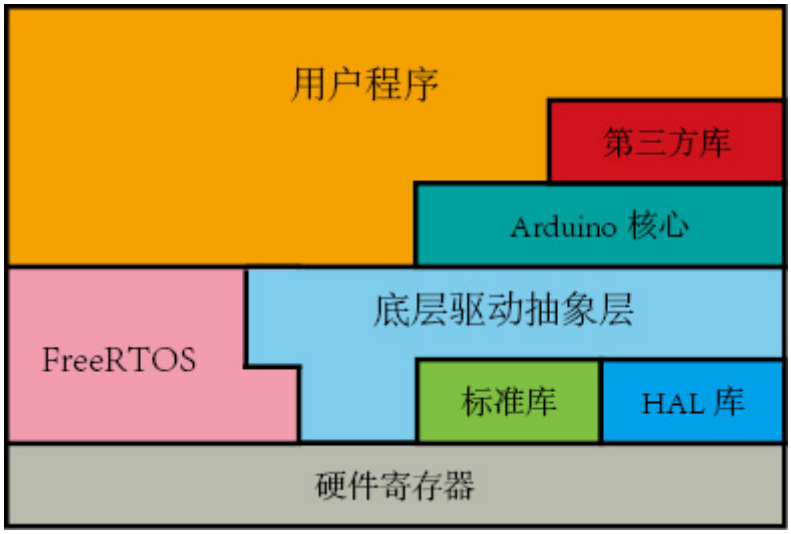
框架介绍

框架示意图



各模块功能说明

层级	模块	功能	移植说明
1	寄存器	程序和硬件直接的桥梁	寄存器的编排和具体厂家以及芯片相关
2	标准库/HAL库/其他库	芯片开发库	由芯片厂家提供
2	FreeRTOS	提供多任务处理功能	根据不同芯片进行移植
3	底层驱动抽象层(LLA_Drivers)	向上层程序提供统一的接口标准	需要移植者根据模板进行移植
4	Arduino核心	提供Arduino核心API功能	已通过底层驱动抽象层实现，无需移植
5	第三方库	由Arduino社区贡献者编写的开源库	无
6	用户程序	由用户根据实际项目需要所编写的程序，可使用所有下层功能	无

底层驱动抽象层

> 底层驱动抽象层 (**Low-lever drivers abstraction layer**) 以后简称**LLA**，向上层程序提供了一个API集合，向下规定了一种移植的规范（函数原型）。而上层程序都是通过LLA层API实现的，所以移植者只需要完成所有函数原型的实现即可完成整体项目在不同芯片平台的迁移。

LLA层模块介绍

序号	模块	作用	是否需要移植
1	LLA_ADC	ADC功能的初始化，以及ADC采集	YES
2	LLA_advanceIO	串行移位输入输出，脉宽测量	YES
3	LLA_assert	LLA层内部断言,对函数输入参数进行断言	YES
4	LLA_baseIO	基础引脚操作，模式配置，电平读写	YES
5	LLA_compiler	编译器相关	NO
6	LLA_errorCode	断言错误提示组件	NO
7	LLA_errorCodeInfos	错误提示中英文信息	NO
8	LLA_EXTI	外部中断	YES
9	LLA_flash	片内Flash读写	YES
10	LLA_PWM	PWM输出	YES
11	LLA_SPI	SPI通讯	YES
12	LLA_SYS_clock	系统时钟初始化，获取	YES
13	LLA_SYS_IRQ	系统中断开关	YES
14	LLA_SYS_rest	系统软件复位	YES
15	LLA_SYS_time	系统计时，精确的阻塞延迟	YES
16	LLA_timer	硬件定时器	YES
17	LLA_UART	串口通讯	YES
18	LLA_WDG	独立看门狗	YES

LLA_ADC模块

1. API一览

序号	函数原型	作用
1	`void LLA_ADC_Init(void)`	初始化ADC模块
2	`uint32_t LLA_ADC_Read(BaseIO_name_t name)`	读引脚ADC值
3	`uint32_t LLA_ADC_ReadSpecial(LLA_ADC_Special_channel_t channel)`	读特殊通道ADC值

2. 类型介绍

2.1 LLA_ADC_Special_channel_t（ADC特殊通道）

```
typedef enum{
    LLA_ADC_SPECIAL_CHANNEL_0,
    LLA_ADC_SPECIAL_CHANNEL_1,
    LLA_ADC_SPECIAL_CHANNEL_2,
    LLA_ADC_SPECIAL_CHANNEL_3,
    LLA_ADC_SPECIAL_CHANNEL_4,

    _LLA_ADC_SPECIAL_CHANNEL_MAX
}LLA_ADC_Special_channel_t;
```

3. 函数介绍

3.1 LLA_ADC_Init（初始化ADC模块）

函数原型`void LLA_ADC_Init(void)`

输入参数：无

返回值：无

例程

```
该函数由框架自动调用，无需用户调用。
```

3.2 LLA_ADC_Read（读引脚ADC值）

函数原型`uint32_t LLA_ADC_Read(BaseIO_name_t name)`

输入参数：1
`BaseIO_name_t name`引脚名称，枚举类型（见 LLA_baseIO 中定义）。

返回值：ADC采样值

例程

```
uint32_t val = LLA_ADC_Read(PA2);//读PA2引脚ADC值
```

3.3 LLA_ADC_ReadSpecial（读特殊通道ADC值）

函数原型`uint32_t LLA_ADC_ReadSpecial(LLA_ADC_Special_channel_t channel)`

输入参数：1
`LLA_ADC_Special_channel_t channel`特殊通道名称，枚举类型。

返回值：ADC采样值

例程

```
uint32_t val = LLA_ADC_Read(LLA_ADC_SPECIAL_CHANNEL_0);//读特殊通道0的ADC值
```

LLA_advanceIO模块

1. API一览

序号	函数原型	作用
1	`uint8_t LLA_AdvanceIO_ShiftIn(BaseIO_name_t dataPin, BaseIO_name_t clockPin, LLA_BitOrder_t bitOrder)`	串行移位输入
2	`void LLA_AdvanceIO_ShiftOut(BaseIO_name_t dataPin, BaseIO_name_t clockPin, LLA_BitOrder_t bitOrder, uint8_t val)`	串行移位输出
3	`uint32_t LLA_AdvanceIO_PluseMeasure(BaseIO_name_t name,AdvanceIO_PWC_t pwc,BaseIO_status_t idle_status,uint32_t time_out)`	脉宽测量

