

# STM32 LL 库使用指南

---By Fengzi

熟悉 STM32 的都知道 ST 官方提供了非常方便好用的库函数供用户使用，多数人都使用过 STM32 标准外设库，STM32Cube 库(即 HAL 库)，这个 LL 库是什么鬼，却从来没听说过。

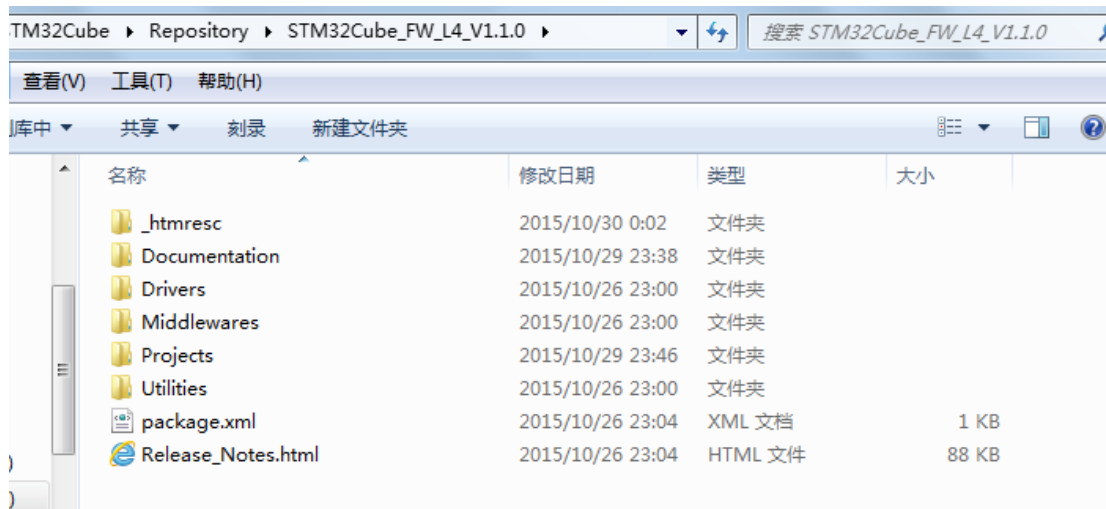
好吧，我承认这个名字是我自己 XJB 取的。。。。。。。

## 目录

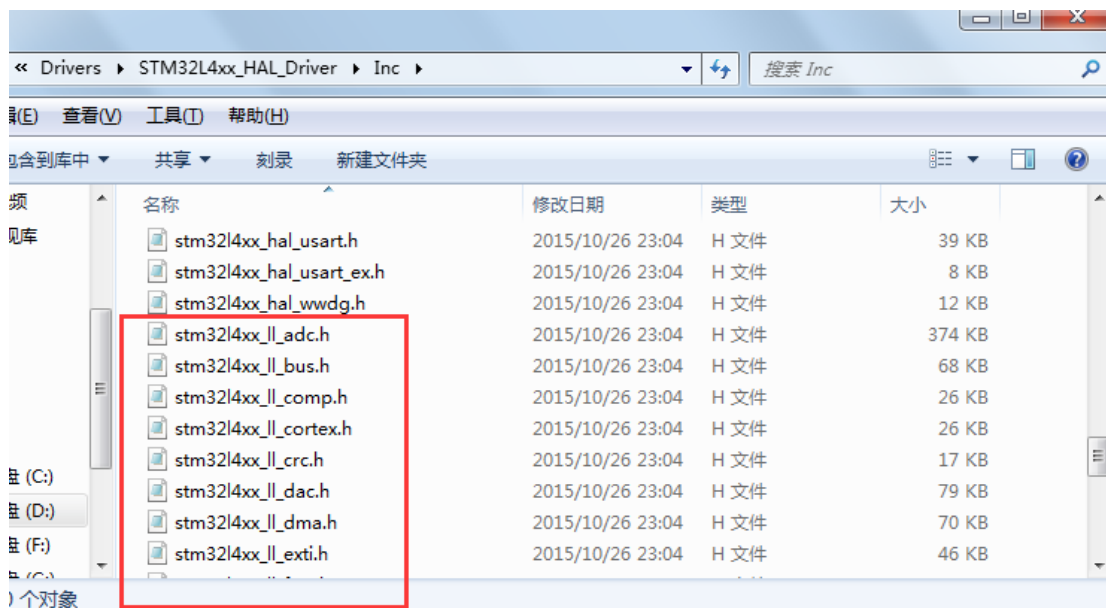
一、	初识 LL 库.....	1
二、	怎么使用 LL 库.....	3
三、	新建 STM32LL 库工程模板.....	5
四、	第一个程序——点亮 LED.....	8
五、	添加其他程序功能.....	10

