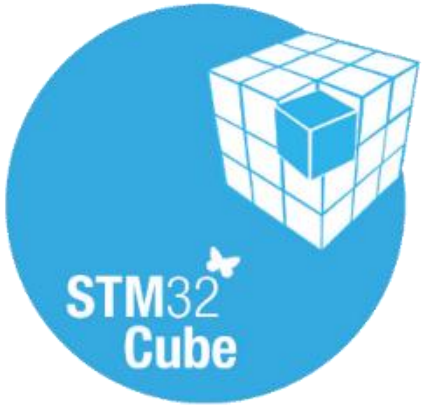
STM32 Ecosystem workshop

T.O.M.A.S Team

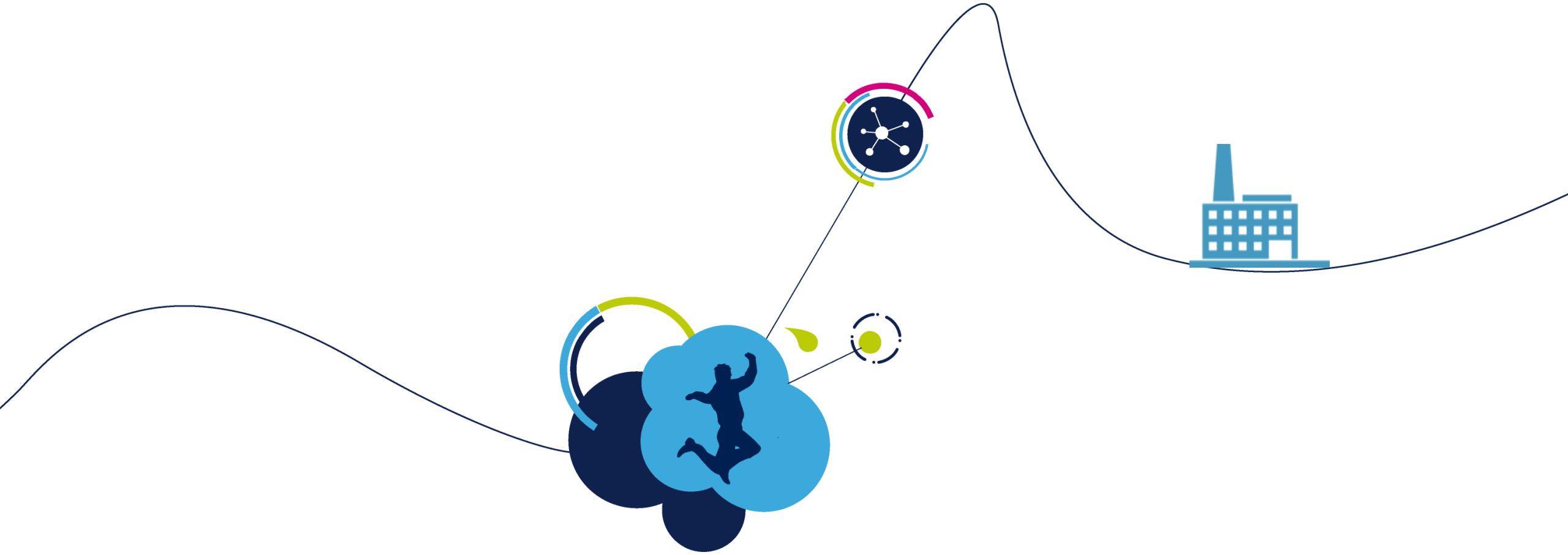
life.augmented



Now it is a right time for some theory – this time we will present basic information about a Low Layer Libraries bundled into Cube library packages

Goal of this part

- Gain knowledge about complete ST software offer for STM32 microcontrollers
- Gain knowledge about Low Layer Library concepts: unitary and init
- Practice Low Layer Library concept on previously generated HAL based project
- Gain knowledge about differences between HAL and LL concepts.



