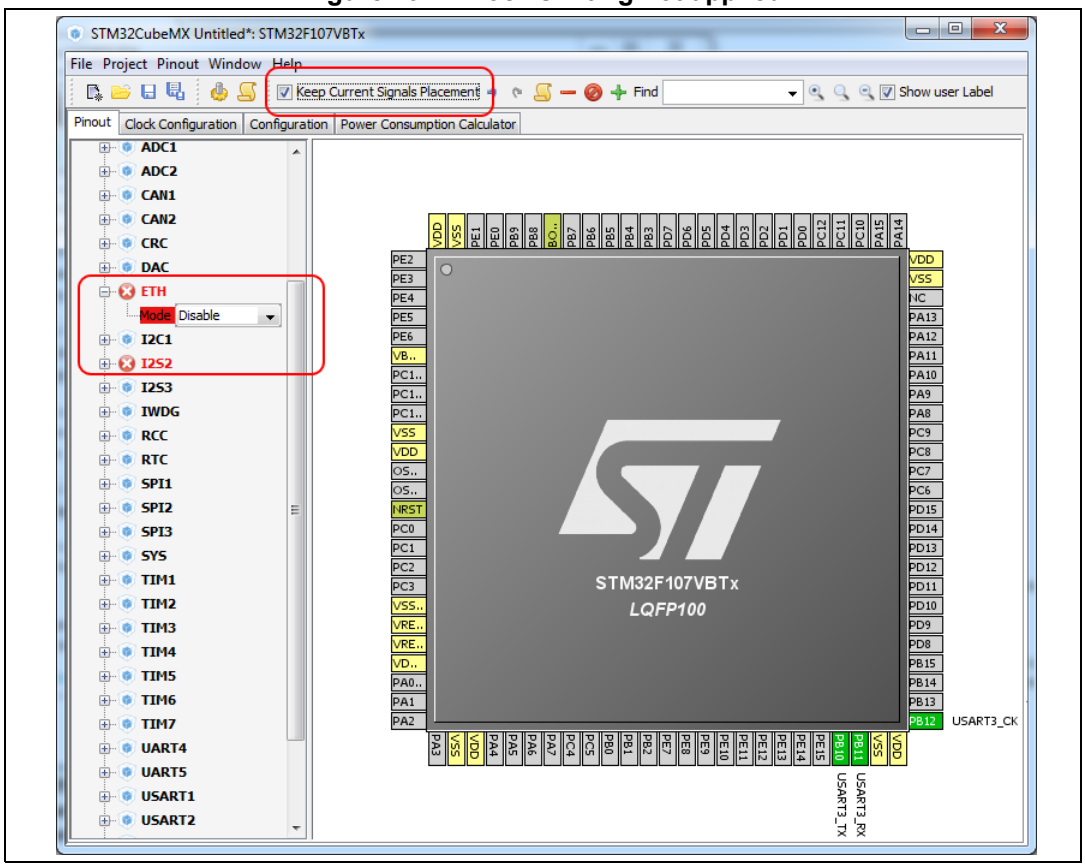
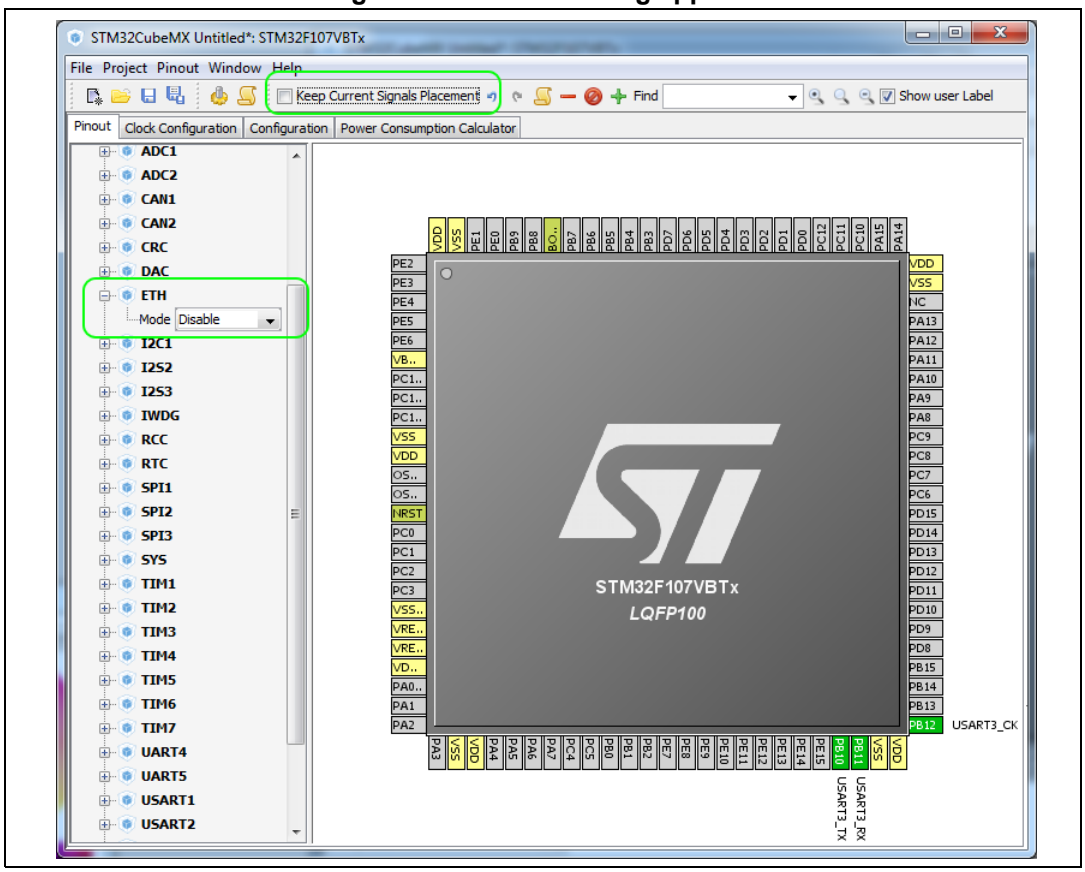


