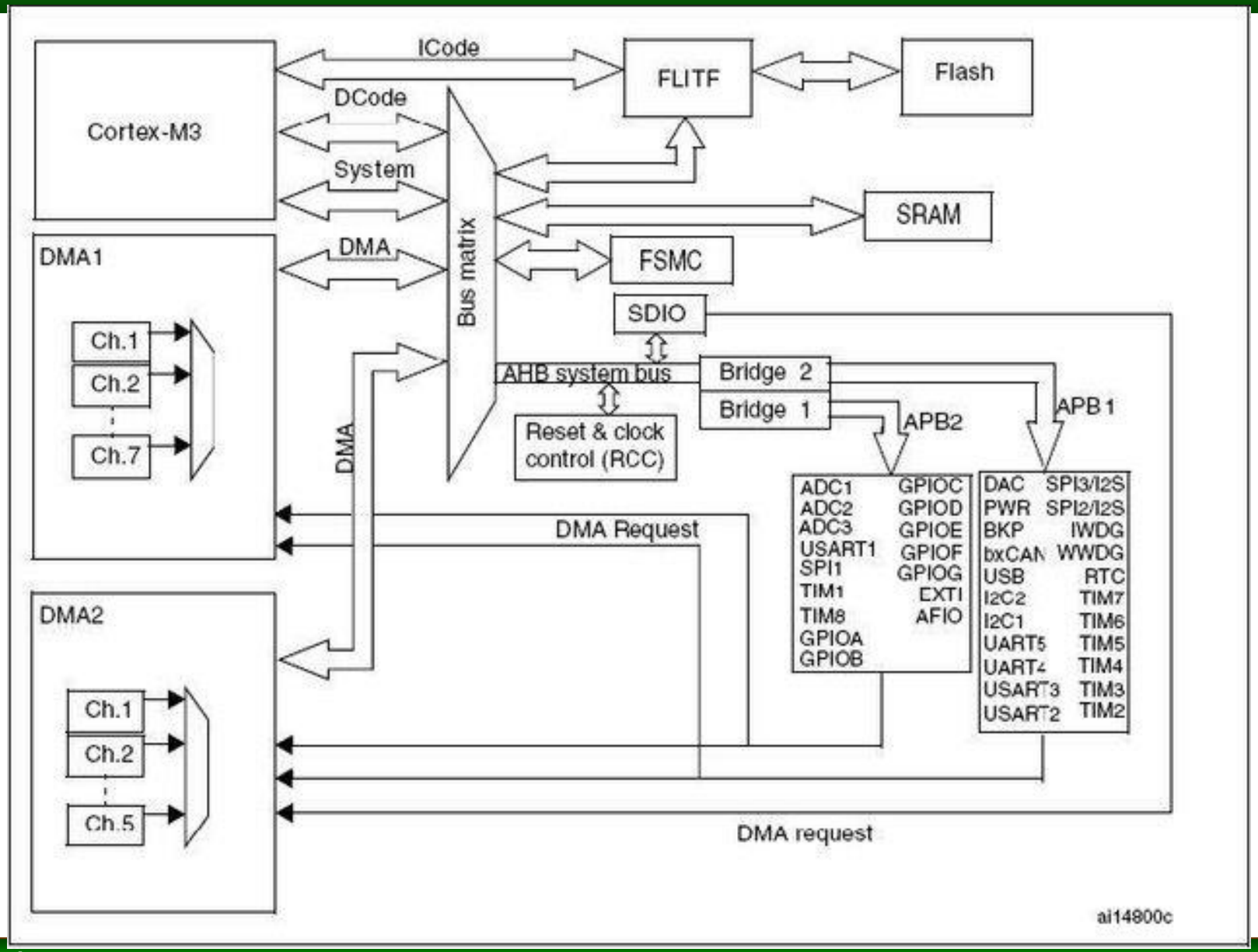
MCU memory map, memory mapped IO

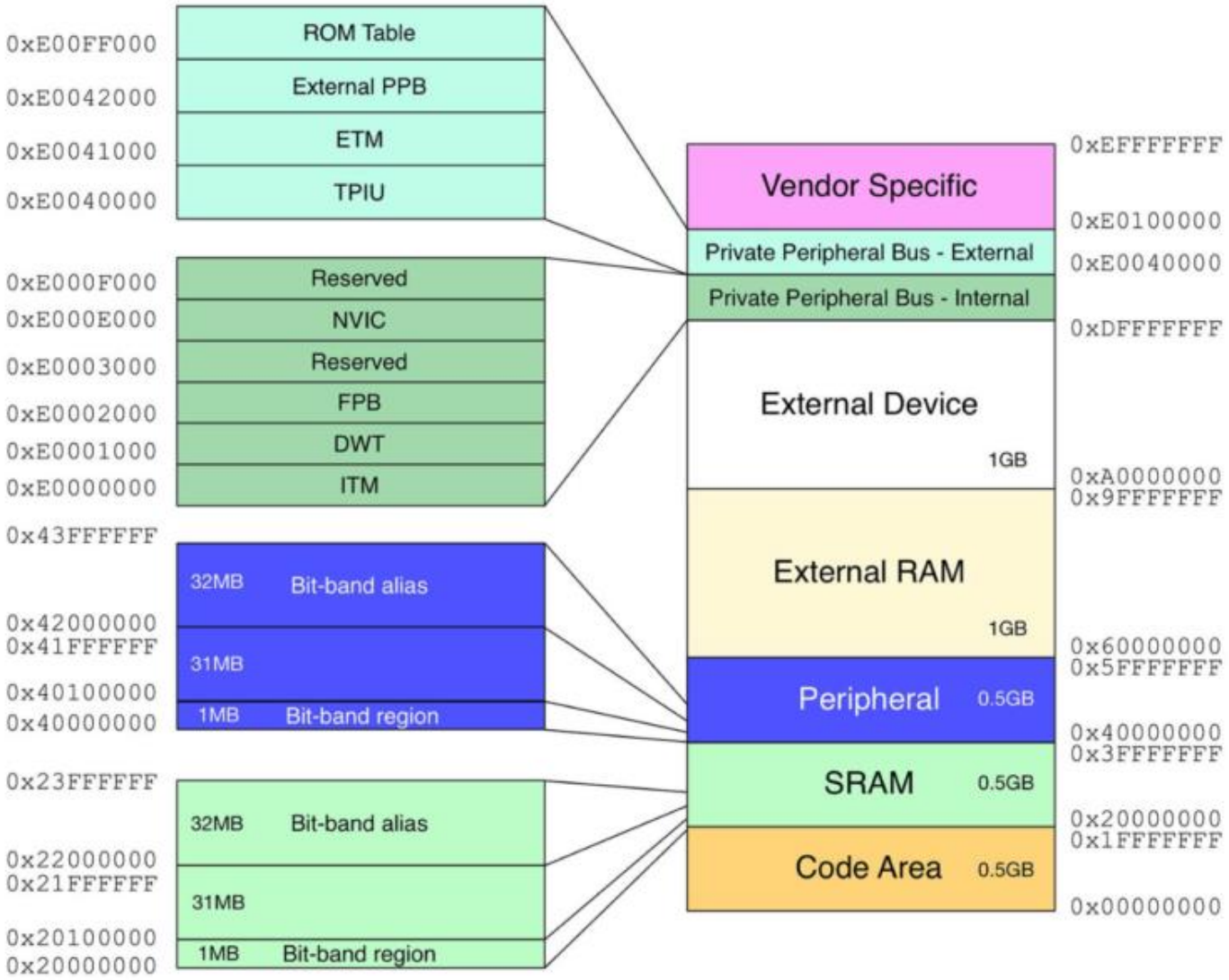
Topics

- STM32 Memory map
- Memory mapped IO
- CMSIS Core Library
- STM32 Embedded Software Libraries

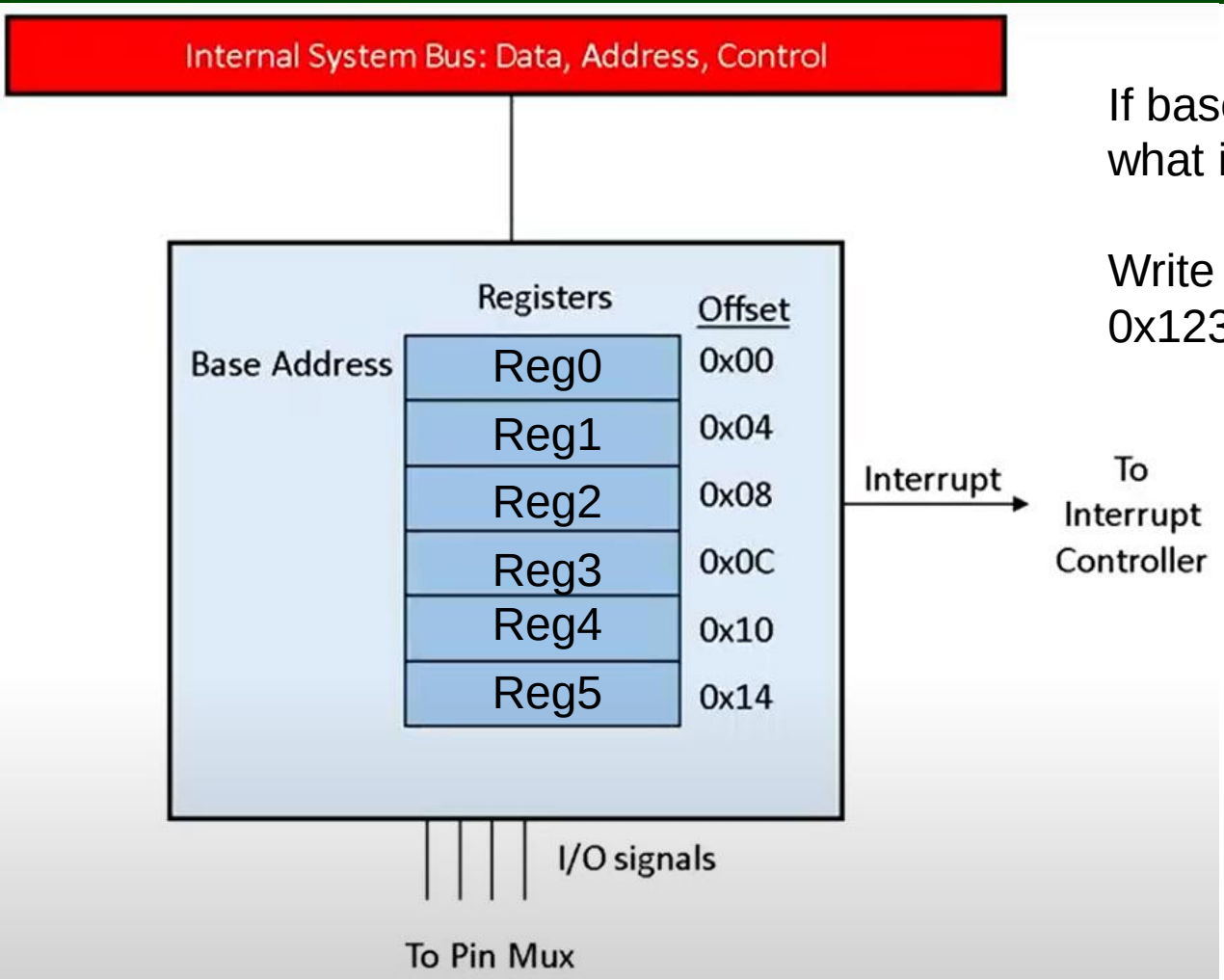
MCU Block Diagram



Memory map


0xE00FF000
ROM Table
0xE0042000
External PPB
0xE0041000
ETM
0xE0040000
TPIU
0xE000F000
Reserved
0xE000E000
NVIC
0xE0003000
Reserved
0xE0002000
FPB
0xE0001000
DWT
0xE0000000
ITM
0x43FFFFFF
32MB
Bit-band alias
0x42000000
0x41FFFFFF
31MB
0x40100000
0x40000000
1MB
Bit-band region
0x23FFFFFF
32MB
Bit-band alias
0x22000000
0x21FFFFFF
31MB
0x20100000
0x20000000
1MB
Bit-band region
0xEFFFFFFF
Vendor Specific
0xE0100000
Private Peripheral Bus - External
0xE0040000
Private Peripheral Bus - Internal
0xDFFFFFFF
External Device
1GB
0xA0000000
0x9FFFFFFF
External RAM
1GB
0x60000000
0x5FFFFFFF
Peripheral
0.5GB
0x40000000
0x3FFFFFFF
SRAM
0.5GB
0x20000000
0x1FFFFFFF
Code Area
0.5GB
0x00000000