Low Layer Library concept and usage

life.augmented

HAL and LL libraries coexistence

general points

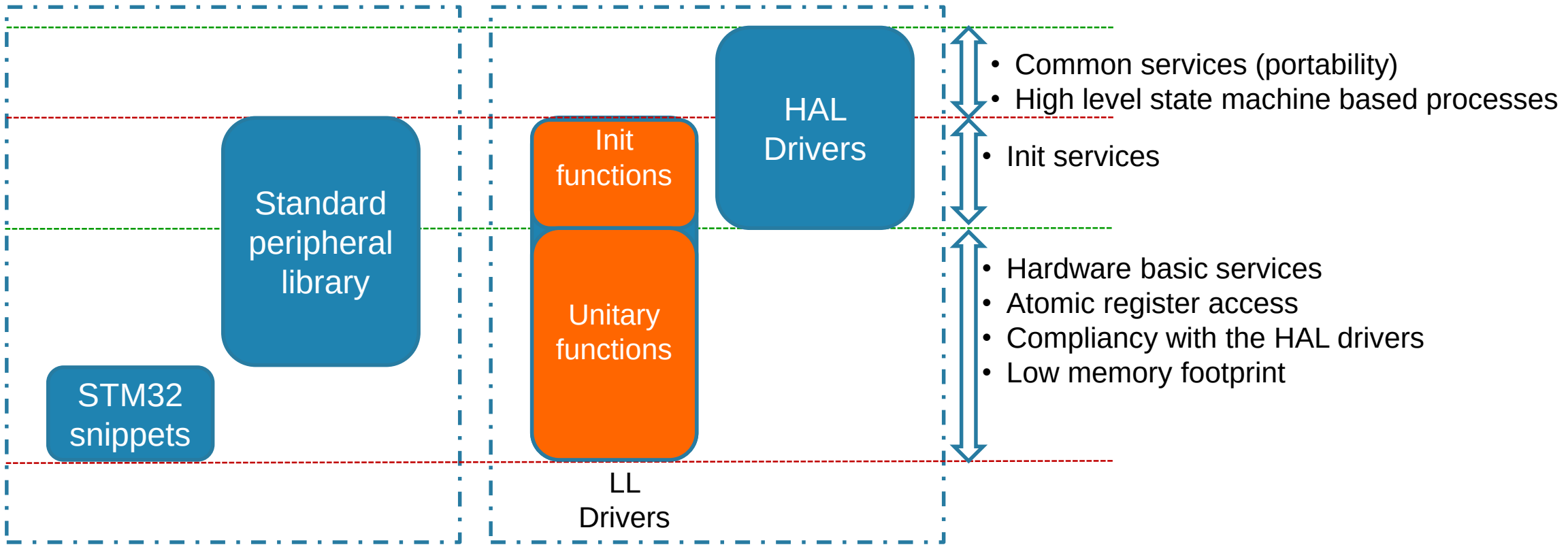
- The Low Layer (LL) drivers can be mixed without any constraints with all the HAL drivers not based on objects handle concept (RCC, Cortex, common HAL, Flash and GPIO)
- To mix the HAL with the LL, user has to be aware about some HAL concepts/constraints:
  - The LL drivers does not support the peripheral handle model
  - The LL drivers are intended to be used in an expert mode (need deep knowledge of hardware aspects).
  - The HAL drivers are just opposite to LL - they use a high abstraction level based on standalone processes, therefore a deep knowledge of the hardware is not mandatory anymore.
- Due to the different data structures used, LL can overwrite registers being mirrored in the HAL handles. Therefore the LL drivers cannot be automatically used with the HAL for the same peripheral instance: mainly can’t run concurrent process on the same IP using both APIs, however sequential use is allowed if done carefully.