2. 类型介绍

2.1 LLA_BitOrder_t （比特顺序）

```
typedef enum{
    LLA_LSBFIRST,//低位先出
    LLA_MSBFIRST,//高位先出
}LLA_BitOrder_t;
```

2.2 AdvanceIO_PWC_t（脉宽测量模式）

```
typedef enum{
    AdvanceIO_PWC_F2F,//下降沿到下降沿模式
    AdvanceIO_PWC_R2R,//上升沿到上升沿模式
    AdvanceIO_PWC_F2R,//下降沿到上升沿模式
    AdvanceIO_PWC_R2F,//上升沿到下降沿模式

    _ADVANCEIO_PWC_MAX
}AdvanceIO_PWC_t;
```

3. 函数介绍

3.1 LLA_AdvanceIO_ShiftIn（串行移位输入）

函数原型`uint8_t LLA_AdvanceIO_ShiftIn(BaseIO_name_t dataPin, BaseIO_name_t clockPin, LLA_BitOrder_t bitOrder)`

输入参数：3
`BaseIO_name_t dataPin`数据引脚名称，枚举类型（见 LLA_baseIO 中定义）。
`BaseIO_name_t clockPin`时钟引脚名称，枚举类型（见 LLA_baseIO 中定义）。
`LLA_BitOrder_t bitOrder`比特顺序，枚举类型。

返回值：`uint8_t` 输入的1字节数据

例程

```
LLA_BaseIO_Mode(PA3,BaseIOMode_INPUT);
LLA_BaseIO_Mode(PA2,BaseIOMode_OUTPUT);
uint8_t data = LLA_AdvanceIO_ShiftIn(PA3,PA2,LLA_MSBFIRST);//PA3引脚作为数据输入引脚，PA2引脚作为时钟输出引脚。高位先出。
```

3.2 LLA_AdvanceIO_ShiftOut（串行移位输出）

函数原型`void LLA_AdvanceIO_ShiftOut(BaseIO_name_t dataPin, BaseIO_name_t clockPin, LLA_BitOrder_t bitOrder, uint8_t val)`

输入参数：4
`BaseIO_name_t dataPin`数据引脚名称，枚举类型（见 LLA_baseIO 中定义）。
`BaseIO_name_t clockPin`时钟引脚名称，枚举类型（见 LLA_baseIO 中定义）。
`LLA_BitOrder_t bitOrder`比特顺序，枚举类型。
`uint8_t val`要移位输出的字节数据。

返回值：无

例程

```
LLA_BaseIO_Mode(PA3,BaseIOMode_OUTPUT);
LLA_BaseIO_Mode(PA2,BaseIOMode_OUTPUT);
LLA_AdvanceIO_ShiftOut(PA3,PA2,LLA_MSBFIRST,0x68);//PA3引脚作为数据输出引脚，PA2引脚作为时钟输出引脚。高位先出。输出0x68
```

3.3 LLA_AdvanceIO_PluseMeasure（PWC脉宽测量）

函数原型：`uint32_t LLA_AdvanceIO_PluseMeasure(BaseIO_name_t name,AdvanceIO_PWC_t pwc,BaseIO_status_t idle_status,uint32_t time_out)`

输入参数：4
`BaseIO_name_t name` 测量引脚名称，枚举类型（见 LLA_baseIO 中定义）。
`AdvanceIO_PWC_t pwc` 脉宽测量模式，枚举类型。
`BaseIO_status_t idle_status` 引脚空闲状态，枚举类型（见 LLA_baseIO 中定义）。
`uint32_t time_out` 最大超时时间，（单位 ms，自函数调用时开始计时。发生超时，返回值为0，并触发断言。

返回值：`uint32_t` 脉宽持续时间，（单位 us）。测量失败则返回值恒等于0。

例程

```
/* 测量高电平持续为230us的脉宽 */
uint32_t val =
LLA_AdvanceIO_PluseMeasure(PB3,AdvanceIO_PWC_R2F,BaseIOStatus_LOW,10);//测量PB3引脚的高脉宽持续时间
```

LLA_assert模块

1. API一览

序号	宏原型	作用
1	LLA_ASSERT(e,driver,error)	LLA层断言
2	LLA_AssertPrintf(...)	断言输出，默认使用串口1进行断言输出。该函数用户不可调用

2. 宏定义介绍

2.1 LLA_ASSERT（LLA层断言）

宏定义`#define LLA_ASSERT(e,driver,error)`

输入参数：3
`e`表达式，若表达式为真则断言成功，若表达式为假则断言失败。失败后会打印错误信息，并进入`LLA_errorCode_Handler`
`driver`驱动代码。
`error`错误代码。

返回值：无

例程

```
LLA_ASSERT(IS_IO_NAME(name),DriverCode_BaseIO,BaseIO_errorIOName);//断言输入的name是否为正确的引脚名称。断言失败则会触发`BaseIO_errorIOName`错误
```

LLA_baseIO模块

1. API一览

序号	函数原型	作用
1	`void LLA_BaseIO_Mode(BaseIO_name_t name,BaseIO_mode_t mode,BaseIO_status_t outputStatus)`	配置引脚的工作模式
2	`void LLA_BaseIO_Write(BaseIO_name_t name,BaseIO_status_t outputStatus)`	设置引脚电平
3	`BaseIO_status_t LLA_BaseIO_ReadInput(BaseIO_name_t name)`	读取引脚输入电平
4	`BaseIO_status_t LLA_BaseIO_ReadOutput(BaseIO_name_t name)`	读取引脚输出电平
5	`BaseIO_status_t LLA_BaseIO_Toggle(BaseIO_name_t name)`	翻转引脚输出电平

2. 类型介绍

2.1 BaseIO_status_t（IO电平状态）

```
typedef enum{
    BaseIOStatus_LOW,
    BaseIOStatus_HIGH,

    _BASEIO_STATUS_MAX
}BaseIO_status_t;
```