## 一、 初识 LL 库

最近论坛发的 STM32L476RG Nucleo 开发板到手了，准备学习玩耍，当然第一步就是下载资料，于是我下载 STM32L4Cube 1.1.0 版本，打开逐个查看，

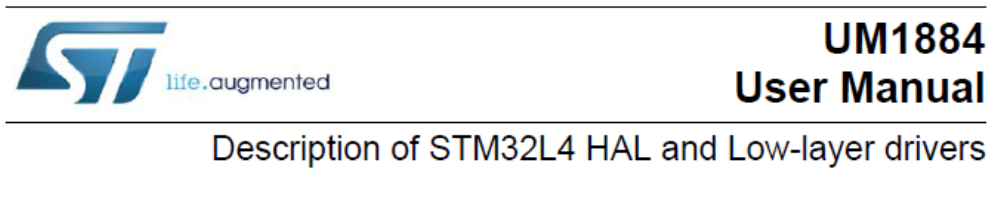


好像和以前一样的，没什么特别嘛，于是准备开始开发。。。等等，好像还真发现了有点不一样：



熟悉 HAL 库的都知道，该库的文件几乎都是以 `stm32xxx_hal_XXX.h/c` 命名的，为了和以前的标准库有个区分，上图中那些是什么鬼????

前辈说，遇到问题赶紧查手册，于是我果断打开 STM32L4Cube 库的说明手册(UM1884):



## Introduction

STMCube™ is an STMicroelectronics original initiative to ease developers life by reducing development efforts, time and cost. STM32Cube covers STM32 portfolio.

STM32Cube Version 1.x includes:

- The STM32CubeMX, a graphical software configuration tool that allows generating C initialization code using graphical wizards.
- A comprehensive embedded software platform, delivered per series (such as STM32CubeL4 for STM32L4 series)
  - The STM32Cube HAL, an STM32 abstraction layer embedded software ensuring maximized portability across STM32 portfolio. The HAL is available for all peripherals.
  - The Low Layer APIs (LL) offering a fast light-weight expert-oriented layer which is closer to the hardware than the HAL. The LL APIs are available only for a set of peripherals.
  - A consistent set of middleware components such as RTOS, USB, Graphics
  - All embedded software utilities coming with a full set of examples.

原来这个东西叫做 Low Layer APIs，作为英文渣渣表示实在不习惯洋里洋气的高大上名字，于是擅自把他叫做【STM32LL 库】了(不服的你咬我啊)。

从这里看好像是说这个东东比 HAL 库更接近硬件，到底什么鬼，还不清楚。但是以前好像没见过这个东西啊，就算是 STM32L4Cube 的 1.0.0 版本中都没有。看看 Cube 发行历史:

**Main Changes**

- **Add Low Layer drivers under Drivers\STM32L4xx\_HAL\_Driver**
  - Low Layer drivers allow performance and memory footprint optimization
    - Low Layer drivers APIs provide register level programming: they require deep knowledge of peripherals described in STM32L4x6 Reference Manual
    - Low Layer drivers are available for: ADC, COMP, Cortex, CRC, DAC, DMA, EXTI, GPIO, I2C, IWDG, LPTIM, LPUART, OPAMP, PWR, RCC, RNG, RTC, SPI, SWPMI, TIM, USART and WWDG peripherals and additional Low Level Bus, System and Utilities APIs.
    - Low Layer drivers APIs are implemented as static inline function in new *Inc/stm32l4xx\_ll\_ppp.h* files for PPP peripherals, there is no configuration file and each *stm32l4xx\_ll\_ppp.h* file must be included in user code.
    - Refer to [UM1860](#) for Low Layer presentation and [UM1884](#) for API list

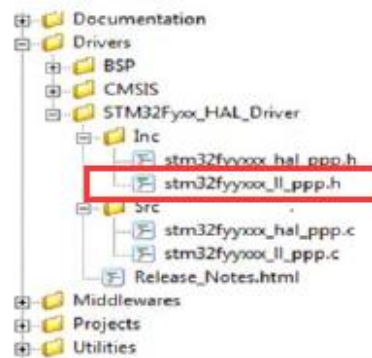
原来 LL 库是在 1.1.0 版本才加上的，大概意思就是：

1. LL APIs 是寄存器级的编程，嗯，也就是说我们常说的直接操作寄存器吧。
2. LL APIs 适用于 xxx 等一大堆外设
3. LL APIs 函数全部定义为 `static inline` 函数，放在对应的头文件中，用户使用需要包含相关头文件