HAL and LL libraries coexistence

mixing HAL operation APIs with the LL

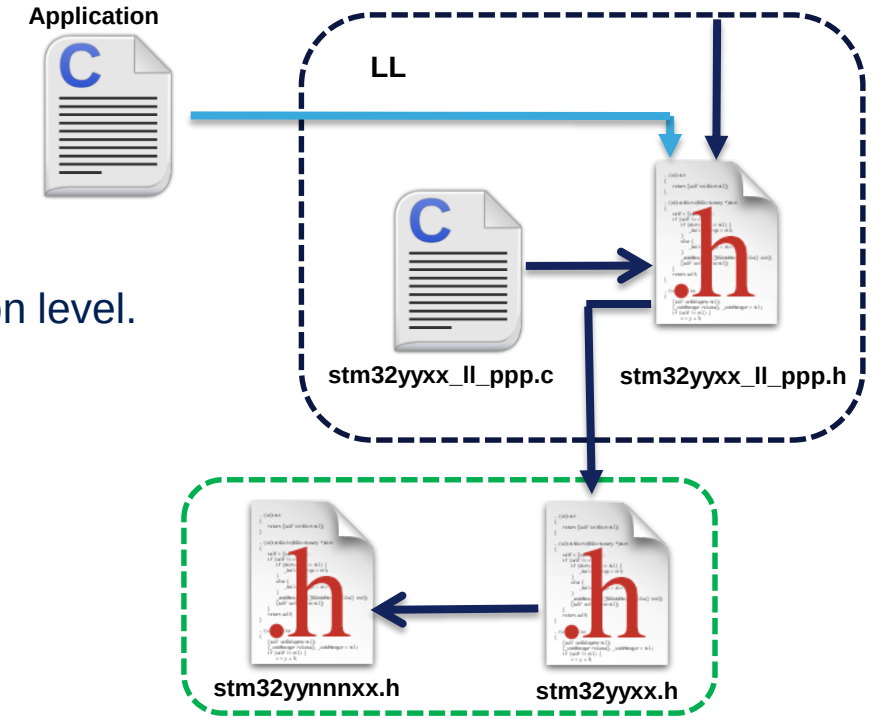
- The HAL I/O operations APIs are generally given in three models:
  - Blocking model (Polling)
  - Interrupt Model (IT)
  - DMA model
- If API of any of the HAL models gets replaced by the LL, it is not necessary to replace other models API.
- For DMA and IT API models - when customized by the LL, the HAL associated callbacks cannot be used for the addressed instance anymore.
- The processes customized should avoid dependencies with other processes ex: customize Transmit APIs with the LL and keep the Receive with the HAL

LL drivers scope



LL library - introduction

- HAL library brings high-level, functionally oriented and highly portable APIs masking product/IPs complexity to the end user
- Low Layer (LL) library offers low-level APIs operating at registers level, w/ better optimization but less portability and requiring deeper knowledge of the product/IPs specification
- LL library is offering the following services:
  - **Unitary functions for direct register access** (provided in stm32yyxx_ll_ppp.h files)
    - One-shot operations that can be used by HAL drivers or from application level.
    - Independent from HAL - can be used as standalone (w/o HAL drivers)
    - Full features coverage of supported IP
  - **Init functions** (provided in stm32yyxx_ll_ppp.c files)
    - Conceptually compatible with Standard Peripheral Library (SPL)



life.augmented

Role of some LL header files

- Most of **ppp** peripherals have their own pairs of **stm32xxxx_ll_ppp.c** and **.h** files
- Some peripherals and buses are grouped within **.c/.h** files described below:

.h file	Serviced peripherals
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

Low Layer library
naming convention

return_value **LL_PPP_Operation** ()