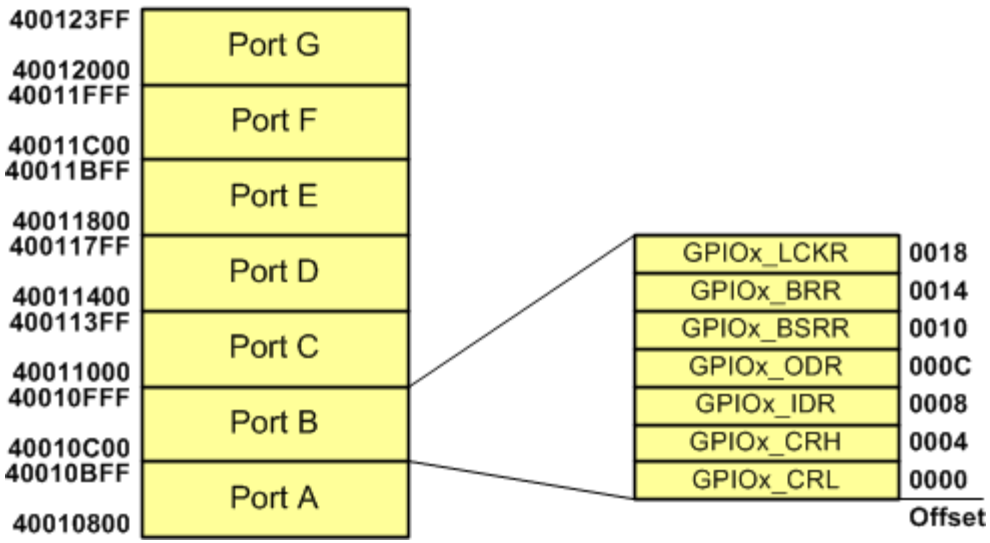
Memory mapped IO



If base address is 0xA0004000, what is the address of Reg5?

Write a C instruction to write 0x12345678 to Reg 5.

CMSIS IO Registers Access



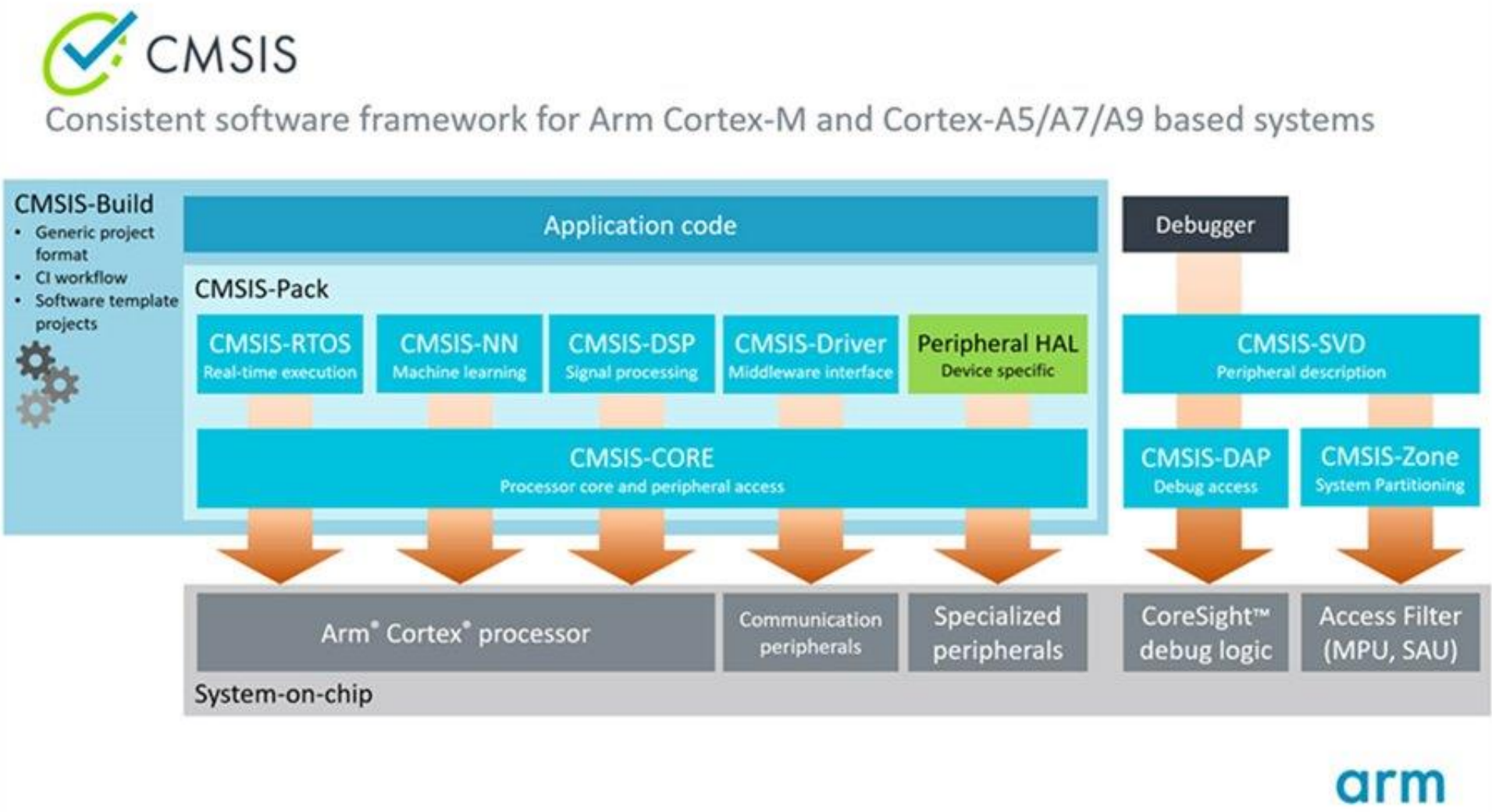

```
#define     __IO    volatile

typedef struct
{
  __IO uint32_t CRL;
  __IO uint32_t CRH;
  __IO uint32_t IDR;
  __IO uint32_t ODR;
  __IO uint32_t BSRR;
  __IO uint32_t BRR;
  __IO uint32_t LCKR;
} GPIO_TypeDef;
```

```
/*!< Peripheral memory map */
#define APB1PERIPH_BASE        PERIPH_BASE
#define APB2PERIPH_BASE        (PERIPH_BASE + 0x00010000UL)
#define AHBPERIPH_BASE         (PERIPH_BASE + 0x00020000UL)

#define GPIOA_BASE             (APB2PERIPH_BASE + 0x00000800UL)
#define GPIOB_BASE             (APB2PERIPH_BASE + 0x00000C00UL)
#define GPIOC_BASE             (APB2PERIPH_BASE + 0x00001000UL)
#define GPIOD BASE             (APB2PERIPH BASE + 0x00001400UL)

#define GPIOA                  ((GPIO_TypeDef *)GPIOA_BASE)
#define GPIOB                  ((GPIO_TypeDef *)GPIOB_BASE)
#define GPIOC                  ((GPIO_TypeDef *)GPIOC_BASE)
#define GPIOD                  ((GPIO_TypeDef *)GPIOD_BASE)
```