2.2 BaseIO_mode_t（IO口模式）

```
typedef enum{
    BaseIOMode_INPUT,//浮空输入
    BaseIOMode_INPUT_PU,//上拉输入
    BaseIOMode_INPUT_PD,//上拉输出
    BaseIOMode_OUTPUT,//推挽输出
    BaseIOMode_OUTPUT_OD,//开漏输出
    BaseIOMode_OUTPUT_OD_PU,//开漏上拉输出
    BaseIOMode_OUTPUT_OD_PD,//开漏下拉输出

    _BASEIO_MODE_MAX
}BaseIO_mode_t;
```

2.3 BaseIO_name_t（IO名称）

```
typedef enum{
    PA0,PA1,PA2,PA3,PA4,PA5,PA6,PA7,PA8,PA9,PA10,PA11,PA12,PA13,PA14,PA15,
    PB0,PB1,PB2,PB3,PB4,PB5,PB6,PB7,PB8,PB9,PB10,PB11,PB12,PB13,PB14,PB15,
    PC0,PC1,PC2,PC3,PC4,PC5,PC6,PC7,PC8,PC9,PC10,PC11,PC12,PC13,PC14,PC15,
    PD0,PD1,PD2,PD3,PD4,PD5,PD6,PD7,PD8,PD9,PD10,PD11,PD12,PD13,PD14,PD15,
    PE0,PE1,PE2,PE3,PE4,PE5,PE6,PE7,PE8,PE9,PE10,PE11,PE12,PE13,PE14,PE15,
    PF0,PF1,PF2,PF3,PF4,PF5,PF6,PF7,PF8,PF9,PF10,PF11,PF12,PF13,PF14,PF15,
    PG0,PG1,PG2,PG3,PG4,PG5,PG6,PG7,PG8,PG9,PG10,PG11,PG12,PG13,PG14,PG15,

    _BASEIO_NAME_MAX,
}BaseIO_name_t;
```

3. 函数介绍

3.1 LLA_BaseIO_Mode （引脚模式配置）

函数原型`void LLA_BaseIO_Mode(BaseIO_name_t name,BaseIO_mode_t mode,BaseIO_status_t outputStatus)`

输入参数：3
`BaseIO_name_t name`引脚名称，枚举类型。
`BaseIO_mode_t mode`引脚模式，枚举类型。
`BaseIO_status_t outputStatus`默认输出电平的状态，枚举类型。只有在输出模式时使用，输入模式该参数被忽略。

返回值：无

例程

```
LLA_BaseIO_Mode(PA4,BaseIOMode_OUTPUT,BaseIOStatus_LOW);//配置引脚PA4为推挽输出模式，并设置IO电平为低电平
LLA_BaseIO_Mode(PA5,BaseIOMode_INPUT_PU,BaseIOStatus_LOW);//配置引脚PA5为上拉输入模式。（输入模式设置IO口电平被忽略）
```

3.2 LLA_BaseIO_Write（写引脚电平）

函数原型`void LLA_BaseIO_Write(BaseIO_name_t name,BaseIO_status_t outputStatus)`

输入参数：2
`BaseIO_name_t name`引脚名称，枚举类型。
`BaseIO_status_t outputStatus`电平状态，枚举类型。

返回值：无

例程

```
LLA_BaseIO_Mode(PA4,BaseIOMode_OUTPUT,BaseIOStatus_LOW);//配置PA4为输出模式
LLA_BaseIO_Write(PA4,BaseIOStatus_HIGH);//设置PA4为高电平
```

3.3 LLA_BaseIO_ReadInput（读引脚输入电平）

函数原型`BaseIO_status_t LLA_BaseIO_ReadInput(BaseIO_name_t name)`

输入参数：1
`BaseIO_name_t name`引脚名称，枚举类型。

返回值：`BaseIO_status_t`输入电平状态，枚举类型。

例程

```
LLA_BaseIO_Mode(PA4,BaseIOMode_INPUT_PU);//配置PA4为上拉输入模式
BaseIO_status_t input = LLA_BaseIO_ReadInput(PA4);//读取PA4输入电平
```

3.4 LLA_BaseIO_ReadOutput（读引脚输出电平）

函数原型`BaseIO_status_t LLA_BaseIO_ReadOutput(BaseIO_name_t name)`

输入参数：1
`BaseIO_name_t name`引脚名称，枚举类型。

返回值：`BaseIO_status_t`输出电平状态，枚举类型。

例程

```
LLA_BaseIO_Mode(PA4,BaseIOMode_OUTPUT,BaseIOStatus_LOW);//配置PA4为输出模式
LLA_BaseIO_Write(PA4,BaseIOStatus_HIGH);//设置PA4为高电平
BaseIO_status_t output = LLA_BaseIO_ReadOutput(PA4);//读取PA4输入电平,此时output为高电平
```

3.5 LLA_BaseIO_Toggle（翻转引脚输出电平）

函数原型`BaseIO_status_t LLA_BaseIO_Toggle(BaseIO_name_t name)`

输入参数：1

返回值：`BaseIO_status_t name`翻转后的电平状态，枚举类型。

例程

```
LLA_BaseIO_Mode(PA4,BaseIOMode_OUTPUT,BaseIOStatus_LOW);//配置PA4为输出模式
BaseIO_status_t output=LLA_BaseIO_Toggle(PA4);//执行该语句后output为高
output=LLA_BaseIO_Toggle(PA4);//执行该语句后outpu为低
```

LLA_EXTI模块

1. API一览

序号	函数原型	作用
1	`void LLA_EXTI_AttachInterrupt(BaseIO_name_t name, LLA_EXTI_cb_t cb,LLA_EXTI_TYPE_t type)`	初始化引脚外部中断，并设置回调
2	`void LLA_EXTI_DetachInterrupt(BaseIO_name_t name)`	关闭引脚外部中断

2. 类型介绍

2.1 LLA_EXTI_TYPE_t（外部中断类型）

```
typedef enum{
    LLA_EXTI_TYPE_LOW,//低电平触发
    LLA_EXTI_TYPE_HIGH,//高电平触发
    LLA_EXTI_TYPE_FALLING,//下降沿触发
    LLA_EXTI_TYPE_RISING,//上升沿触发
    LLA_EXTI_TYPE_CHANGE,//下降和上升沿都触发

    _LLA_EXTI_TYPE_MAX
}LLA_EXTI_TYPE_t;
```