4. 参考这两个文档

看看 LL 库文件在 Cube 库中的位置，有 20 多个文件，全部以 `stm32l4xx_ll_XXX.h` 命名：

STM32Cube\_FW\_L4\_V1.1.0\Drivers\STM32L4xx\_HAL\_Driver\Inc



STM32L4 是面向低功耗市场的，同时不失高性能，功耗和性能往往是两个矛盾的东西，ST 在硬件设计上想了各种办法来实现兼顾低功耗高性能(例如各种低功耗模式，LP 外设等)，而在软件层面，程序也讲求效率，LL 库全是直接操作寄存器，直接操作寄存器往往效率较高，而且函数定义为内联函数，调用函数时不是堆栈调用，而是直接把函数的代码嵌入到调用的地方，利于提高代码相率，我想这也是 ST 在 STM32L4 系列中推出这个直接操作寄存器的 LL 库的原因之一吧。

## 二、 怎么使用 LL 库

先看看手册里怎么说的，它有什么特点：

### 3 Overview of Low Layer drivers

The Low Layer (LL) drivers are designed to offer a fast light-weight expert-oriented layer which is closer to the hardware than the HAL. Contrary to the HAL, LL APIs are not provided for peripherals where optimized access is not a key feature, or those requiring heavy software configuration and/or complex upper-level stack (such as FSMC, USB or SDMMC).

LL drivers feature:

- A set of inline functions for direct and atomic register access
- Full independence from HAL since they can be used in standalone mode (without HAL drivers) or in mixed mode (with HAL drivers)
- Full coverage of the supported peripheral features.

The Low Layer drivers provide hardware services based on the available features of the STM32 peripherals. These services reflect exactly the hardware capabilities and provide one-shot operations that must be called following the programming model described in the microcontroller line reference manual. As a result, the LL services do not implement any processing and do not require any additional memory resources to save their states, counter or data pointers: all the operations are performed by changing the associated peripheral registers content.

英文渣渣就不翻译了，反正大概就是说

LL 库更接近硬件层，对需要复杂上层协议栈的外设不适用，直接操作寄存器等等一大堆，到这里，可以看到它的使用方法：

1. 独立使用，该库完全独立实现，可以完全抛开 HAL 库，只用 LL 库编程完成。
2. 混合使用，和 HAL 库结合使用。

本人就常在编程的时候库函数和寄存器操作混合，所以觉得混合使用应该是不错的方式。

最后一段还说到该库不需要额外的内存资源来存储程序状态，数据指针等东西，所有的操作都通过直接修改外设的寄存器来完成。

下面是手册中对各个 LL 文件的描述：

#### 3.1 Low Layer files

The Low Layer drivers are built around one single header file per peripheral plus four header files for some System and Cortex related features.

Table 16: LL driver files

File	Description
<i>stm32l4xx_ll_bus.h</i>	This is the h-source file for core bus control and peripheral clock activation and deactivation <i>Example: LL_AHB2_GRP1_EnableClock</i>
<i>stm32l4xx_ll_ppp.h</i>	This is the h-source file of the PPP Low Layer driver. The Low Layer PPP driver is a standalone module. To use it, the application must include the <i>stm32l4xx_ll_ppp.h</i> .
<i>stm32YYxx_ll_cortex.h</i>	CortexM related register operation APIs including the SysTick, Low-power (LL_SYSTICK_XXXXX, LL_LPM_XXXXX "Low Power Mode" ...)
<i>stm32YYxx_ll_utils.h</i>	Utilities related operations (LL_GetLLVersion, LL_mDelay, LL_Init1msTick, LL_GetUID_Word, ...)
<i>stm32YYxx_ll_system.h</i>	System related operations (LL_IsActiveSystickCounterFlag, LL_SetSystickClkSource, LL_GetSystickClkSource, LL_FLASH_XXX ...)

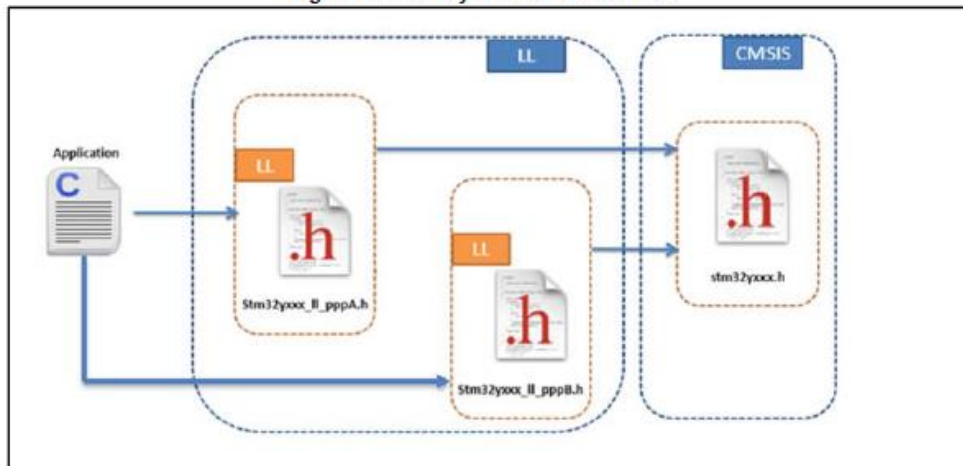
就是讲 LL 库由 5 部分组成：每个外设对应一个头文件组成一部分，以及系统相关的 bus, cortex, utils, system 四个部分。