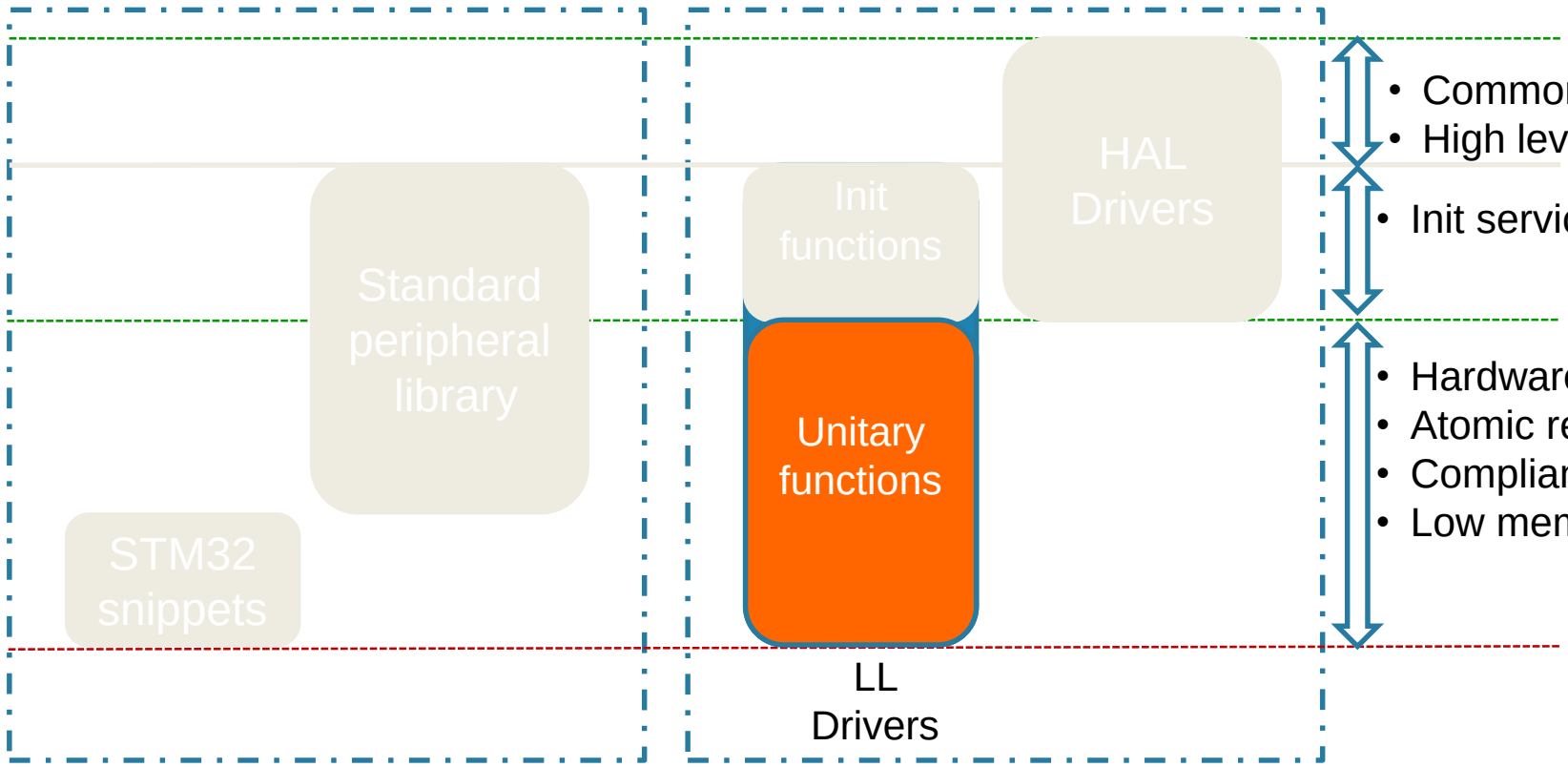
LL prefix indicating type of the library (in contrary to Hardware Abstraction Layer (HAL) libraries

PPP – peripheral name.