2.2 LLA_EXTI_cb_t（外部中断回调）

`typedef void (*LLA_EXTI_cb_t)(void);`

3. 函数介绍

3.1 LLA_EXTI_AttachInterrupt（设置外部中断）

函数原型：`void LLA_EXTI_AttachInterrupt(BaseIO_name_t name, LLA_EXTI_cb_t cb,LLA_EXTI_TYPE_t type)`

输入参数：3
`BaseIO_name_t name` 引脚名称，枚举类型（见 LLA baseIO 中定义）。
`LLA_EXTI_cb_t cb` 回调函数，函数指针。
`LLA_EXTI_TYPE_t type` 外部中断类型，枚举类型。

返回值：无

例程

```
/* 定义外部中断回调 */
void exti_cb(void){

}
LLA_EXTI_AttachInterrupt(PA4,exti_cb,LLA_EXTI_TYPE_FALLING);//为PA4设置外部中断及回调函数
```

3.2 LLA_EXTI_DetachInterrupt（关闭外部中断）

函数原型：`void LLA_EXTI_DetachInterrupt(BaseIO_name_t name)`

输入参数：1
`BaseIO_name_t name` 引脚名称，枚举类型（见 LLA baseIO 中定义）。

返回值：无

例程

```
/* 定义外部中断回调 */
void exti_cb(void){
 LLA_EXTI_DetachInterrupt(PA4);//关闭PA4的外部回调
}
LLA_EXTI_AttachInterrupt(PA4,exti_cb,LLA_EXTI_TYPE_FALLING);//为PA4设置外部中断及回调函数

/* 以上程序当外部中断触发满足后只会触发一次exti_cb */
```

LLA_flash模块

1. API一览

序号	函数原型	作用
1	`void LLA_Flash_Init(void)`	初始化Flash控制模块
2	`uint32_t LLA_Flash_GetSize(void)`	获取Flash总可用大小
3	`void LLA_Flash_Erase(uint32_t offset,uint32_t len)`	擦除FLASH
4	`void LLA_Flash_WriteBytes(uint32_t offset,uint8_t *data,uint32_t len)`	写入Flash数据
5	`void LLA_Flash_ReadBytes(uint32_t offset,uint8_t *data,uint32_t len)`	读取Flash数据

2. 函数介绍

2.1 LLA_Flash_Init（初始化Flash控制模块）

函数原型`void LLA_Flash_Init(void)`

输入参数：无

返回值：无

例程

```
该函数由框架自动调用，无需用户调用。
```

2.2 LLA_Flash_GetSize（获取Flash总可用大小）

函数原型`uint32_t LLA_Flash_GetSize(void)`

输入参数：无

返回值：`uint32_t`Flash可用大小（字节）

例程

```
uint32_t flash_size = LLA_Flash_GetSize(void);
```

2.3 LLA_Flash_Erase（擦除Flash）

函数原型`void LLA_Flash_Erase(uint32_t offset,uint32_t len)`

输入参数：2
`uint32_t offset`擦除区域起始位置，0表示从Flash可用区域0位置开始。
`uint32_t len`擦除长度，`len`必须小于Flash总大小

返回值：无

例程

```
LLA_Flash_Erase(0,LLA_Flash_GetSize());//擦除整个Flash可用区域
LLA_Flash_Erase(0,256);//擦除第一扇区
```

2.4 LLA_Flash_WriteBytes（写入Flash数据）

函数原型`void LLA_Flash_WriteBytes(uint32_t offset,uint8_t *data,uint32_t len)`

输入参数：3
`uint32_t offset`写入区域起始位置，0表示从Flash可用区域0位置开始。
`uint8_t *data`写入数据的指针。
`uint32_t len`写入长度，`len`必须小于Flash总大小。

返回值：无

例程

```
char buf[]="Apple";
float pi = 3.14;
int num = 10;
LLA_Flash_WriteBytes(0,(uint8_t*)buf,sizeof(buf));//写入数组
LLA_Flash_WriteBytes(6,(uint8_t*)&pi,sizeof(float));//写入浮点数
LLA_Flash_WriteBytes(10,(uint8_t*)&num,sizeof(int));//写入整数
```

2.5 LLA_Flash_ReadBytes（读取Flash数据）

函数原型`void LLA_Flash_ReadBytes(uint32_t offset,uint8_t *data,uint32_t len)`

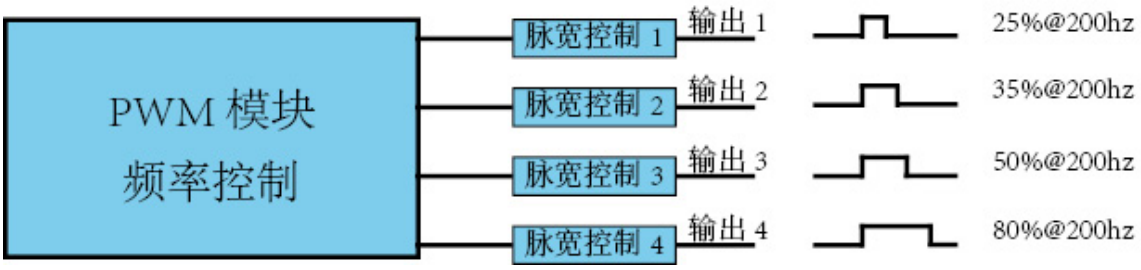
输入参数：3
`uint32_t offset`读取区域起始位置，0表示从Flash可用区域0位置开始。
`uint8_t *data`缓存数据的指针。
`uint32_t len`读取长度，`len`必须小于Flash总大小。

返回值：无

例程

```
uint8_t buf[200]={0};
LLA_Flash_ReadBytes(0,buf,200);//读取Flash前200字节到buf数组
```

LLA_PWM模块



> pwm由pwm模块生成一个统一的pwm基础频率，通过每个功能引脚的脉宽控制。可独立控制引脚占空比（不可单独控制频率）。

1. API一览

序号	函数原型	作用
1	`void LLA_PWM_BaseHZ(uint32_t hz)`	设置PWM频率
2	`void LLA_PWM_Output1(BaseIO_name_t name,uint8_t duty,PWM_Polar_t polar)`	PWM输出方式一
3	`void LLA_PWM_OutPut2(BaseIO_name_t name,uint32_t hz,uint8_t duty,PWM_Polar_t polar)`	PWM输出方式二
4	`void LLA_PWM_Stop(BaseIO_name_t name)`	停止PWM输出