Driver Libraries

- CMSIS
  - Cortex Microcontroller Software Interface Standard
  - Support for creating reusable software components for ARM Cortex-M based systems
- Low level (LL)
- Hardware Abstraction Layer (HAL)

CMSIS Library



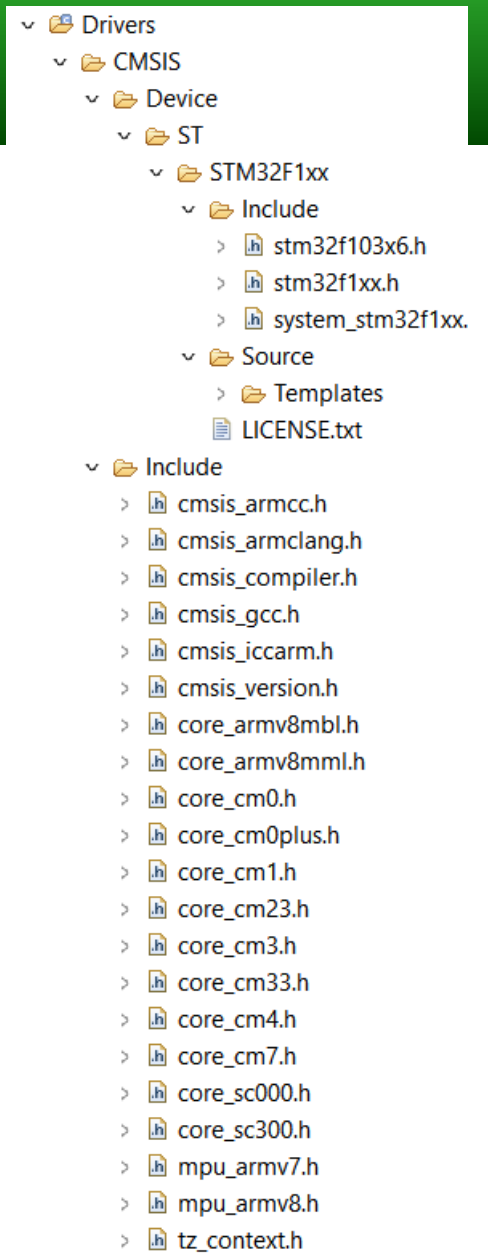
CMSIS Core

- CMSIS-Core (Cortex-M): define user access to the processor core and the device peripherals
  - **Hardware Abstraction Layer (HAL):** interface SysTick, NVIC, System Control Block registers, MPU registers, FPU registers, and core access functions.
  - **System exception names**: interface to system exceptions
  - **Methods to organize header files** that makes it easy to learn new Cortex-M microcontroller products and improve software portability.
  - **Methods for system initialization** to be used by each MCU vendor, e.g. SystemInit() function.
  - **Intrinsic functions** used to generate CPU instructions that are not supported by standard C functions.
  - A variable to determine the system clock frequency.

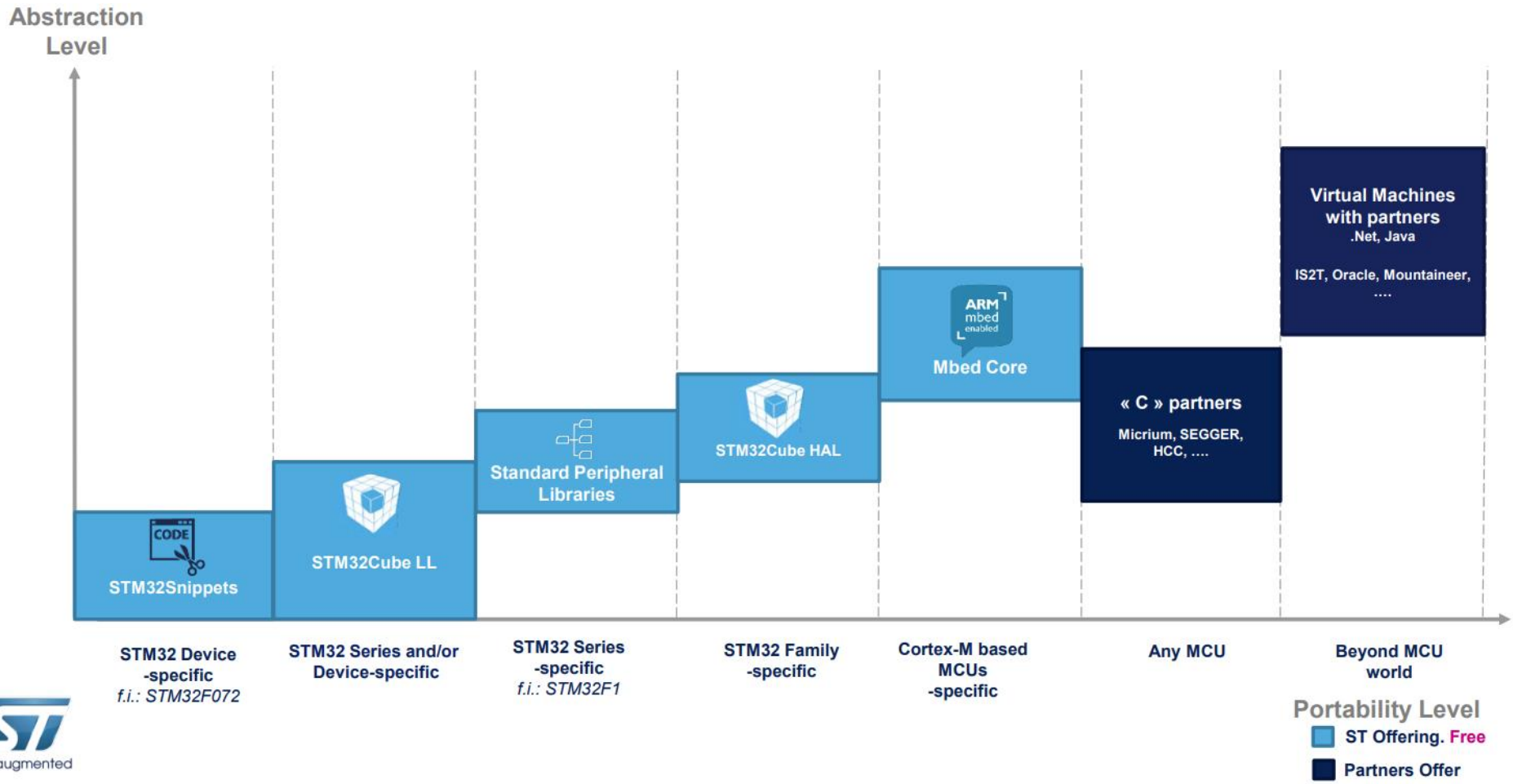
Reference: https://www.keil.com/pack/doc/CMSIS/Core/html/using_CMSIS.html