前面提到，要使用 LL 库，需要包含对应头文件，各头文件之间有如下关系：

In general, Low Layer drivers include only the STM32 CMSIS device file.

```
#include "stm32yyxx.h"
```

Figure 9: Low Layer driver CMSIS files



Application files have to include only the used Low Layer drivers header files.

看来，我们编程的时候只需要#include 某外设的头文件，就可以使用 LL 库了，但是同时，系统启动文件，初始化文件等一系列不能少，具体讲就是：

stm32l4xx.h

stm32l476xx.h

system\_stm32l4xx.h

system\_stm32l4xx.c

startup\_stm32l476xx.s

这几个文件，这在标准库，Cube 库都不曾变过的铁律。

### 三、新建 STM32LL 库工程模板

要开发首先要搭建开发环境，也就是简历所谓的工程模板。由于独立使用 LL 库的情况下不能使用 Cube 来建立工程，所以需要手动建立，这里的建工程方法和以前使用标准库建工程模板比较类似：

#### 1. 准备相关文件：

新建一个文件夹用来放所有的文件

把需要用到相关文件全部复制过来：

首先是上一节说到的几个必不可少的文件

stm32l4xx.h,stm32l476xx.h,

system\_stm32l4xx.h,system\_stm32l4xx.c,

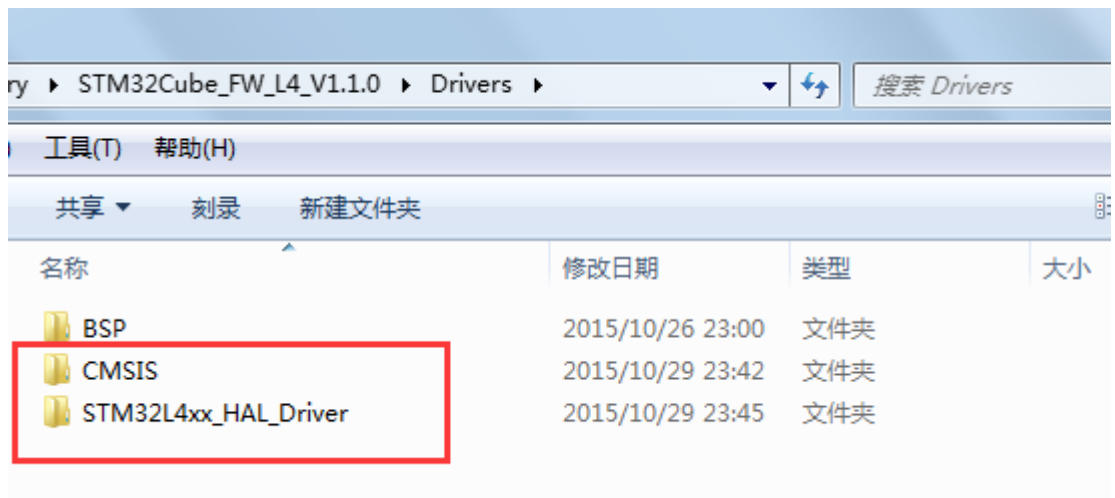
startup\_stm32l476xx.s

这些文件都在 STM32Cube\_FW\_L4\_V1.1.0\Drivers\CMSIS\Device\ST\STM32L4xx 文件夹下，为了方便查找，我保持了和库文件中相同的文件夹结构。如果闲麻烦，可以把 STM32Cube\_FW\_L4\_V1.1.0\Drivers\CMSIS 整个文件夹复制过来，该文件夹下的其他文件如 Include 文件夹里面的也需要用到。