2. 类型介绍

2.1 PWM_Polar_t（PWM极性）

```
typedef enum{
    PWM_Polar_HIGH,//PWM极性高电平
    PWM_Polar_LOW,//PWM极性低电平

    _PWM_POLAR_MAX
}PWM_Polar_t;
```

3. 函数介绍

3.1 LLA_PWM_BaseHZ （设置PWM频率）

函数原型`void LLA_PWM_BaseHZ(uint32_t hz)`

输入参数：1
`uint32_t hz`设置PWM模块频率

返回值：无

例程

```
LLA_PWM_BaseHZ(1000);//设置pwm模块频率为 1KHz
```

3.2 LLA_PWM_Output1 （PWM输出方式一）

函数原型`void LLA_PWM_Output1(BaseIO_name_t name,uint8_t duty,PWM_Polar_t polar)`

输入参数：3
`BaseIO_name_t name`pwm输出引脚名称
`uint8_t duty`占空比，`[0,255]`表示0~100%的占空比。
`PWM_Polar_t polar`输出极性

返回值：无

例程

```
LLA_PWM_BaseHZ(1000);//设置pwm模块频率为 1KHz
LLA_PWM_Output1(PA4,64,PWM_Polar_HIGH);//输出高电平占空比25%，频率为1Khz的PWM波形
```

3.3 LLA_PWM_OutPut2 （PWM输出方式二）

函数原型`void LLA_PWM_OutPut2(BaseIO_name_t name,uint32_t hz,uint8_t duty,PWM_Polar_t polar)`

输入参数：4
`BaseIO_name_t name`pwm输出引脚名称
`uint32_t hz`pwm模块频率
`uint8_t duty`占空比，`[0,255]`表示0~100%的占空比。
`PWM_Polar_t polar`输出极性

返回值：无

例程

```
LLA_PWM_Output2(PA4,1000,64,PWM_Polar_HIGH);//输出高电平占空比25%，频率为1Khz的PWM波形
```

3.4 LLA_PWM_Stop （停止PWM输出）

函数原型`void LLA_PWM_Stop(BaseIO_name_t name)`

输入参数：1
`BaseIO_name_t name`pwm输出引脚名称

返回值：无

例程

```
LLA_PWM_Output2(PA4,1000,64,PWM_Polar_HIGH);//输出高电平占空比25%，频率为1Khz的PWM波形
LLA_PWM_Stop(PA4);//停止PA4引脚的PWM输出
```

LLA_SPI模块

1. API一览

序号	函数原型	作用
1	`void LLA_SPI_Init(LLA_SPI_t spi,uint32_t clock,LLA_SPI_Mode_t mode,LLA_SPI_CS_t cs_mode)`	初始化SPI模块
2	`void LLA_SPI_DeInit(LLA_SPI_t spi)`	失能SPI模块
3	`void LLA_SPI_WriteBytes(LLA_SPI_t spi,uint8_t * data,uint32_t len)`	SPI写数据
4	`void LLA_SPI_ReadBytes(LLA_SPI_t spi,uint8_t * buf,uint32_t len)`	SPI读数据

2. 类型介绍

2.1 LLA_SPI_t（SPI名称）

```
typedef enum{
    LLA_SPI0,
    LLA_SPI1,
    LLA_SPI2,
    LLA_SPI3,

    _LLA_SPI_MAX
}LLA_SPI_t;
```

2.2 LLA_SPI_Mode_t（SPI时钟模式）

```
typedef enum{
    LLA_SPI_Mode_CLK_L_CPHA_L,//时钟空闲为低电平，第一边沿采样
    LLA_SPI_Mode_CLK_L_CPHA_H,//时钟空闲为低电平，第二边沿采样
    LLA_SPI_Mode_CLK_H_CPHA_L,//时钟空闲为高电平，第一边沿采样
    LLA_SPI_Mode_CLK_H_CPHA_H,//时钟空闲为高电平，第二边沿采样

    _LLA_SPI_MODE_MAX
}LLA_SPI_Mode_t;
```

2.3 LLA_SPI_CS_t（SPI片选模式）

```
typedef enum{
    LLA_SPI_CS_soft,//软件控制片选
    LLA_SPI_CS_Auto,//自动片选

    _LLA_SPI_CS_MAX
}LLA_SPI_CS_t;
```

3. 函数介绍

3.1 LLA_SPI_Init （初始化SPI模块）

函数原型`void LLA_SPI_Init(LLA_SPI_t spi,uint32_t clock,LLA_SPI_Mode_t mode,LLA_SPI_CS_t cs_mode)`

输入参数：4
`LLA_SPI_t spi`spi名称，枚举类型。
`uint32_t clock`时钟频率，单位hz
`LLA_SPI_Mode_t mode`时钟模式，枚举类型。
`LLA_SPI_CS_t cs_mode`片选模式，枚举类型。

返回值：无

例程

```
LLA_SPI_Init(LLA_SPI0,24000000,LLA_SPI_Mode_CLK_L_CPHA_L,LLA_SPI_CS_soft);//设置SPI0时钟频率24Mhz，上升沿采样，软件控制片选
```

3.2 LLA_SPI_DeInit （失能SPI模块）

函数原型`void LLA_SPI_DeInit(LLA_SPI_t spi)`

输入参数：1
`LLA_SPI_t spi`spi名称，枚举类型。

返回值：无

例程

```
LLA_SPI_Init(LLA_SPI0,24000000,LLA_SPI_Mode_CLK_L_CPHA_L,LLA_SPI_CS_soft);//设置SPI0时钟频率24Mhz，上升沿采样，软件控制片选
LLA_SPI_DeInit(LLA_SPI0);//使能SPI0
```