CMSIS Core files

File/Folder	Content
CMSIS\Documentation\Core	This documentation
CMSIS\Core\Include	CMSIS-Core (Cortex-M) header files (for example core_cm3.h, core_cmInstr.h, etc.)
Device	**Arm reference implementations** of Cortex-M devices
Device_Template_Vendor	**CMSIS-Core Device Templates** for extension by silicon vendors



STM32 Embedded Software Offer



STM32Snippets

What is it ?

A collection of code examples, directly based on STM32 peripheral registers, available in documentation and as softtware bundles

Targe Audience

Low level embedded system developers, typically coming from an 8 bit background, used to assembly or C with little abstraction

Features:

- Highly Optimized
- Register Level Access
- Small code expressions
- Closely follows the reference manual
- Debugging close to register level

Limitations:

- Specific to STM32 devices, not portable directly between series
- Not matching complex peripherals such as USB
- Lack of abstraction means developers must understand peripheral operation at register level
- Available (today) on STM32 L0 and F0 series

Standard Peripheral Libraries

What is it ?

Collection of C Libraries covering STM32 peripherals

Targe Audience

Embedded systems developers with procedural C background. All existing STM32 customer base prior to the STM32Cube launch, willing to keep same supporting technology for future projects, and same STM32 series

Features:

- Average optimization, fitting lots of situations
- No need for direct register manipulation
- 100% coverage of all peripherals
- Easier debugging of procedural code
- Extensions for complex middleware such as
- USB/TCP-IP/Graphics/Touch Sense

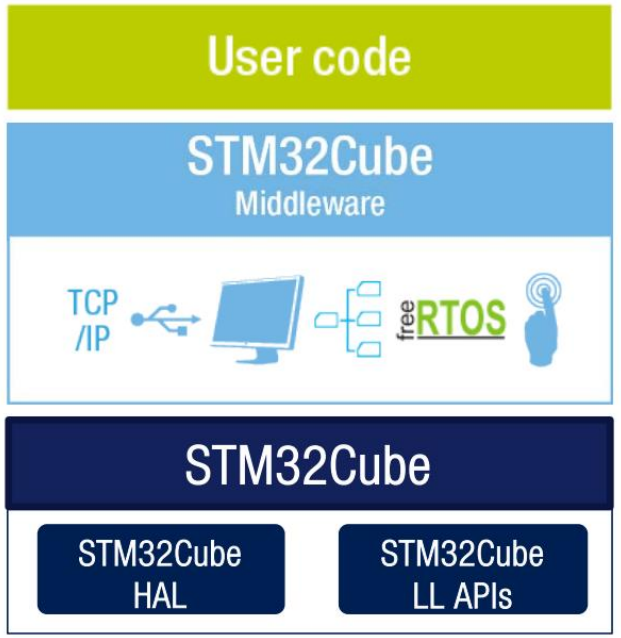
Limitations:

- Specific to certain STM32 series.
- No common HAL API prevents application portability between series
- Middleware libraries may not be unified for each series
- Doesn’t support forward STM32 series starting with STM32 L0, L4 and F7

STM32Cube

What is it ?

- Full featured packages with drivers, USB, TCP/IP, Graphics, File system and RTOS
- Set of common application programming interfaces, ensuring high portability inside whole STM32 family
- Set of APIs directly based on STM32 peripheral registers
- Set of initialization APIs functionally similar to the SPL block peripheral initialization functions



What is it ?

- Hardware Abstraction Layer (HAL) APIs: embedded system developers with a strong structured background. New customers looking for a fast way to evaluate STM32 and easy portability
- Low-Layer (LL) APIs: low level embedded system developers, typically coming from an 8-bit background, used to assembly or C with little abstraction. Stronger focus on customers migrating from the SPL environment .

HAL

Features:

- High level and functional abstraction
- Easy port from one series to another
- 100% coverage of all peripherals
- Integrates complex middleware such as USB/TCP-IP/Graphics/Touch Sense/RTOS
- Can work with STM32CubeMX to generate initialization code

Limitations:

- May be challenging to low level C programmers in the embedded space.
- Higher portability creates bigger software footprints or more time spent executing adaptation code

LL

Features:

- Highly Optimized
- Register Level Access
- Small code expressions
- Closely follows the reference manual
- Debugging close to register level
- Can work with STM32CubeMX to generate
- Initialization code for STM32L0/F0/F3/L4

- Peripheral block initialization APIs
  - ✓ Initialization, de-initialization and default initialization routines SPL-Like functionally speaking
  - ✓ More optimized than SPL, fitting lots of situations
  - ✓ No need for direct register manipulation
  - ✓ Easier debugging of procedural code

Limitations:

- Specific to STM32 devices, not portable directly between series
- Not matching complex peripherals such as USB
- Lack of abstraction for runtime means developers must understand peripheral operation at register level
- Peripheral block initialization APIs have the same limitations as the SPLs (except availability considerations)