Type of the operation on the peripheral, much more detailed like in HAL , i.e. “SetPinMode” or “EnableDMAReq”

Hint: to find proper function for the peripheral, please type **`LL_PPP_`** and press **Ctrl+Space**

LL drivers scope
unitary functions



- Common services (portability)
- High level state machine based processes

- Init services

- Hardware basic services
- Atomic register access
- Compliancy with the HAL drivers
- Low memory footprint

Unitary LL drivers APIs

Each LL peripheral driver provides the following three APIs levels:

- ***Low level*** : Basic registers write and read

```
LL_PPP_WriteReg(I2C1,CR1,0x20001000);
LL_PPP_ReadReg (I2C1,CR1);
```

- ***Middle level*** : one-shot operation APIs (atomic) with elementary *LL_PPP_SetItem()* and *LL_PPP_Action()* functions: set directly one bit field in register for a single feature

```
LL_ADC_Enable();
LL_TIM_EnableCounter(TIM_TypeDef * TIMx);
LL_TIM_SetAutoReload(TIM_TypeDef * TIMx, uint32_t AutoReload);
```

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

```
LL_RCC_PLL_ConfigDomain_SYS(LL_RCC_PLLSOURCE_HSI, LL_RCC_PLLM_DIV_1, 10, LL_RCC_PLLR_DIV_2);
LL_DAC_SetTriggerSource(DAC1, LL_DAC_CHANNEL_1, LL_DAC_TRIG_EXT_TIM2_TRGO);
LL_TIM_OC_SetMode(TIM2, LL_TIM_CHANNEL_CH2, LL_TIM_OCMODE_TOGGLE);
```

stm32yyxx_ll_ppp.h

life.augmented

Unitary LL Typical Usage

The LL services have to be called *following the programming model of the reference manual document* by calling the *elementary* LL drivers services

```c
int main(void)
{
  /* Configure the system clock to 80 MHz */
  SystemClock_Config();

  /* -2- Configure IO in output push-pull mode to drive external LED */
  Configure_GPIO();
```

Ex : System Clock Configuration

```c
__STATIC_INLINE void SystemClock_Config(void)
{
  /* MSI configuration and activation */
  LL_FLASH_SetLatency(LL_FLASH_LATENCY_4);
  LL_RCC_MSI_SetRange(LL_RCC_MSIRANGE_6);
  LL_RCC_MSI_Enable();
  while(LL_RCC_MSI_IsReady() != 1)
  {
  };

  /* Main PLL configuration and activation */
  LL_RCC_PLL_ConfigDomain_SYS(LL_RCC_PLLSOURCE_MSI, LL_RCC_PLLM_DIV1, 40, LL_RCC_PLLR_DIV2);
  LL_RCC_PLL_Enable();
  LL_RCC_PLL_EnableDomain_SYS();
  while(LL_RCC_PLL_IsReady() != 1)
  {
  };

  /* Sysclk activation on the main PLL */
  LL_RCC_SetAHBPrescaler(LL_RCC_SYSCLK_DIV1);
  LL_RCC_SetSysClockSource(LL_RCC_SYS_CLKSOURCE_PLL);
  while(LL_RCC_GetSysClockSource() != LL_RCC_SYS_CLKSOURCE_STATUS_PLL)
  {
  };
```

Ex : Use GPIO to toggle LED continuously

```c
__STATIC_INLINE void Configure_GPIO(void)
{
  /* Enable the LED2 Clock */
  LED2_GPIO_CLK_ENABLE();
                    LL_AHB2_GRP1_EnableClock(LL_AHB2_GRP1_PERIPH_GPIOA);
  /* Configure IO in output push-pull mode to drive external LED2 */
  LL_GPIO_SetPinMode(LED2_GPIO_PORT, LED2_PIN, LL_GPIO_MODE_OUTPUT);
  LL_GPIO_SetPinOutputType(LED2_GPIO_PORT, LED2_PIN, LL_GPIO_OUTPUT_PUSHPULL);
  LL_GPIO_SetPinSpeed(LED2_GPIO_PORT, LED2_PIN, LL_GPIO_SPEED_LOW);
  LL_GPIO_SetPinPull(LED2_GPIO_PORT, LED2_PIN, LL_GPIO_PULL_NO);
}
```