3.3 LLA_SPI_WriteBytes （SPI写数据）

函数原型`void LLA_SPI_WriteBytes(LLA_SPI_t spi,uint8_t * data,uint32_t len)`

输入参数：3
`LLA_SPI_t spi`spi名称，枚举类型。
`uint8_t * data`写入数据的指针
`uint32_t len` 数据长度

返回值：无

例程

```
uint8_t data[]={0xc2,0x63,0x7d};
LLA_BaseIO_Write(PA4,BaseIOStatus_LOW);//片选选中(见 LLA_baseIO)
LLA_SPI_WriteBytes(LLA_SPI0,data,sizeof(data));
LLA_BaseIO_Write(PA4,BaseIOStatus_HIGH);//片选释放
```

3.4 LLA_SPI_ReadBytes （SPI读数据）

函数原型`void LLA_SPI_ReadBytes(LLA_SPI_t spi,uint8_t * buf,uint32_t len)`

输入参数：3
`LLA_SPI_t spi`spi名称，枚举类型。
`uint8_t * data`缓存数据的指针
`uint32_t len` 读取数据长度

返回值：无

例程

```
uint8_t data[6]={0};
LLA_BaseIO_Write(PA4,BaseIOStatus_LOW);//片选选中(见 LLA_baseIO)
LLA_SPI_ReadBytes(LLA_SPI0,data,sizeof(data));//从SPI0读取6字节数据到data数组
LLA_BaseIO_Write(PA4,BaseIOStatus_HIGH);//片选释放
```

LLA_SYS_clock模块

1. API一览

序号	函数原型	作用
1	`void LLA_SYS_clock_Init(void)`	初始化系统时钟
2	`void LLA_SYS_clock_Update(void)`	更新系统时钟
3	`uint32_t LLA_SYS_clock_Get(void)`	获取系统时钟

2. 函数介绍

2.1 LLA_SYS_clock_Init （初始化系统时钟）

函数原型`void LLA_SYS_clock_Init(void)`

输入参数：无

返回值：无

例程

```
该函数由框架自动调用，无需用户调用。
```

2.2 LLA_SYS_clock_Update （更新系统时钟）

函数原型`void LLA_SYS_clock_Update(void)`

输入参数：无

返回值：无

例程

```
该函数由框架自动调用，无需用户调用。
```

2.3 LLA_SYS_clock_Get （获取系统时钟）

函数原型`uint32_t LLA_SYS_clock_Get(void)`

输入参数：无

返回值：`uint32_t`系统时钟，单位hz

例程

```
uint32_t clk = LLA_SYS_clock_Get();//读取系统时钟速度到clk变量（单位hz）
```

LLA_SYS_IRQ模块

1. API一览

序号	函数原型	作用
1	`void LLA_SYS_IRQ_Enable(void)`	允许系统发生中断
2	`void LLA_SYS_IRQ_Disable(void)`	不允许所有中断

2. 函数介绍

2.1 LLA_SYS_IRQ_Enable （允许系统发生中断）

函数原型`void LLA_SYS_IRQ_Enable(void)`

输入参数：无

返回值：无

例程

```
LLA_SYS_IRQ_Enable();//允许系统中断
```

2.2 LLA_SYS_IRQ_Disable （不允许所有中断）

函数原型`void LLA_SYS_IRQ_Disable(void)`

输入参数：无

返回值：无

例程

```
LLA_SYS_IRQ_Disable();//禁止系统中断
```

LLA_SYS_rest模块

1. API一览

序号	函数原型	作用
1	`void LLA_SYS_rest(void)`	系统软重启

2. 函数介绍

2.1 LLA_SYS_rest （系统软重启）

函数原型`void LLA_SYS_rest(void)`

输入参数：无

返回值：无

例程

```
LLA_SYS_rest();//重启系统
```

LLA_SYS_time模块

1. API一览

序号	函数原型	作用
1	`void LLA_SYS_Time_Init(void)`	初始化系统时间模块
2	`uint32_t LLA_SYS_Time_Millis(void)`	获取系统运行毫秒数
3	`uint32_t LLA_SYS_Time_Micros(void)`	获取系统运行微秒数
4	`void LLA_SYS_Time_DelayMS(uint32_t ms)`	系统毫秒级延时
5	`void LLA_SYS_Time_DelayUS(uint32_t us)`	系统微秒级延时
6	`uint32_t LLA_SYS_Time_ConsumeMillis(uint32_t pre_ms)`	计算经过的毫秒数
7	`uint32_t LLA_SYS_Time_ConsumeMicros(uint32_t pre_us)`	计算经过的微秒数

2. 函数介绍

2.1 LLA_SYS_Time_Init （初始化系统时间模块）

函数原型`void LLA_SYS_Time_Init(void)`

输入参数：无

返回值：无

例程

```
该函数由框架自动调用，无需用户调用。
```

2.2 LLA_SYS_Time_Millis （获取系统运行毫秒数）

函数原型`uint32_t LLA_SYS_Time_Millis(void)`

输入参数：无

返回值：`uint32_t`系统运行毫秒数

例程

```
uint32_t current = LLA_SYS_Time_Millis();//获取系统自上电以来经过的毫秒数
```

2.3 LLA_SYS_Time_Micros （获取系统运行微秒数）

函数原型`LLA_SYS_Time_Micros(void)`

输入参数：无

返回值：`uint32_t`系统运行微秒数

例程

```
uint32_t current = LLA_SYS_Time_Micros();//获取系统自上电以来经过的微秒数
```

2.4 LLA_SYS_Time_DelayMS （系统毫秒级延时）

函数原型`void LLA_SYS_Time_DelayMS(uint32_t ms)`

输入参数：1
`uint32_t ms`延迟的毫秒数

返回值：无

例程

```
LLA_SYS_Time_DelayMS(1000);//延迟1000毫秒
```

2.5 LLA_SYS_Time_DelayUS （系统微秒级延时）

函数原型`void LLA_SYS_Time_DelayUS(uint32_t us)`

输入参数：1
`uint32_t us`延迟的微秒数

返回值：无

例程

```
LLA_SYS_Time_DelayUS(200);//延迟200微秒
```

2.6 LLA_SYS_Time_ConsumeMillis （计算经过的毫秒数）

函数原型`uint32_t LLA_SYS_Time_ConsumeMillis(uint32_t pre_ms)`

输入参数：1
`uint32_t pre_ms`上一次记录的系统毫秒数

返回值：`uint32_t`经过的毫秒数

例程

```
uint32_t ms = LLA_SYS_Time_Millis();
do_something();
uint32_t ms_consume = LLA_SYS_Time_ConsumeMillis(ms);//计算do_something用时
```