Figure 198. Block shifting applied



A.7 Setting and clearing a peripheral mode

The Peripherals panel and the **Chip** view are linked: when a peripheral mode is set or cleared, the corresponding pin functions are set or cleared.

A.8 Mapping a function individually

When STM32CubeMX needs a pin that has already been assigned manually to a function (no peripheral mode set), it can move this function to another pin, only if Keep Current Signals Placement is unchecked and the function is not pinned (no pin icon).

A.9 GPIO signals mapping

I/O signals (GPIO_Input, GPIO_Output, GPIO_Analog) can be assigned to pins either manually through the **Chip** view or automatically through the Pinout menu. Such pins can no longer be assigned automatically to another signal: STM32CubeMX signal automatic placement does not take into account this pin anymore since it does not shift I/O signals to other pins.

The pin can still be manually assigned to another signal or to a reset state.

Appendix B STM32CubeMX C code generation design choices and limitations

This section summarizes STM32CubeMX design choices and limitations.

B.1 STM32CubeMX generated C code and user sections

The C code generated by STM32CubeMX provides user sections as illustrated below. They allow user C code to be inserted and preserved at next C code generation.

User sections shall neither be moved nor renamed. Only the user sections defined by STM32CubeMX are preserved. User created sections will be ignored and lost at next C code generation.

```
 /* USER CODE BEGIN 0 */
(..)
/* USER CODE END 0 */
```

Note: STM32CubeMX may generate C code in some user sections. It will be up to the user to clean the parts that may become obsolete in this section. For example, the while(1) loop in the main function is placed inside a user section as illustrated below:

```
/* Infinite loop */
  /* USER CODE BEGIN WHILE */
  while (1)
  {
  /* USER CODE END WHILE */

  /* USER CODE BEGIN 3 */
  }
/* USER CODE END 3 */
```

B.2 STM32CubeMX design choices for peripheral initialization

STM32CubeMX generates peripheral *_Init* functions that can be easily identified thanks to the MX_ prefix:

```
static void MX_GPIO_Init(void);
static void MX_<Peripheral Instance Name>_Init(void);
static void MX_I2S2_Init(void);
```

An *MX_<peripheral instance name>_Ini*t function exists for each peripheral instance selected by the user (e.g, *MX_I2S2_Init*). It performs the initialization of the relevant handle structure (e.g, &hi2s2 for I2S second instance) that is required for HAL driver initialization (e.g., *HAL_I2S_Init*) and the actual call to this function:

```
void MX_I2S2_Init(void)
{
 hi2s2.Instance = SPI2;
  hi2s2.Init.Mode = I2S_MODE_MASTER_TX;
  hi2s2.Init.Standard = I2S_STANDARD_PHILLIPS;
```

```
  hi2s2.Init.DataFormat = I2S_DATAFORMAT_16B;
  hi2s2.Init.MCLKOutput = I2S_MCLKOUTPUT_DISABLE;
  hi2s2.Init.AudioFreq = I2S_AUDIOFREQ_192K;
  hi2s2.Init.CPOL = I2S_CPOL_LOW;
  hi2s2.Init.ClockSource = I2S_CLOCK_PLL;
  hi2s2.Init.FullDuplexMode = I2S_FULLDUPLEXMODE_ENABLE;
  HAL_I2S_Init(&hi2s2);
}
```

By default, the peripheral initialization is done in *main.c*. If the peripheral is used by a middleware mode, the peripheral initialization can be done in the middleware corresponding .c file.

Customized *HAL_<IP Name>_MspInit()* functions are created in the stm32f4xx_hal_msp.c file to configure the low level hardware (GPIO, CLOCK) for the selected IPs.

B.3 STM32CubeMX design choices and limitations for middleware initialization

B.3.1 Overview

STM32CubeMX does not support C user code insertion in Middleware stack native files although stacks such as LwIP might require it in some use cases.