DMA Init Example

using HAL DMA Driver


```
static void DMA_Config(void)
{
  __HAL_RCC_DMA2_CLK_ENABLE();

  DmaHandle.Init.Channel = DMA_CHANNEL;                     /* DMA_CHANNEL_0                    */
  DmaHandle.Init.Direction = DMA_MEMORY_TO_MEMORY;          /* M2M transfer mode                */
  DmaHandle.Init.PeriphInc = DMA_PINC_ENABLE;               /* Peripheral increment mode Enable */
  DmaHandle.Init.MemInc = DMA_MINC_ENABLE;                  /* Memory increment mode Enable     */
  DmaHandle.Init.PeriphDataAlignment = DMA_PDATAALIGN_WORD; /* Peripheral data alignment : Word */
  DmaHandle.Init.MemDataAlignment = DMA_MDATAALIGN_WORD;    /* memory data alignment : Word     */
  DmaHandle.Init.Mode = DMA_NORMAL;                         /* Normal DMA mode                  */
  DmaHandle.Init.Priority = DMA_PRIORITY_HIGH;              /* priority level : high            */
  DmaHandle.Init.FIFOMode = DMA_FIFOMODE_DISABLE;           /* FIFO mode disabled               */
  DmaHandle.Init.FIFOThreshold = DMA_FIFO_THRESHOLD_FULL;
  DmaHandle.Init.MemBurst = DMA_MBURST_SINGLE;              /* Memory burst                     */
  DmaHandle.Init.PeriphBurst = DMA_PBURST_SINGLE;           /* Peripheral burst                 */
  DmaHandle.Instance = DMA_INSTANCE;                        /* DMA Instance                */

  DmaHandle.XferCpltCallback  = TransferComplete;
  DmaHandle.XferErrorCallback = TransferError;

  /*##-5- Initialize the DMA stream ##########################################*/
  HAL_DMA_Init(&DmaHandle);

  HAL_NVIC_SetPriority(DMA_INSTANCE_IRQ, 0, 0);
  HAL_NVIC_EnableIRQ(DMA_INSTANCE_IRQ);

  HAL_DMA_Start_IT(&DmaHandle, (uint32_t)&aSRC_Const_Buffer, (uint32_t)&aDST_Buffer, BUFFER_SIZE);
}
```