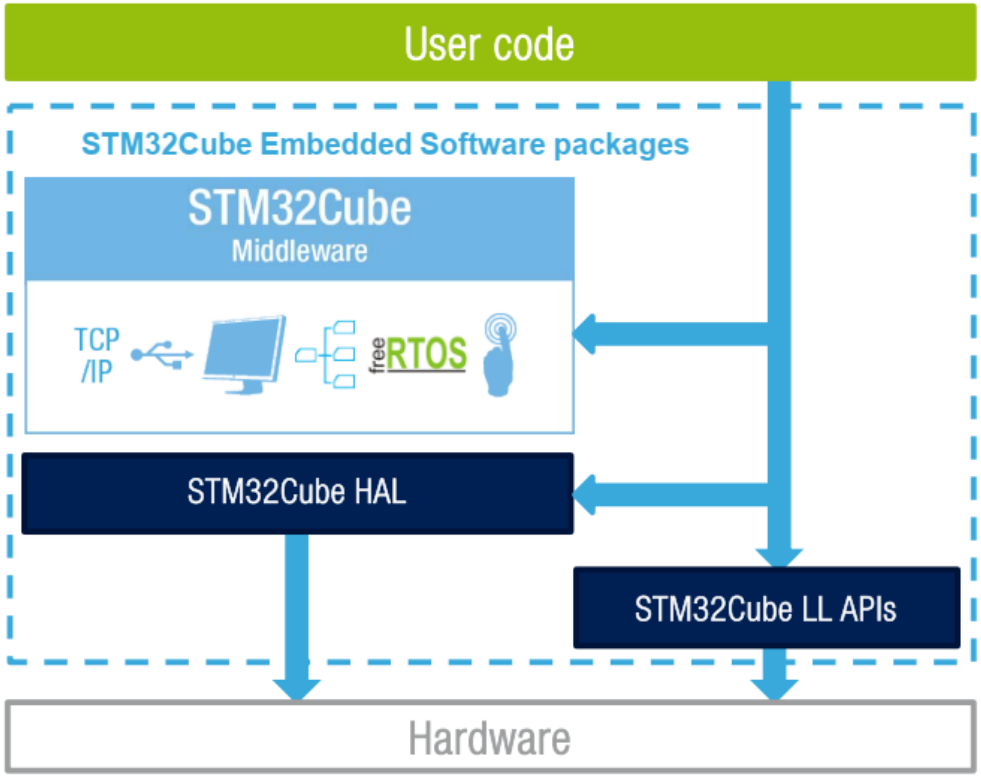
Possible concurrent usage of HAL and LL

Limitation:

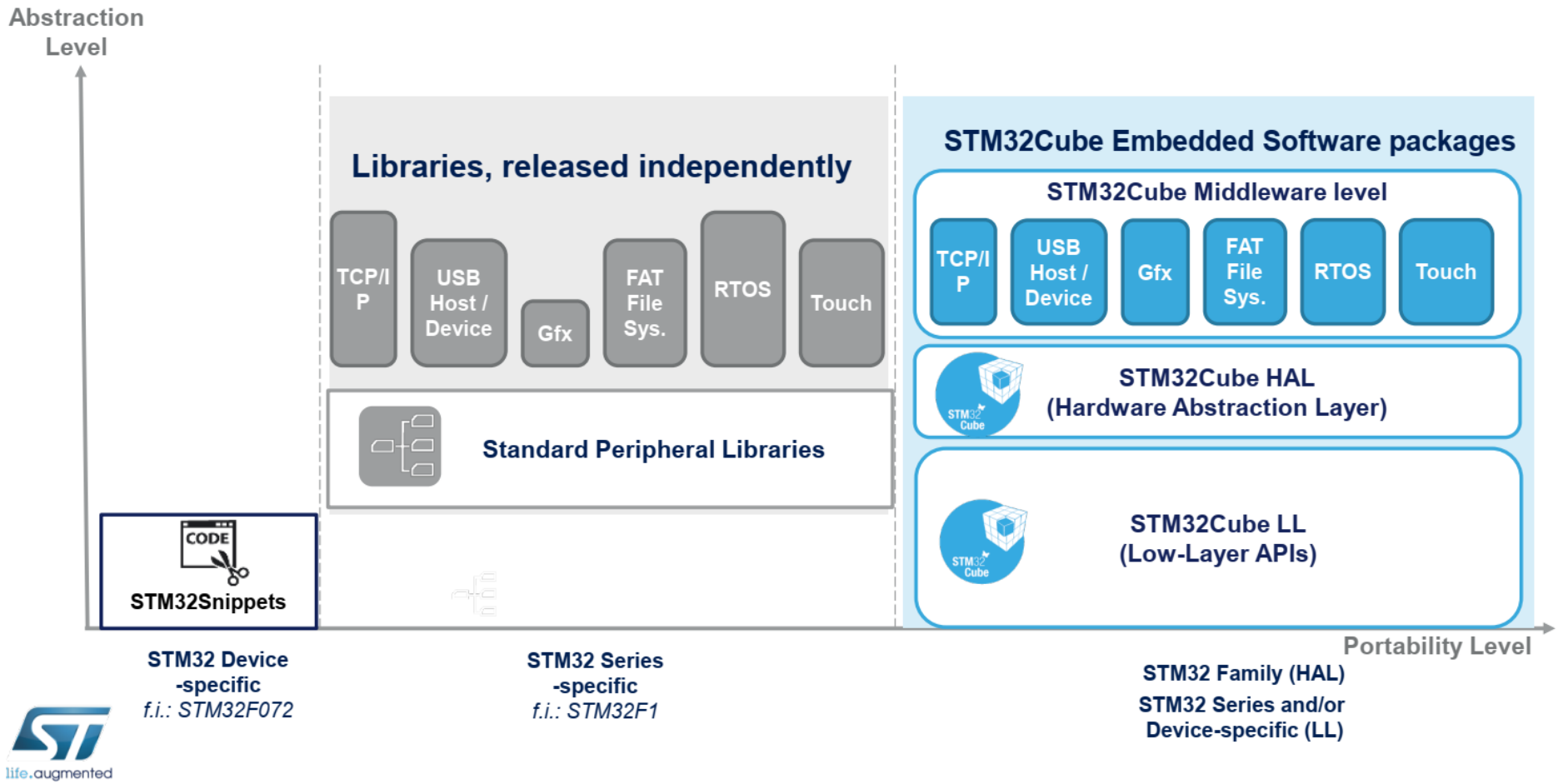
- LL cannot be used with HAL for the same peripheral instance. Impossible to run concurrent processes on the same IP using both APIs, but sequential use is allowed

Example of hybrid model:

- Simpler static peripheral initialization with HAL
- Optimized runtime peripheral handling with LL calls