STM32CubeMX generates middleware *Init* functions that can be easily identified thanks to the MX_ prefix:

```
MX_LWIP_Init(); // defined in lwip.h file
MX_USB_HOST_Init(); // defined in usb_host.h file
MX_FATFS_Init(); // defined in fatfs.h file
```

Note however the following exceptions:

- No *Init* function is generated for FreeRTOS unless the user chooses, from the Project settings window, to generate *Init* functions as pairs of .c/.h files. Instead, a *StartDefaultTask* function is defined in the *main.c* file and CMSIS-RTOS native function (*osKernelStart*) is called in the main function.
- If FreeRTOS is enabled, the *Init* functions for the other middlewares in use are called from the *StartDefaultTask* function in the main.c file.

  Example:

```
void StartDefaultTask(void const * argument)
{
   /* init code for FATFS */
   MX_FATFS_Init();
   /* init code for LWIP */
   MX_LWIP_Init();
  /* init code for USB_HOST */
  MX_USB_HOST_Init();
  /* USER CODE BEGIN 5 */
  /* Infinite loop */
```

```
  for(;;)
  {
    osDelay(1);
  }
  /* USER CODE END 5 */
}
```

B.3.2 USB Host

USB peripheral initialization is performed within the middleware initialization C code in the *usbh_conf.c* file, while USB stack initialization is done within the *usb_host.c* file.

When using the USB Host middleware, the user is responsible for implementing the *USBH_UserProcess* callback function in the generated *usb_host.c* file.

From STM32CubeMX user interface, the user can select to register one class or all classes if the application requires switching dynamically between classes.

B.3.3 USB Device

USB peripheral initialization is performed within the middleware initialization C code in the *usbd_conf.c* file, while USB stack initialization is done within the *usb_device.*c file.

USB VID, PID and String standard descriptors are configured via STM32CubeMX user interface and available in the *usbd_desc.c* generated file. Other standard descriptors (configuration, interface) are hard-coded in the same file preventing support for USB composite devices.

When using the USB Device middleware, the user is responsible for implementing the functions in the *usbd_<classname>_if.c* class interface file for all device classes (e.g., usbd_storage_if.c).

USB MTP and CCID classes are not supported.

B.3.4 FatFs

FatFs configuration is available in the *ffconf.h* generated file.

The initialization of the SDIO peripheral for the FatFs SD Card mode and of the FMC peripheral for the FatFs External SDRAM and External SRAM modes are kept in the *main.c* file.

Some files need to be modified by the user to match user board specificities (BSP drivers in STM32Cube embedded software package can be used as example):

- *bsp_driver_sd.c/.h* generated files when using FatFs SD Card mode
- *bsp_driver_sram.c/.h* generated files when using FatFs External SRAM mode
- *bsp_driver_sdram.c/.h* generated files when using FatFs External SDRAM mode.

Multi-drive FatFs is supported, which means that multiple logical drives can be used by the application (External SDRAM, External SRAM, SD Card, USB Disk, User defined). However support for multiple instances of a given logical drive is not available (e.g. FatFs using two instances of USB hosts or several RAM disks).

NOR and NAND Flash memory are not supported. In this case, the user shall select the FatFs user-defined mode and update the *user_diskio.c* driver file generated to implement the interface between the middleware and the selected peripheral.