DMA Init Example

using LL DMA Driver

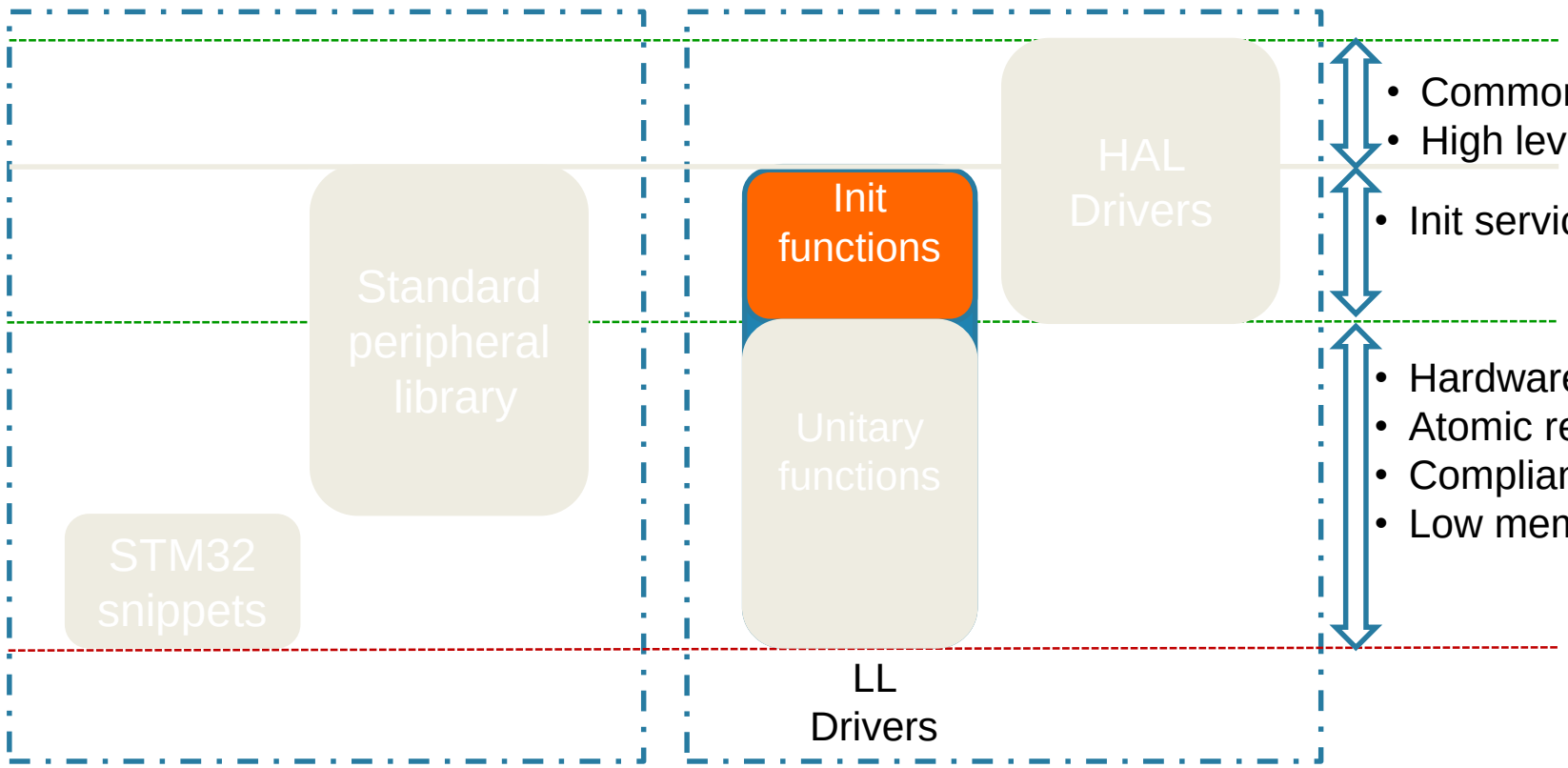

```
void Configure_DMA(void)
{
  /* (1) Enable the clock of DMA1 */
  LL_AHB1_GRP1_EnableClock(LL_AHB1_GRP1_PERIPH_DMA1);

  /* (2) Configure the DMA functional parameters */
  /* Configuration of the DMA parameters can be done using unitary functions or using the spec
  /* Unitary Functions */
  LL_DMA_SetDataTransferDirection(DMA1, LL_DMA_CHANNEL_1, LL_DMA_DIRECTION_MEMORY_TO_MEMORY);
  LL_DMA_SetChannelPriorityLevel(DMA1, LL_DMA_CHANNEL_1, LL_DMA_PRIORITY_HIGH);
  LL_DMA_SetMode(DMA1, LL_DMA_CHANNEL_1, LL_DMA_MODE_NORMAL);
  LL_DMA_SetPeriphIncMode(DMA1, LL_DMA_CHANNEL_1, LL_DMA_PERIPH_INCREMENT);
  LL_DMA_SetMemoryIncMode(DMA1, LL_DMA_CHANNEL_1, LL_DMA_MEMORY_INCREMENT);
  LL_DMA_SetPeriphSize(DMA1, LL_DMA_CHANNEL_1, LL_DMA_PDATAALIGN_WORD);
  LL_DMA_SetMemorySize(DMA1, LL_DMA_CHANNEL_1, LL_DMA_MDATAALIGN_WORD);
  LL_DMA_SetDataLength(DMA1, LL_DMA_CHANNEL_1, BUFFER_SIZE);
  LL_DMA_ConfigAddresses(DMA1, LL_DMA_CHANNEL_1,
                         (uint32_t)&aSRC_Const_Buffer,
                         (uint32_t)&aDST_Buffer,
                         LL_DMA_GetDataTransferDirection(DMA1, LL_DMA_CHANNEL_1));

  /* (3) Configure NVIC for DMA transfer complete/error interrupts */
  LL_DMA_EnableIT_TC(DMA1, LL_DMA_CHANNEL_1);
  LL_DMA_EnableIT_TE(DMA1, LL_DMA_CHANNEL_1);
  NVIC_SetPriority(DMA1_Channel1_IRQn, 0);
  NVIC_EnableIRQ(DMA1_Channel1_IRQn);
}
```