2.7 LLA_SYS_Time_ConsumeMicros （计算经过的微秒数）

函数原型`uint32_t LLA_SYS_Time_ConsumeMicros(uint32_t pre_us)`

输入参数：1
`uint32_t pre_us`上一次记录的系统微秒数

返回值：`uint32_t`经过的微秒数

例程

```
uint32_t us = LLA_SYS_Time_Micros();
do_something();
uint32_t us_consume = LLA_SYS_Time_ConsumeMicros(us);//计算do_something用时
```

LLA_timer模块

1. API一览

序号	函数原型	作用
1	`void LLA_Timer_Set(LLA_Timer_t timer,uint32_t period,Timer_Handler_t handler)`	配置定时器
2	`void LLA_Timer_Start(LLA_Timer_t timer)`	启动定时器
3	`void LLA_Timer_Stop(LLA_Timer_t timer)`	停止定时器

2. 类型介绍

2.1 LLA_Timer_t（定时器名称）

```
typedef enum{
    LLA_TIMER0,
    LLA_TIMER1,
    LLA_TIMER2,
    LLA_TIMER3,
    LLA_TIMER4,
    LLA_TIMER5,
    LLA_TIMER6,
    LLA_TIMER7,

    _LLA_TIMER_MAX
}LLA_Timer_t;
```

2.2 Timer_Handler_t（定时器回调类型）

```
typedef void (*Timer_Handler_t)(void);
```

3. 函数介绍

3.1 LLA_Timer_Set （配置定时器）

函数原型`void LLA_Timer_Set(LLA_Timer_t timer,uint32_t period,Timer_Handler_t handler)`

输入参数：3

返回值：无
`LLA_Timer_t timer`定时器名称，枚举类型。
`uint32_t period`定时周期，单位ms。
`Timer_Handler_t handler`回调函数，该函数在中断内执行，请保证快进快出。

例程

```
void timer1_handler(){
do_something();//Do not use blocked delay in do_somethin() function.
}

void setup(){
LLA_Timer_Set(LLA_TIMER0,100,timer1_handler);
}
```

3.2 LLA_Timer_Start （启动定时器）

函数原型`void LLA_Timer_Start(LLA_Timer_t timer)`

输入参数：1

返回值：无
`LLA_Timer_t timer`定时器名称，枚举类型。

例程

```
void timer1_handler(){
do_something();//Do not use blocked delay in do_somethin() function.
}

void setup(){
LLA_Timer_Set(LLA_TIMER0,100,timer1_handler);//配置定时器1,此时定时器并未开始运行
LLA_Timer_Start(LLA_TIMER0);//启动定时器1
}
```

3.3 LLA_Timer_Stop （停止定时器）

函数原型`void LLA_Timer_Stop(LLA_Timer_t timer)`

输入参数：1

返回值：无
`LLA_Timer_t timer`定时器名称，枚举类型。

例程

```
LLA_Timer_Stop(LLA_TIMER0);//停止定时器1
```

LLA_UART模块

1. API一览

序号	函数原型	作用
1	`void LLA_UART_Init(UART_name_t name,uint32_t baudRate,UART_config_t config)`	初始化串口
2	`void LLA_UART_DeInit(UART_name_t name)`	失能串口
3	`void LLA_UART_Write(UART_name_t name,uint8_t data)`	串口写数据
4	`void LLA_UART_WriteBuffer(UART_name_t name,uint8_t *data,uint16_t len)`	串口写多字节数据
5	`int LLA_UART_Printf(UART_name_t name,const char *__restrict __format, ...)`	串口格式化输出
6	`LLA_WEAK void LLA_UART1_IRQHandler(uint8_t data)`	串口1数据接收钩子函数
7	`LLA_WEAK void LLA_UART2_IRQHandler(uint8_t data)`	串口2数据接收钩子函数
8	`LLA_WEAK void LLA_UART3_IRQHandler(uint8_t data)`	串口3数据接收钩子函数
9	`LLA_WEAK void LLA_UART4_IRQHandler(uint8_t data)`	串口4数据接收钩子函数
10	`LLA_WEAK void LLA_UART5_IRQHandler(uint8_t data)`	串口5数据接收钩子函数
11	`LLA_WEAK void LLA_UART6_IRQHandler(uint8_t data)`	串口6数据接收钩子函数
12	`LLA_WEAK void LLA_UART7_IRQHandler(uint8_t data)`	串口7数据接收钩子函数
13	`LLA_WEAK void LLA_UART8_IRQHandler(uint8_t data)`	串口8数据接收钩子函数

2. 宏定义介绍

2.1 LLA_UART_PRINTF_BUFFER_LENGTH（串口格式化输出缓冲区大小）

```
#define LLA_UART_PRINTF_BUFFER_LENGTH 512
```

该宏用于配置串口格式化输出函数的静态缓冲区大小，表示一次格式化输出最多输出多少字节字符。所有串口共用一个静态缓冲区，该函数线程不安全，请勿在多线程中使用。

3. 类型介绍

3.1 UART_name_t（串口名称）

```
typedef enum{
    LLA_UART1,
    LLA_UART2,
    LLA_UART3,
    LLA_UART4,
    LLA_UART5,
    LLA_UART6,
    LLA_UART7,
    LLA_UART8,

    _LLA_UART_MAX
}UART_name_t;
```

3.2 UART_config_t（串口配置）

```
typedef enum{
    UART_CONFIG_8N1,//8数据位，无校验，1停止位
    UART_CONFIG_8N1_5,//8数据位，无校验，1.5停止位
    UART_CONFIG_8N2,//8数据位，无校验，2停止位
    UART_CONFIG_8E1,//8数据位，偶校验，1停止位
    UART_CONFIG_8E1_5,//8数据位，偶校验，1.5停止位
    UART_CONFIG_8E2,//8数据位，偶校验，2停止位
    UART_CONFIG_8O1,//8数据位，奇校验，1停止位
    UART_CONFIG_8O1_5,//8数据位，奇校验，1.5停止位
    UART_CONFIG_8O2,//8数据位，奇校验，2停止位

    _UART_CONCIG_MAX
}UART_config_t;
```