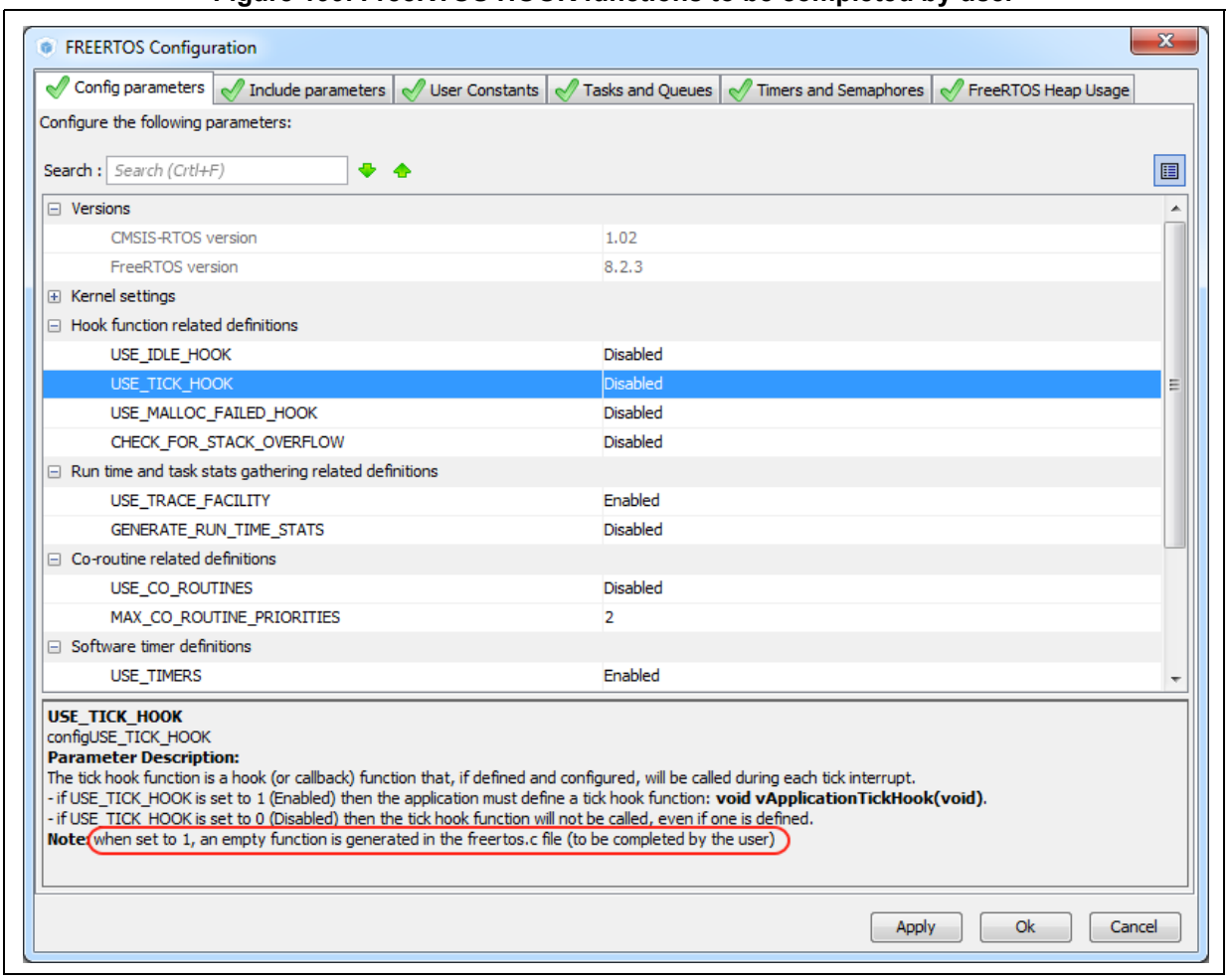
B.3.5 FreeRTOS

FreeRTOS configuration is available in *FreeRTOSConfig.h* generated file.

When FreeRTOS is enabled, all other selected middleware modes (e.g., LwIP, FatFs, USB) will be initialized within the same FreeRTOS thread in the main.c file.

When GENERATE_RUN_TIME_STATS, CHECK_FOR_STACK_OVERFLOW, USE_IDLE_HOOK, USE_TICK_HOOK and USE_MALLOC_FAILED_HOOK parameters are activated, STM32CubeMX generates *freertos.c* file with empty functions that the user shall implement. This is highlighted by the tooltip (see *Figure 199*).

Figure 199. FreeRTOS HOOK functions to be completed by user



B.3.6 LwIP

LwIP initialization function is defined in *lwip.c*, while LwIP configuration is available in *lwipopts.h* generated file.

STM32CubeMX supports LwIP over Ethernet only. The Ethernet peripheral initialization is done within the middleware initialization C code.

STM32CubeMX does not support user C code insertion in stack native files. However, some LwIP use cases require modifying stack native files (e.g., *cc.h, mib2.c*): user modifications shall be backed up since they will be lost at next STM32CubeMX generation.

Starting with release 1.5, STM32CubeMX LwIP supports IPv6 (see *Figure 202*).

DHCP must be disabled, to configure a static IP address.