然后把所有 LL 库的文件复制过来，

STM32Cube\_FW\_L4\_V1.1.0\Drivers\STM32L4xx\_HAL\_Driver\Inc 文件夹下面所有

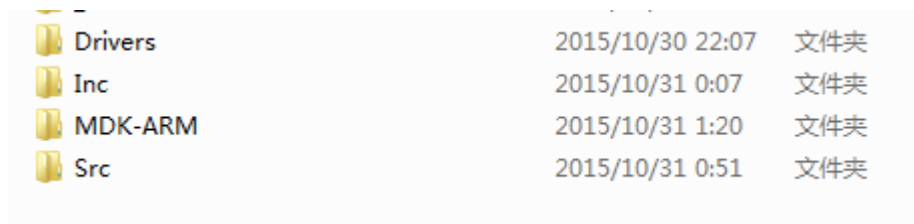
stm32l4xx\_ll\_\*.h 命名的文件，也可以整个文件夹复制。



其实主要就是这两个文件夹下面相关的一些文件复制过来就好。当然一股脑全部粘贴过去也没什么影响，就是工程大一点(henduo)。


最后建立 Inc, Src, MDK-ARM 文件夹放头文件、源文件和工程文件，, stm32l4xx\_it.h/.c 这两个文件其实也可以不要，因为我们可以把中断函数处理函数放在任何地方都可以，main.c/main.h 自己新建或者复制个模板。

所有文件，文件夹名字都可以随便取，这里只是为了保持和库、Cube 建立的工程文件结构保持一致，所以取这些名字。

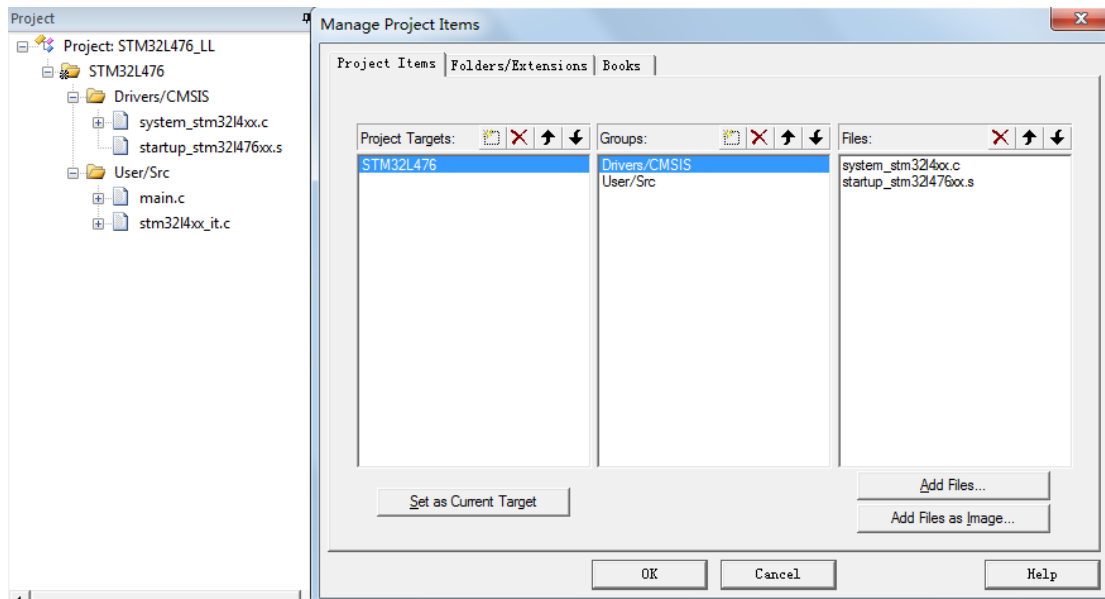


## 2. 好了，准备工作做好了，开始建工程

打开 Keil MDK，新建一个工程选择型号 STM32L4RG，在此之前需要确保已经安装了 Keil STM32L4 的 Pack，不然新建工程的时候找不到对应的型号。