4. 函数介绍

4.1 LLA_UART_Init（初始化串口）

函数原型`void LLA_UART_Init(UART_name_t name,uint32_t baudRate,UART_config_t config)`

输入参数：3
`UART_name_t name`串口名称，枚举类型。
`uint32_t baudRate`波特率。
`UART_config_t config`串口配置，枚举类型。

返回值：无

例程

```
LLA_UART_Init(LLA_UART1,9600,UART_CONFIG_8N1);//初始化串口1为9600波特率，8数据位无校验1停止位。
```

4.2 LLA_UART_DeInit（失能串口）

函数原型`void LLA_UART_DeInit(UART_name_t name)`

输入参数：1
`UART_name_t name`串口名称，枚举类型。

返回值：无

例程

```
LLA_UART_Init(LLA_UART1,9600,UART_CONFIG_8N1);//初始化串口1为9600波特率，8数据位无校验1停止位。
do_something();
LLA_UART_DeInit(LLA_UART1);//失能串口，关闭串口外设。
```

4.3 LLA_UART_Write（串口写数据）

函数原型`void LLA_UART_Write(UART_name_t name,uint8_t data)`

输入参数：2
`UART_name_t name`串口名称，枚举类型。
`uint8_t data`要发送的数据。

返回值：无

例程

```
/* 串口1发送 "HELLO" */
LLA_UART_Init(LLA_UART1,9600,UART_CONFIG_8N1);//初始化串口1为9600波特率，8数据位无校验1停止位。
LLA_UART_Write(LLA_UART1,'H');
LLA_UART_Write(LLA_UART1,'E');
LLA_UART_Write(LLA_UART1,'L');
LLA_UART_Write(LLA_UART1,'L');
LLA_UART_Write(LLA_UART1,'O');
```

4.4 LLA_UART_Write（串口写多字节数据）

函数原型`void LLA_UART_WriteBuffer(UART_name_t name,uint8_t *data,uint16_t len)`

输入参数：3
`UART_name_t name`串口名称，枚举类型。
`uint8_t *data`要发送数据缓冲区的指针。
`uint16_t len`数据字节数。

返回值：无

例程

```
/* 串口1发送 "HELLO" */
LLA_UART_Init(LLA_UART1,9600,UART_CONFIG_8N1);//初始化串口1为9600波特率，8数据位无校验1停止位。
char buf[]="HELLO"
LLA_UART_WriteBuffer(LLA_UART1,(uint8_t*)buf,strlen(buf));
```

4.5 LLA_UART_Printf（串口格式化输出）

函数原型`int LLA_UART_Printf(UART_name_t name,const char *__restrict __format, ...)`

输入参数：2+n
`UART_name_t name`串口名称，枚举类型。
`const char *__restrict __format`格式化字符。
`...`可变参数。

返回值：`int`发送字符串的长度

例程

```
/* 串口1发送 "HELLO" */
LLA_UART_Init(LLA_UART1,9600,UART_CONFIG_8N1);//初始化串口1为9600波特率，8数据位无校验1停止位。
char buf[]="HELLO"
LLA_UART_Printf(LLA_UART1,"%s",buf);
```

5. 钩子函数介绍

5.1 LLA_UART1_IRQHandler（串口1数据接收钩子函数）

当发生串口接收到数据后，由系统自动调用该函数。其中`uint8_t data`就是串口所接收到的数据。由用户进行接收逻辑编写。若使用ArduinoAPI，则由ArduinoAPI层编写的钩子函数处理。该函数只可以重写，不可用调用。
函数原型`LLA_WEAK void LLA_UART1_IRQHandler(uint8_t data)`

系统输入参数：
`uint8_t data`串口1所接收到的数据。

返回值：无

5.2 LLA_UART2_IRQHandler（串口2数据接收钩子函数）

见 *5.1*

5.3 LLA_UART2_IRQHandler（串口3数据接收钩子函数）

见 *5.1*

5.4 LLA_UART2_IRQHandler（串口4数据接收钩子函数）

见 *5.1*

5.5 LLA_UART2_IRQHandler（串口5数据接收钩子函数）

见 *5.1*

5.6 LLA_UART2_IRQHandler（串口6数据接收钩子函数）

见 *5.1*

5.7 LLA_UART2_IRQHandler（串口7数据接收钩子函数）

见 *5.1*

5.8 LLA_UART2_IRQHandler（串口8数据接收钩子函数）

见 *5.1*

LLA_WDG模块

1. API一览

序号	函数原型	作用
1	`void LLA_WDG_Start(uint16_t ms)`	开启看门狗
2	`void LLA_WDG_Feed(void)`	喂狗
3	`void LLA_WDG_Stop(void)`	停止看门狗
4	`float LLA_WDG_GetOVRTime(void)`	获取最大喂狗时间

2. 函数介绍

2.1 LLA_WDG_Start （开启看门狗）

函数原型`void LLA_WDG_Start(uint16_t ms)`

输入参数：1
`uint16_t ms`喂狗时间。

返回值：无

例程

```
LLA_WDG_Start(20000);//设置看门狗喂狗时间为20s，若在20s内没有执行LLA_WDG_Feed,则会复位系统
```

2.2 LLA_WDG_Feed （喂狗）

函数原型`void LLA_WDG_Feed(void)`

输入参数：无

返回值：无

例程

```
LLA_WDG_Start(20000);//设置看门狗喂狗时间为20s，若在20s内没有执行LLA_WDG_Feed,则会复位系统
do_something();//执行时间小于20s
LLA_WDG_Feed();
```

2.3 LLA_WDG_Stop （停止看门狗）

函数原型`void LLA_WDG_Stop(void)`

输入参数：无

返回值：无

2.4 LLA_WDG_GetOVRTime （获取最大喂狗时间）

函数原型`float LLA_WDG_GetOVRTime(void)`

输入参数：无

返回值：`float`最大喂狗时间。

例程

```
LLA_WDG_Start(20000);//设置看门狗喂狗时间为20s，若在20s内没有执行LLA_WDG_Feed,则会复位系统
float feed_time = LLA_WDG_GetOVRTime();//feed_time == 20s
```