Figure 200. LwIP 1.4.1 configuration

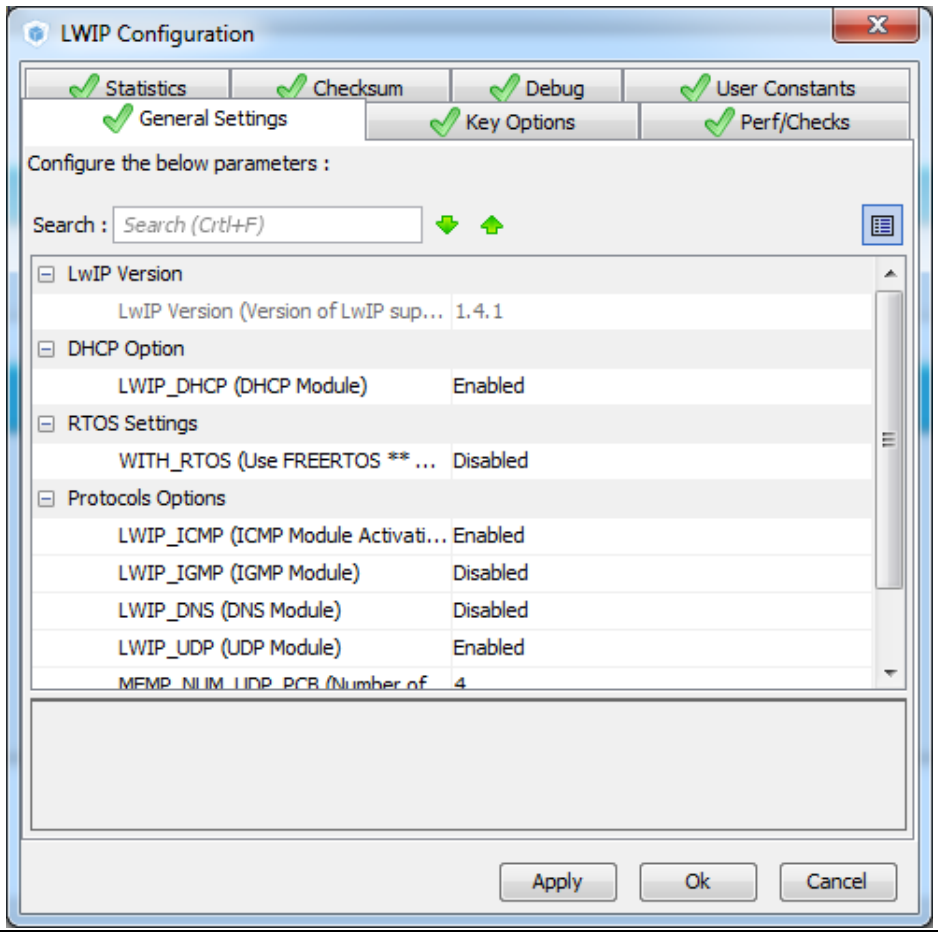
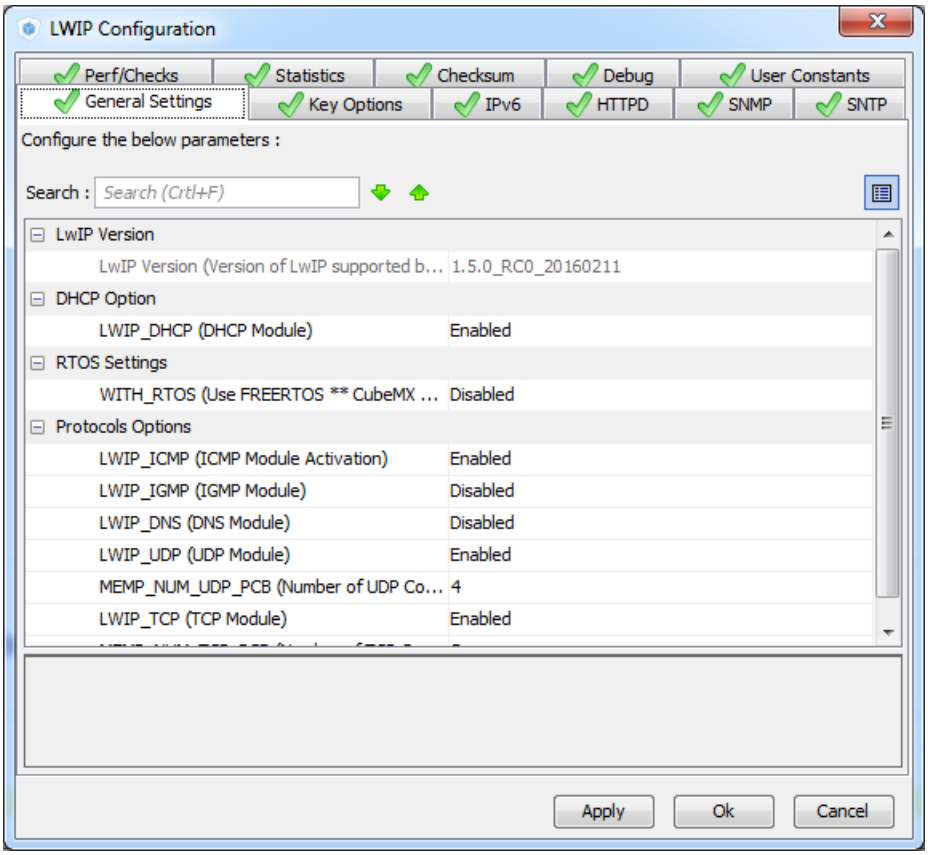


Figure 201. LwIP 1.5 configuration



STM32CubeMX generated C code will report compilation errors when specific parameters are enabled (disabled by default). The user must fix the issues with a stack patch (downloaded from Internet) or user C code. The following parameters generate an error:

- MEM_USE_POOLS: user C code to be added either in *lwipopts.h* or in *cc.h* (stack file).
- PPP_SUPPORT, PPPOE_SUPPORT: user C code required
- MEMP_SEPARATE_POOLS with MEMP_OVERFLOW_CHECK > 0: a stack patch required
- MEM_LIBC_MALLOC & RTOS enabled: stack patch required
- LWIP_EVENT_API: stack patch required

In STM32CubeMX, the user must enable FreeRTOS in order to use LwIP with the netconn and sockets APIs. These APIs require the use of threads and consequently of an operating system. Without FreeRTOS, only the LwIP event-driven raw API can be used.

Appendix C STM32 microcontrollers naming conventions

STM32 microcontroller part numbers are codified following the below naming conventions:

- Device subfamilies

  The higher the number, the more features available.