 Keil.STM32L4xx\_DFP.1.0.0.pack

然后往工程里添加刚刚准备好的相关文件：

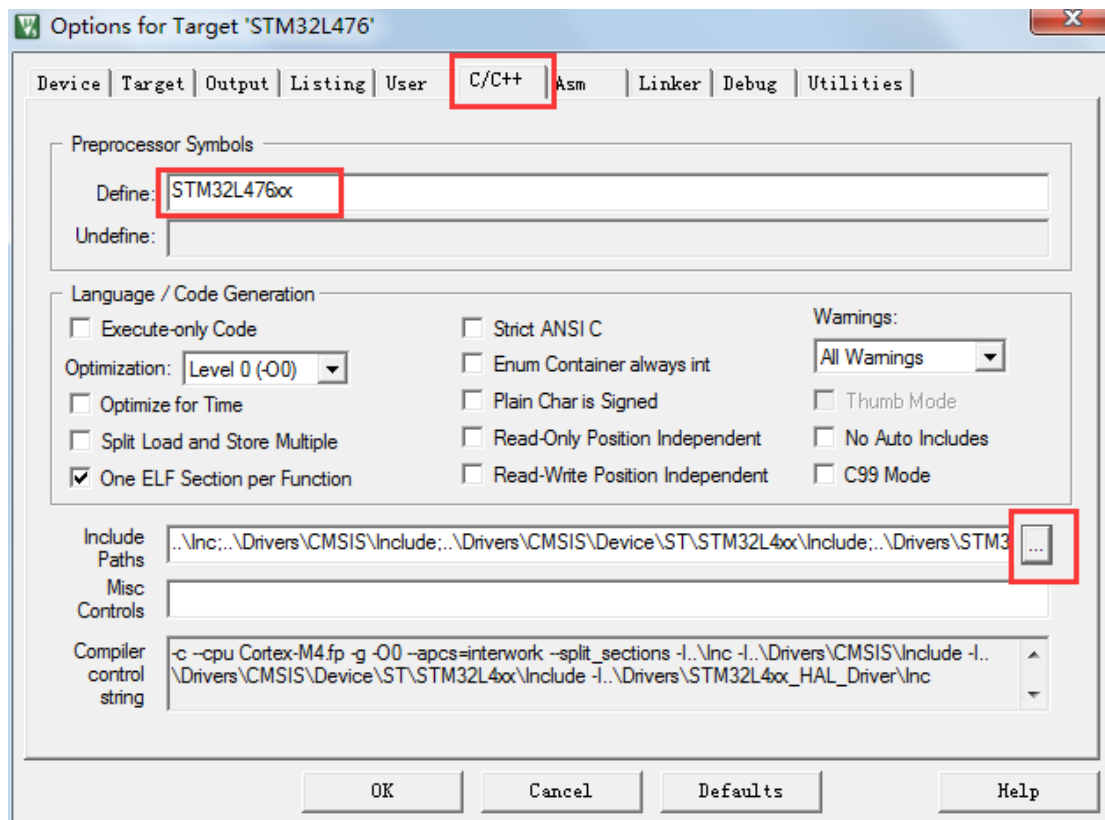


只需添加\*.c 和\*.s 文件，\*.h 文件设置头文件包含路径就行，也可以加到工程中方便查看。

接下来就是最重要的部分，工程设置：

有几个主要地方需要设置，

1. 定义使用的芯片型号 STM32L4xx



2. 设置文件包含路径

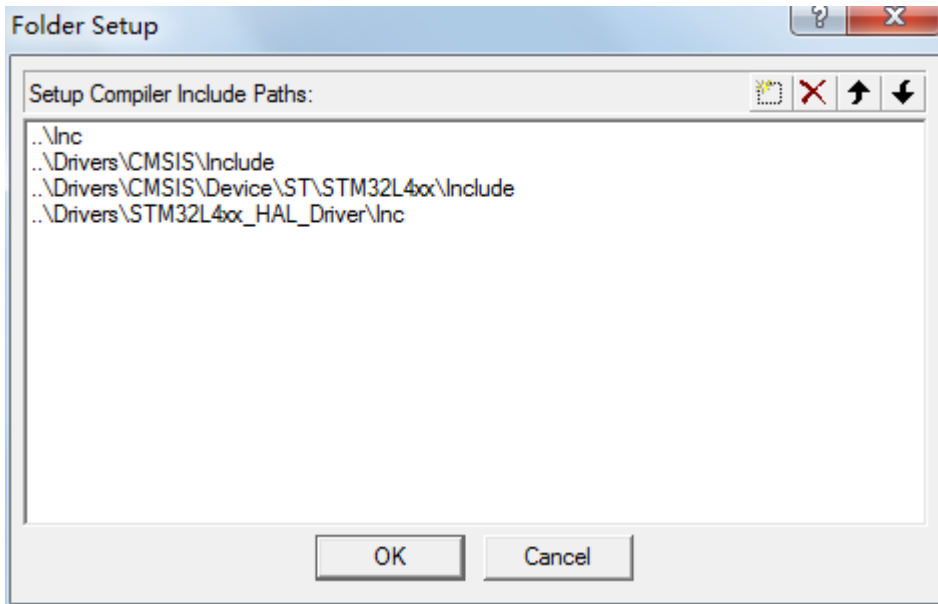
..\Inc

//自己写的头文件路径

```

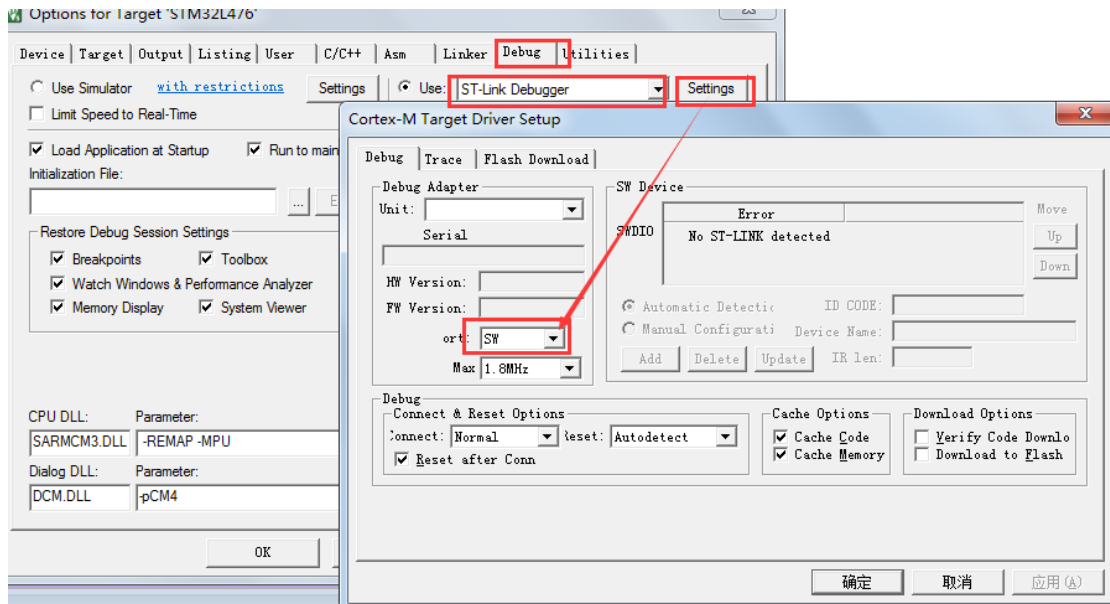
..\Drivers\CMSIS\Include //关于 Cortex 的一些头文件
..\Drivers\CMSIS\Device\ST\STM32L4xx\Include // stm32l4xx.h,stm32l476xx.h 等文件
..\Drivers\STM32L4xx_HAL_Driver\Inc //LL 库文件的路径

```



3. 设置所用调试仿真器，使用 Nucleo 板载 ST-Link

4.



差不多就这些了，如果芯片型号选择正确，pack 安装正确，其他的工程已经默认设置好，编译一下无错误无警告，下面就是真正的开发，写代码了。

## 四、 第一个程序——点亮 LED

首先点亮 LED 来验证下工程设置，代码编写是否正确。



## 1. 包含头文件

在 main.c 中

```
#include "main.h"
```

在 main.h 中

```
#include "stm32l4xx_ll_bus.h"
#include "stm32l4xx_ll_rcc.h"
#include "stm32l4xx_ll_system.h"
#include "stm32l4xx_ll_utils.h"
#include "stm32l4xx_ll_gpio.h"
```