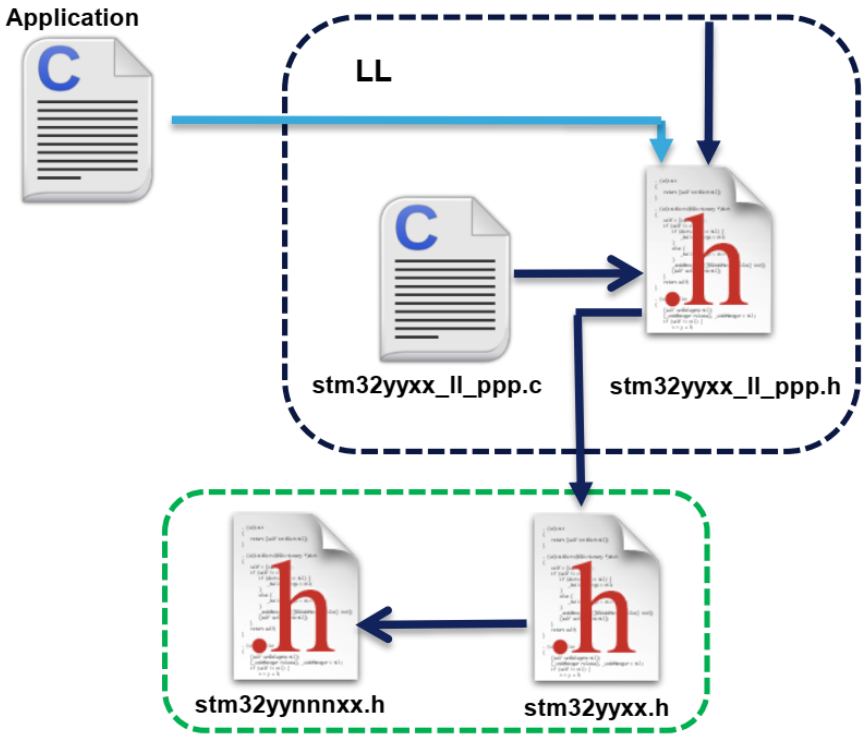


ST software packages



LL library - introduction

- Stm32yyxx_ll_ppp.h:
  - Unitary functions for direct register access
- Stm32yyxx_ll_ppp.h:
  - Init functions
  - Conceptual compatible with SPL libraries



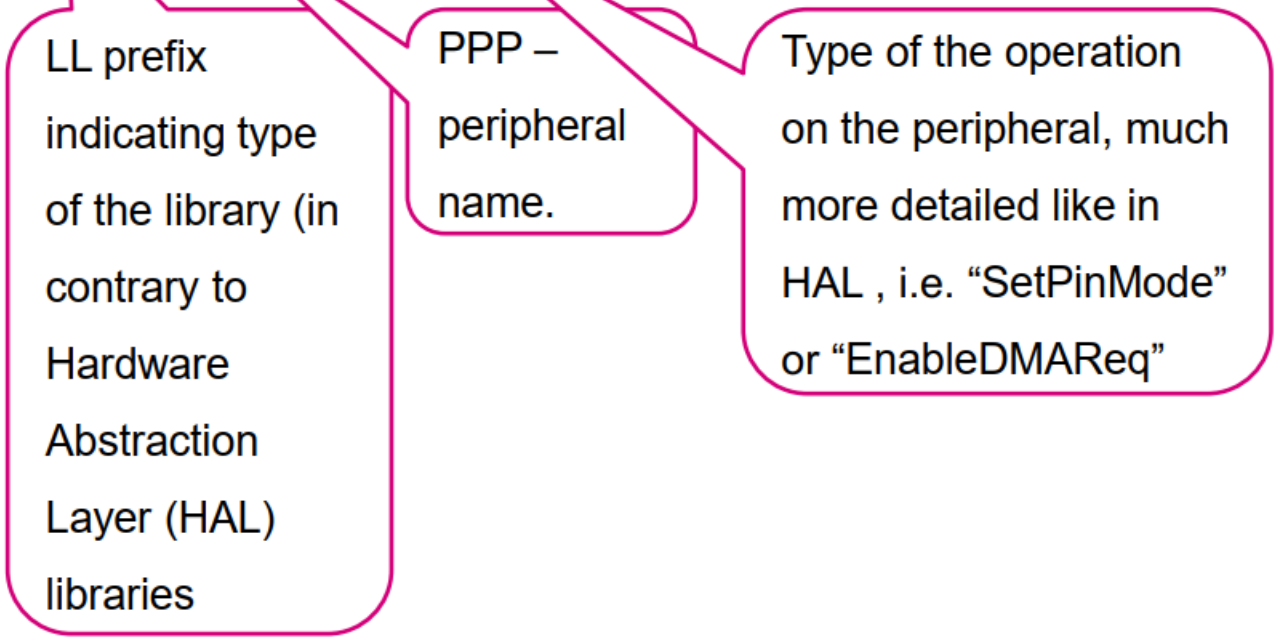
Role of some LL header file

- Most of **ppp** peripherals have their own pairs of **stm32xxxx_ll_ppp.c** and **.h** files

.h file	Serviced peripheral
stm32xxxx_ll_bus.h	APB1, APB2, IOP, AHB registers
stm32xxxx_ll_cortex.h	SYSTICK registers• Low power mode configuration (SCB register of Cortex-MCU) MPU API API to access to MCU info (CPUID register)
stm32xxxx_ll_system.h	Some of the FLASH features (accelerator, latency, power down modes) DBGCMU registers SYSCFG registers (including remap and EXTI)
stm32xxxx_ll_utils.h	Device electronic signature Timing functions PLL configuration functions

LL functions

return_value **LL_PPP_Operation** ()



LL_functions

- Low level: Basic register read and write

```
__STATIC_INLINE void LL_GPIO_WriteOutputPort(GPIO_TypeDef *GPIOx, uint32_t PortValue)
{
  WRITE_REG(GPIOx->ODR, PortValue);
}
```