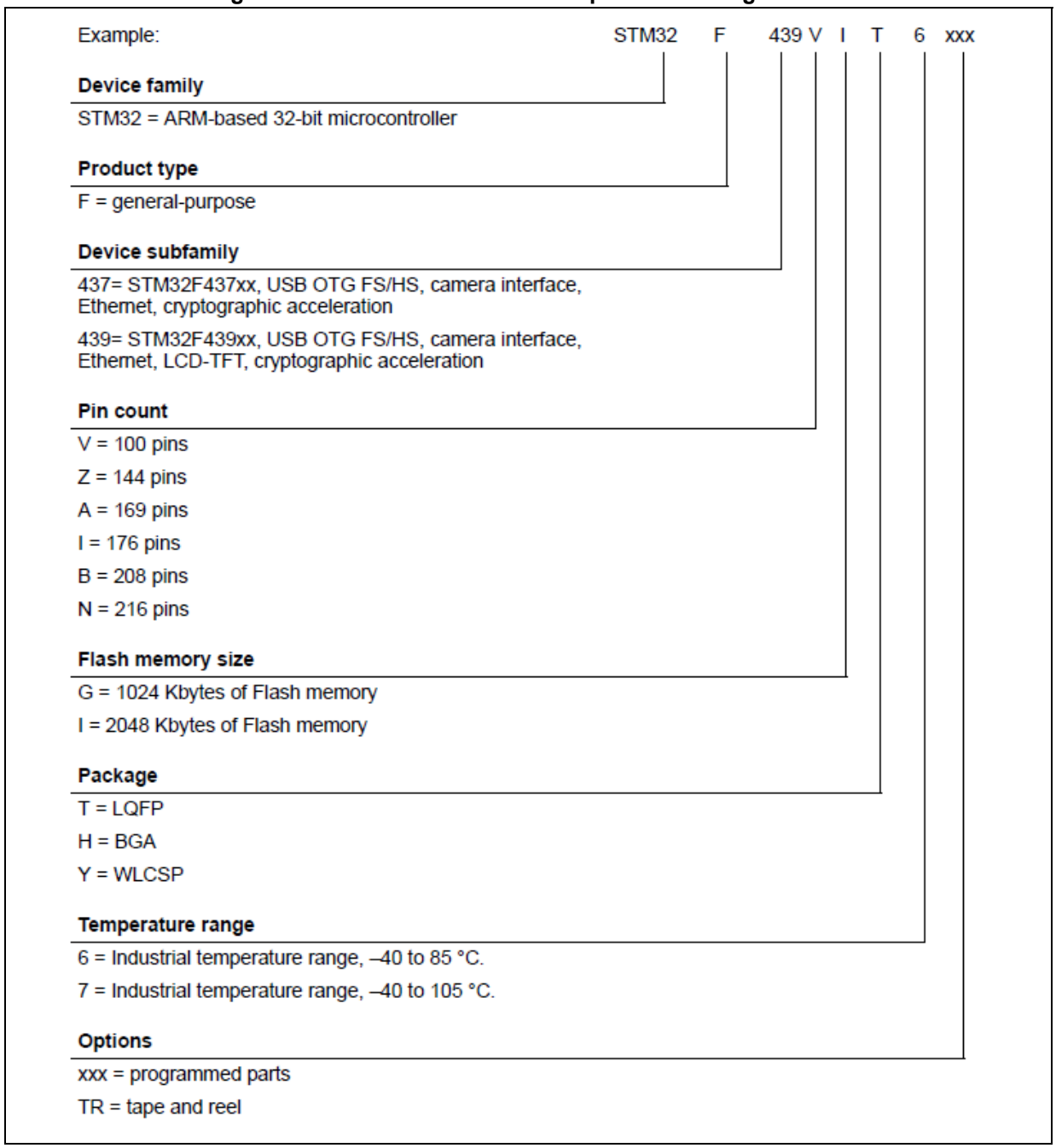
  For example STM32L0 line includes STM32L051, L052, L053, L061, L062, L063 subfamilies where STM32L06x part numbers come with AES while STM32L05x do not.

  The last digit indicates the level of features. In the above example:
  - 1 =Access line
  - 2 = with USB
  - 3 = with USB and LCD.
- Pin counts
  - F = 20 pins
  - G = 28 pins
  - K = 32 pins
  - T = 36 pins
  - S = 44 pins
  - C = 48 pins
  - R = 64 pins (or 66 pins)
  - M = 80 pins
  - O = 90 pins
  - V = 100 pins
  - Q= 132 pins (e. g. STM32L162QDH6)
  - Z=144
  - I=176 (+25)
  - B = 208 pins (e. g.: STM32F429BIT6)
  - N = 216 pins
- Flash memory sizes
  - 4 = 16 Kbytes of Flash memory
  - 6 = 32 Kbytes of Flash memory
  - 8 = 64 Kbytes of Flash memory
  - B = 128 Kbytes of Flash memory
  - C = 256 Kbytes of Flash memory
  - D = 384 Kbytes of Flash memory
  - E = 512 Kbytes of Flash memory
  - F = 768 Kbytes of Flash memory
  - G = 1024 Kbytes of Flash memory
  - I = 2048 Kbytes of Flash memory
- Packages
  - B = SDIP
  - H = BGA

- M = SO
- P = TSSOP
- T = LQFP
- U = VFQFPN
- Y = WLCSP

Figure 202 shows an example of STM32 microcontroller part numbering scheme.