包含需要使用的相关模块，要用那些就包含哪些，不知道用哪几个就全部包含进去。

## 2. 配置时钟

用这个 LL 库开发 STM32 跟用标准外设库很像，很类似，只不过封装层次更低。首先第一步要配置时钟，因为启动文件里没有默认配置，这和 STM32F1 早期的标准外设库一样，不过较高版本和后来的系列就可以不用了(如果使用默认配置的话)，启动文件里默认配置了时钟。怎么配置这里就不细说了，看代码应该都能懂。

```
__STATIC_INLINE void SystemClock_Config(void)
{
    LL_FLASH_SetLatency(LL_FLASH_LATENCY_4);
    LL_RCC_HSI_Enable();
    while(LL_RCC_HSI_IsReady() != 1);
    LL_RCC_PLL_ConfigDomain_SYS(LL_RCC_PLLSOURCE_HSI,LL_RCC_PLLM_DIV_1,10,LL_RCC_PLLR_
    DIV_2);
    LL_RCC_PLL_Enable();
    LL_RCC_PLL_EnableDomain_SYS();
    while(LL_RCC_PLL_IsReady() != 1);
    LL_RCC_SetAHBPrescaler(LL_RCC_SYSCLK_DIV_1);
    LL_RCC_SetSysClkSource(LL_RCC_SYS_CLKSOURCE_PLL);
    while(LL_RCC_GetSysClkSource() != LL_RCC_SYS_CLKSOURCE_STATUS_PLL);
    LL_RCC_SetAPB1Prescaler(LL_RCC_APB1_DIV_1);
    LL_RCC_SetAPB2Prescaler(LL_RCC_APB2_DIV_1);
    LL_Init1msTick(80000000);
    LL_SetSystemCoreClock(80000000);
}
```