LL drivers scope

init functions



- Common services (portability)
- High level state machine based processes

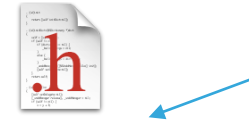
- Init services

- Hardware basic services
- Atomic register access
- Compliancy with the HAL drivers
- Low memory footprint

Init functions LL drivers APIs

- The init LL functions are based on the same concept as Standard Peripherals Library:
  - Series of data structures defined in corresponding header file (stm32l4xx_ll_gpio.h)
  - Initialization functions for:
    - data structures (setting all data fields to default values) and
    - peripherals or their functional part (copying data from structure fields to physical registers of selected peripheral).

  All these functions are defined in related source files (i.e. stm32l4xx_ll_gpio.c)



stm32yyxx_ll_gpio.h

stm32yyxx_ll_gpio.c

```
LL_GPIO_InitTypeDef GPIO_InitStruct;


void GPIO_LL_configuration(void)
{
  LL_AHB2_GRP1_EnableClock(LL_AHB2_GRP1_PERIPH_GPIOA); //enable clock to the GPIOA peripheral
  LL_GPIO_StructInit(&GPIO_InitStruct);
  GPIO_InitStruct.Pin= GPIO_PIN_1 | GPIO_PIN_4; //set pin 1 (ADC_IN6), 4 (DAC_OUT1)
  GPIO_InitStruct.Mode= LL_GPIO_MODE_ANALOG;    //set GPIO as analog mode
  GPIO_InitStruct.Pull= LL_GPIO_PULL_NO;        //no pull up or pull down
  LL_GPIO_Init(GPIOA,&GPIO_InitStruct);         //initialize GPIOA, pins 1 and 4
}
```

life.augmented

LL library - init functions architecture

- The init LL functions are considered as complementary services to the LL unitary ones and the HAL.

- There are no init LL functions for Core, PWR and system drivers, they are provided in header file only.
- Usage of init functions requires:
  - Definition of **USE_FULL_LL_DRIVER** in the user code
  - Inclusion all LL library **.c** files for each used **ppp** peripheral (i.e. **stm32l4xx_ll_gpio.c**)

What have we learnt?

- ✓Gain knowledge about complete ST software offer for STM32 microcontrollers
- ✓Gain knowledge about Low Layer Library concepts: unitary and init
- ❑Practice Low Layer Library concept on previously generated HAL based project
- ❑Gain knowledge about differences between HAL and LL concepts.

Further reading

More information can be found in the following documents:

- **UM1860** - Getting started with STM32CubeL4 for STM32L4 Series, available on the web:

  http://www.st.com/resource/en/user_manual/dm00157440.pdf

- **UM1884** - Description of STM32L4 HAL and Low-layer drivers, available on the web:

  http://www.st.com/resource/en/user_manual/dm00173145.pdf

- Doxygen based html manual: **STM32L486xx_User_Manual.chm**, available within STM32L4xx Cube library in the path:

  \STM32Cube_FW_L4_V1.5.0\Drivers\STM32L4xx_HAL_Driver\

Enjoy!

www.st.com/mcu

life.augmented