Figure 202. STM32 microcontroller part numbering scheme



Appendix D STM32 microcontrollers power consumption parameters

This section provides an overview on how to use STM32CubeMX Power Consumption Calculator (PCC).

Microcontroller power consumption depends on chip size, supply voltage, clock frequency and operating mode. Embedded applications can optimize STM32 MCU power consumption by reducing the clock frequency when fast processing is not required and choosing the optimal operating mode and voltage range to run from. A description of STM32 power modes and voltage range is provided below.

D.1 Power modes

STM32 MCUs support different power modes (refer to STM32 MCU datasheets for full details).

D.1.1 STM32L1 series

STM32L1 microcontrollers feature up to 6 power modes, including 5 low-power modes:

- **Run** mode

  This mode offers the highest performance using HSE/HSI clock sources. The CPU runs up to 32 MHz and the voltage regulator is enabled.

- **Sleep** mode

  This mode uses HSE or HSI as system clock sources. The voltage regulator is enabled and the CPU is stopped. All peripherals continue to operate and can wake up the CPU when an interrupt/event occurs.

- **Low- power run** mode

  This mode uses the multispeed internal (MSI) RC oscillator set to the minimum clock frequency (131 kHz) and the internal regulator in low-power mode. The clock frequency and the number of enabled peripherals are limited.

- **Low-power sleep** mode

  This mode is achieved by entering Sleep mode. The internal voltage regulator is in low-power mode. The clock frequency and the number of enabled peripherals are limited. A typical example would be a timer running at 32 kHz.

  When the wakeup is triggered by an event or an interrupt, the system returns to the Run mode with the regulator ON.

- **Stop** mode

  This mode achieves the lowest power consumption while retaining RAM and register contents. Clocks are stopped. The real-time clock (RTC) an be backed up by using LSE/LSI at 32 kHz/37 kHz. The number of enabled peripherals is limited. The voltage regulator is in low-power mode.

  The device can be woken up from Stop mode by any of the EXTI lines.

- **Standby** mode

  This mode achieves the lowest power consumption. The internal voltage regulator is switched off so that the entire V_{CORE} domain is powered off. Clocks are stopped and the real-time clock (RTC) can be preserved up by using LSE/LSI at 32 kHz/37 kHz.

RAM and register contents are lost except for the registers in the Standby circuitry. The number of enabled peripherals is even more limited than in Stop mode.

The device exits Standby mode upon reset, rising edge on one of the three WKUP pins, or if an RTC event occurs (if the RTC is ON).

*Note: When exiting Stop or Standby modes to enter the Run mode, STM32L1 MCUs go through a state where the MSI oscillator is used as clock source. This transition can have a significant impact on the global power consumption. For this reason, STM32CubeMX PCC introduces two transition steps: **WU_FROM_STOP** and **WU_FROM_STANDBY**. During these steps, the clock is automatically configured to MSI.*

D.1.2 STM32F4 series

STM32F4 microcontrollers feature a total of 5 power modes, including 4 low-power modes:

- **Run** mode

  This is the default mode at power-on or after a system reset. It offers the highest performance using HSE/HSI clock sources. The CPU can run at the maximum frequency depending on the selected power scale.

- **Sleep** mode

  Only the CPU is stopped. All peripherals continue to operate and can wake up the CPU when an interrupt/even occurs. The clock source is the clock that was set before entering Sleep mode.

- **Stop** mode

  This mode achieves a very low power consumption using the RC oscillator as clock source. All clocks in the 1.2 V domain are stopped as well as CPU and peripherals. PLL, HSI RC and HSE crystal oscillators are disabled. The content of registers and internal SRAM are kept.

  The voltage regulator can be put either in normal Main regulator mode (MR) or in Low-power regulator mode (LPR). Selecting the regulator in low-power regulator mode increases the wakeup time.

  The Flash memory can be put either in Stop mode to achieve a fast wakeup time or in Deep power-down to obtain a lower consumption with a slow wakeup time.

  The Stop mode features two sub-modes:

  – **Stop in Normal mode (default mode)**

    In this mode, the 1.2 V domain is preserved in nominal leakage mode and the minimum V12 voltage is 1.08 V.

  – **Stop in Under-drive mode**

    In this mode, the 1.2 V domain is preserved in reduced leakage mode and V12 voltage is less than 1.08 V. The regulator (in Main or Low-power mode) is in under-drive or low-voltage mode. The Flash memory must be in Deep-power-down mode. The wakeup time is about 100 µs higher than in normal mode.