函数声明为 \_\_STATIC\_INLINE(即 static inline)是因为该库所有函数都是 \_\_STATIC\_INLINE，前面提到这样做不需要额外的内存，因此这样最是为了保持这一点。

## 3. 将 LED 对应引脚 PA5 配置为推挽输出模式

```
__STATIC_INLINE void Configure_GPIO(void)
{
    LL_AHB2_GRP1_EnableClock(LL_AHB2_GRP1_PERIPH_GPIOA);

    LL_GPIO_SetPinMode(GPIOA, LL_GPIO_PIN_5, LL_GPIO_MODE_OUTPUT);
```

```

LL_GPIO_SetPinOutputType(GPIOA, LL_GPIO_PIN_5, LL_GPIO_OUTPUT_PUSHPULL);
LL_GPIO_SetPinSpeed(GPIOA, LL_GPIO_PIN_5, LL_GPIO_SPEED_LOW);
LL_GPIO_SetPinPull(GPIOA, LL_GPIO_PIN_5, LL_GPIO_PULL_NO);
}

```

#### 4. PA5 输出高电平，点亮 LED

```
LL_GPIO_SetOutputPin(GPIOA, LL_GPIO_PIN_5);
```

- 最后需要在主函数中依次调用上述函数，编译下载运行即可看到 Nucleo 上的绿色 LED 被成功点亮了。

```
int main(void)
```

```

{
    SystemClock_Config();
    Configure_GPIO();
    LL_GPIO_SetOutputPin(GPIOA, LL_GPIO_PIN_5);
    while (1)
    { ;
    }
}

```

*/\*碎碎念：虽然已经不知道这是第几百还是第几千次做点灯实验，依然还是不厌其烦啊，LED 不愧是被称为单片机届的“Hello World!”\*/*

## 五、 添加其他程序功能

会了第一个程序，其他的就按照手册讲的写就行了，再举个简单的例子，利用按键中断模式来控制 LED 的闪烁频率。

只说下大概步奏，不详细说明：将按键连接的引脚配置为中断输入模式，开启中断，配置中断线，写中断服务函数来处理中断。

```

__STATIC_INLINE void Configure_EXTI(void)
{
    LL_AHB2_GRP1_EnableClock(LL_AHB2_GRP1_PERIPH_GPIOC);
    LL_APB2_GRP1_EnableClock(LL_APB2_GRP1_PERIPH_SYSCFG);
    LL_GPIO_SetPinMode(GPIOC, LL_GPIO_PIN_13, LL_GPIO_MODE_INPUT);
    LL_GPIO_SetPinPull(GPIOC, LL_GPIO_PIN_13, LL_GPIO_PULL_NO);
    NVIC_EnableIRQ(EXTI15_10_IRQn);
    NVIC_SetPriority(EXTI15_10_IRQn, 0);
    LL_SYSCFG_SetEXTISource(LL_SYSCFG_EXTI_PORTC, LL_SYSCFG_EXTI_LINE13);
    LL_EXTI_EnableIT_0_31(LL_EXTI_LINE_13);
    LL_EXTI_EnableFallingTrig_0_31(LL_EXTI_LINE_13);
}

void EXTI15_10_IRQHandler(void)
{
    if(LL_EXTI_IsActiveFlag_0_31(LL_EXTI_LINE_13) != RESET)
    {
        LL_EXTI_ClearFlag_0_31(LL_EXTI_LINE_13);
        i++;
    }
}

```

```

        if(i == 3)
            i=0;
    }
}

```

在主函数调用 `Configure_EXTI()`并添加代码:

```

while (1)
{
    LL_GPIO_TogglePin(GPIOA, LL_GPIO_PIN_5);
    if(i == 0)
    {
        LL_mDelay(100);
    }
    else if(i == 1)
    {
        LL_mDelay(500);
    }
    else
    {
        LL_mDelay(1000);
    }
}

```

实现功能: 每按一次按键, LED 改变一次闪烁频率, 间隔 0.1s,0.5s,1s 三种不同频率循环闪烁。

最后, 这个 LL 库总体给人感觉是使用方法跟标准外设库很像, 但也有不同, 前面也有提到, 它全是直接操作寄存器, 程序效率一定程度上会比较高, 但使用起来有时候不一定那么方便, 可移植性也不好, 所以把它和 HAL 库结合起来使用是最好的方式, 在程序要求效率的地方就可以使用这些函数, 而其他地方使用 HAL 库, 这也是我们常常使用的方法, 只不过以前是使用 HAL 库, 需要的地方我们自己直接写操作寄存器, 而现在可以换成使用这个【LL 库】了。