- Middle level: one-shot operation API with some processing

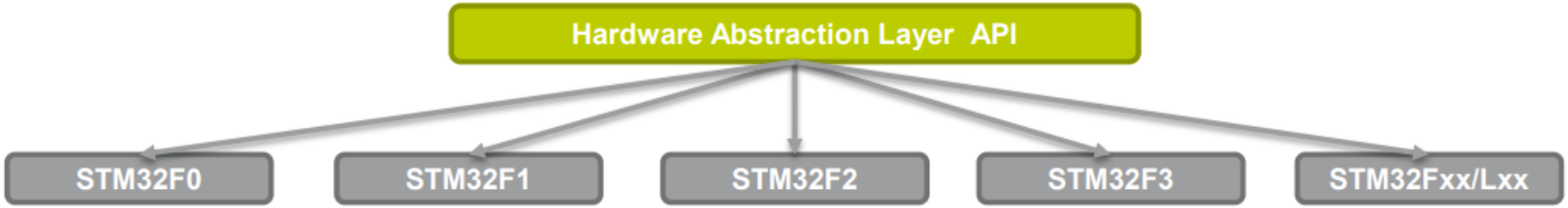
```
__STATIC_INLINE void LL_GPIO_SetPinSpeed(GPIO_TypeDef *GPIOx, uint32_t Pin, uint32_t  Speed)
{
  register uint32_t *pReg = (uint32_t *)((uint32_t)((uint32_t)(&GPIOx->CRL) + (Pin >> 24)));
  MODIFY_REG(*pReg, (GPIO_CRL_MODE0 << (POSITION_VAL(Pin) * 4U)),
             (Speed << (POSITION_VAL(Pin) * 4U)));
}
```

- High level: global configuration and initialization functions that cover full standalone operations on related peripheral registers

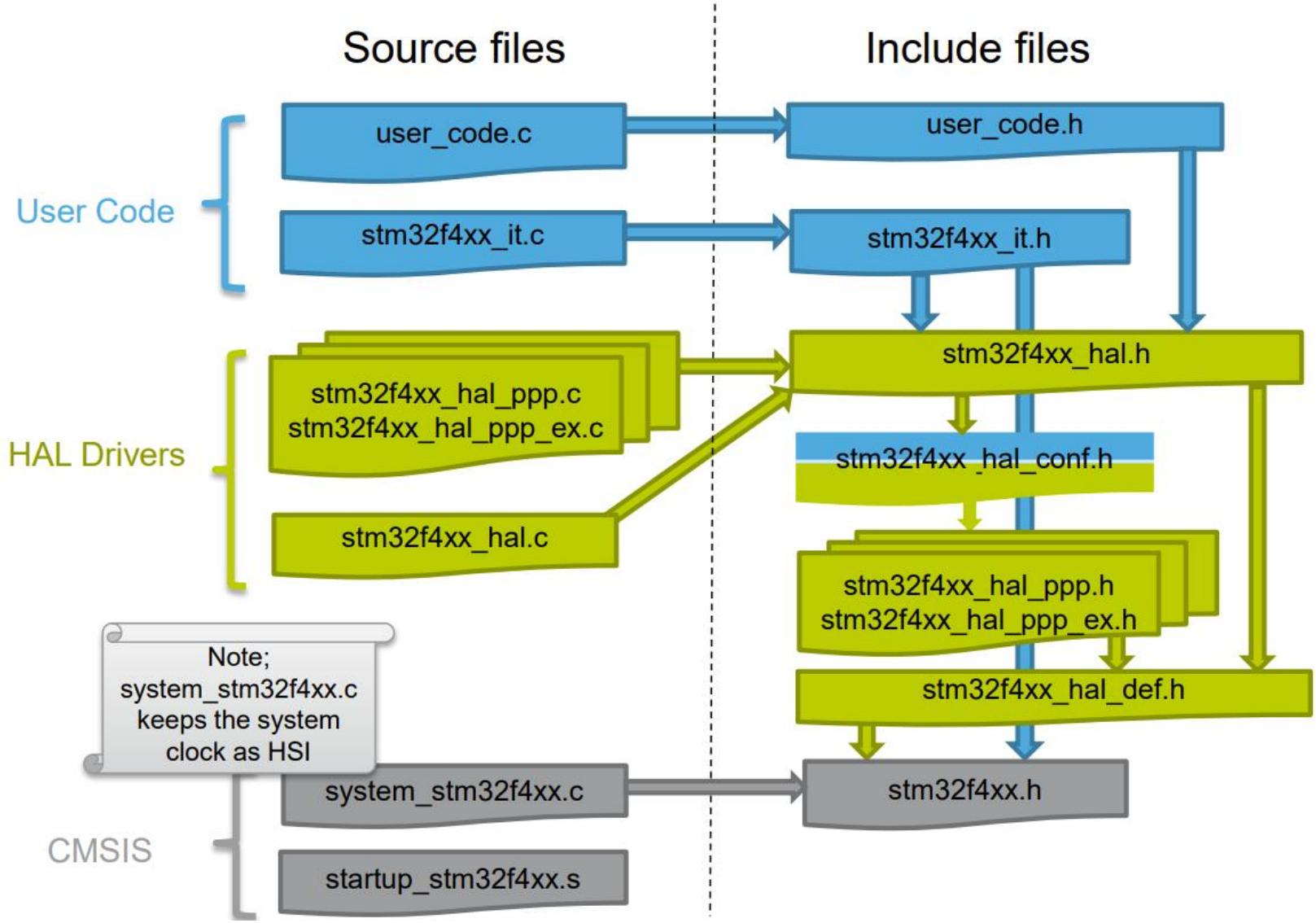
```
/**
  * @brief  Initialize GPIO registers according to the specified parameters in GPIO_InitStruct.
  * @param  GPIOx GPIO Port
  * @param  GPIO_InitStruct: pointer to a @ref LL_GPIO_InitTypeDef structure
  *         that contains the configuration information for the specified GPIO peripheral.
  * @retval An ErrorStatus enumeration value:
  *          - SUCCESS: GPIO registers are initialized according to GPIO_InitStruct content
  *          - ERROR:   Not applicable
  */
ErrorStatus LL_GPIO_Init(GPIO_TypeDef *GPIOx, LL_GPIO_InitTypeDef *GPIO_InitStruct)
```

Hardware abstraction layer (HAL)

- Set of APIs and to interact easily with the application upper layers
- Cross-family portable set of APIs
- Three API programming models: polling, interrupt and DMA
- All HAL APIs implement user-callback functions mechanism:
    - Peripheral Init/DeInit HAL APIs can call user-callback functions to perform peripheral system level Initialization/De-Initialization (clock, GPIOs, interrupt, DMA)
    - Peripherals interrupt events
    - Error events.



HAL Project struture



HAL features

- High level of abstraction
- API to perform complete procedure, e.g. ADC conversion
- Handle register access internally
- Use variable to hold the state of procedure, buffers

Choose drivers for implementation

- Use CMSIS for the core hardware, e.g. NVIC
- Use LL for register level programming of peripherals
- HAL can be used with care:
  - Make sure you understand the mechanism under the hood, e.g. interrupt handling, error handling, timing, etc

Opinion:

- Consider using LL libraries
- Provide the hardware abstraction at application / protocol stack module