- **Standby** mode

  This mode achieves very low power consumption with the RC oscillator as a clock source. The internal voltage regulator is switched off so that the entire 1.2 V domain is powered off: CPU and peripherals are stopped. The PLL, the HSI RC and the HSE crystal oscillators are disabled. SRAM and register contents are lost except for registers in the backup domain and the 4-byte backup SRAM when selected. Only RTC and LSE oscillator blocks are powered. The device exits Standby mode when an

external reset (NRST pin), an IWDG reset, a rising edge on the WKUP pin, or an RTC alarm/ wakeup/ tamper/time stamp event occurs.

- **V_{BAT}** operation

  It allows to significantly reduced power consumption compared to the Standby mode. This mode is available when the V_{BAT} pin powering the Backup domain is connected to an optional standby voltage supplied by a battery or by another source. The V_{BAT} domain is preserved (RTC registers, RTC backup register and backup SRAM) and RTC and LSE oscillator blocks powered. The main difference compared to the Standby mode is external interrupts and RTC alarm/events do not exit the device from V_{BAT} operation. Increasing V_{DD} to reach the minimum threshold does.

D.1.3 STM32L0 series

STM32L0 microcontrollers feature up to 8 power modes, including 7 low-power modes to achieve the best compromise between low-power consumption, short startup time and available wakeup sources:

- **Run mode**

  This mode offers the highest performance using HSE/HSI clock sources. The CPU can run up to 32 MHz and the voltage regulator is enabled.

- **Sleep mode**

  This mode uses HSE or HSI as system clock sources. The voltage regulator is enabled and only the CPU is stopped. All peripherals continue to operate and can wake up the CPU when an interrupt/event occurs.

- **Low-power run mode**

  This mode uses the internal regulator in low-power mode and the multispeed internal (MSI) RC oscillator set to the minimum clock frequency (131 kHz). In Low-power run mode, the clock frequency and the number of enabled peripherals are both limited.

- **Low-power sleep mode**

  This mode is achieved by entering Sleep mode with the internal voltage regulator in low-power mode. Both the clock frequency and the number of enabled peripherals are limited. Event or interrupt can revert the system to Run mode with regulator on.

- **Stop mode with RTC**

  The Stop mode achieves the lowest power consumption with, while retaining the RAM, register contents and real time clock. The voltage regulator is in low-power mode. LSE or LSI is still running. All clocks in the V_{CORE} domain are stopped, the PLL, MSI RC, HSE crystal and HSI RC oscillators are disabled.

  Some peripherals featuring wakeup capability can enable the HSI RC during Stop mode to detect their wakeup condition. The device can be woken up from Stop mode by any of the EXTI line, in 3.5 µs, and the processor can serve the interrupt or resume the code.

- **Stop mode without RTC**

  This mode is identical to “Stop mode with RTC “, except for the RTC clock which is stopped here.

- **Standby mode with RTC**

  The Standby mode achieves the lowest power consumption with the real time clock running. The internal voltage regulator is switched off so that the entire V_{CORE} domain

is powered off. The PLL, MSI RC, HSE crystal and HSI RC oscillators are also switched off. The LSE or LSI is still running.

After entering Standby mode, the RAM and register contents are lost except for registers in the Standby circuitry (wakeup logic, IWDG, RTC, LSI, LSE Crystal 32 KHz oscillator, RCC_CSR register).

The device exits Standby mode in 60 µs when an external reset (NRST pin), an IWDG reset, a rising edge on one of the three WKUP pins, RTC alarm (Alarm A or Alarm B),

RTC tamper event, RTC timestamp event or RTC Wakeup event occurs.

- **Standby mode without RTC**

  This mode is identical to Standby mode with RTC, except that the RTC, LSE and LSI clocks are stopped.

  The device exits Standby mode in 60 µs when an external reset (NRST pin) or a rising edge on one of the three WKUP pin occurs.

Note: The RTC, the IWDG, and the corresponding clock sources are not stopped automatically by entering Stop or Standby mode. The LCD is not stopped automatically by entering Stop mode.

D.2 Power consumption ranges

STM32 MCUs power consumption can be further optimized thanks to the dynamic voltage scaling feature: the main internal regulator output voltage V12 that supplies the logic (CPU, digital peripherals, SRAM and Flash memory) can be adjusted by software by selecting a power range (STM32L1 and STM32L0) or power scale (STM32 F4).

Power consumption range definitions are provided below (refer to STM32 MCU datasheets for full details).

D.2.1 STM32L1 series feature 3 V_{CORE} ranges

- High Performance **Range 1** (V_{DD} range limited to 2.0-3.6 V), with the CPU running at up to 32 MHz

  The voltage regulator outputs a 1.8 V voltage (typical) as long as the V_{DD} input voltage is above 2.0 V. Flash program and erase operations can be performed.
- Medium Performance **Range 2** (full V_{DD} range), with a maximum CPU frequency of 16 MHz

  At 1.5 V, the Flash memory is still functional but with medium read access time. Flash program and erase operations are still possible.
- Low Performance **Range 3** (full V_{DD} range), with a maximum CPU frequency limited to 4 MHz (generated only with the multispeed internal RC oscillator clock source)

  At 1.2 V, the Flash memory is still functional but with slow read access time. Flash Program and erase operations are no longer available.