**eBox标准API手册**

**打造eBox生态圈**

**版本：版本16.4.19**

**发布日期：2016年2月**

本指南内容及产如有更新，请参考最新手

勘误记录

|  |  |  |
| --- | --- | --- |
| 日期 | 摘要 | 提交者 |
| 2018.12.10 | 添加Stream类说明 | 好心情 |
| 2019.01.02 | 添加ADC类说明 | 好心情 |

**如何使用本API手册**

本API手册只提供了公共基础函数和类中public函数的解释。并没有对程序中所有的变量和宏作出详细解释。如果涉及到相关宏定义的内容请阅读程序代码中的注释。

本手册更新可能没有程序更新的快，部分内容可能有所书写错误，最终以头文件中的内容为准。

本手册只作为引导性学习使用，请多读程序头文件已获得更全面的信息。

完整的应用示例可参考example文件夹下的内容

如发现有文字性错误或者部分代码错误请联系995207301@qq.com。

# 目录

[目录 1](#_Toc3323151)

[第1章 公共基础接口 1](#_Toc3323152)

[1.1 ebox\_init(void); 1](#_Toc3323153)

[1.2 millis( void ) ; 2](#_Toc3323154)

[1.3 delay\_ms(uint64\_t ms); 2](#_Toc3323155)

[1.4 delay\_us(uint64\_t us); 3](#_Toc3323156)

[1.5 shift\_out(GPIO\* data\_pin, GPIO\* clock\_pin, 3](#_Toc3323157)

[1.6 shift\_in(GPIO\* data\_pin, GPIO\* clock\_pin, uint8\_t bit\_order) 4](#_Toc3323158)

[1.7 uint32\_t chip\_id[3]; 4](#_Toc3323159)

[1.8 uint16\_t flash\_size; 4](#_Toc3323160)

[第2章 GPIO类 5](#_Toc3323161)

[2.1 GPIO(GPIO\_TypeDef \*port, uint16\_t pin); 5](#_Toc3323162)

[2.2 mode(); 6](#_Toc3323163)

[2.3 set(); 6](#_Toc3323164)

[2.4 reset(); 6](#_Toc3323165)

[2.5 write(); 7](#_Toc3323166)

[2.6 toggle(); 7](#_Toc3323167)

[2.7 read(uint8\_t\* val); 8](#_Toc3323168)

[2.8 read(void); 8](#_Toc3323169)

[第3章 ADC类 9](#_Toc3323170)

[3.1 add\_ch(Gpio \*io) 9](#_Toc3323171)

[3.2 add\_temp\_senser() 9](#_Toc3323172)

[3.3 begin() 9](#_Toc3323173)

[第4章 PARALLEL\_GPIO类 10](#_Toc3323174)

[4.1 GPIO \*bit[8]; 10](#_Toc3323175)

[4.2 mode(PIN\_MODE mode); 10](#_Toc3323176)

[4.3 write(uint8\_t data); 11](#_Toc3323177)

[4.4 read(void); 11](#_Toc3323178)

[第5章 PWM类 13](#_Toc3323179)

[5.1 PWM(GPIO\* pwm\_pin); 13](#_Toc3323180)

[5.2 begin(uint32\_t frq,uint16\_t duty); 14](#_Toc3323181)

[5.3 set\_frq(uint32\_t frq); 14](#_Toc3323182)

[5.4 set\_duty(uint16\_t duty); 15](#_Toc3323183)

[5.5 set\_oc\_polarity(uint8\_t flag); 15](#_Toc3323184)

[第6章 IN\_CAPTURE类 16](#_Toc3323185)

[6.1 IN\_CAPTURE(GPIO \*capture\_pin); 16](#_Toc3323186)

[6.2 begin(uint16\_t prescaler); 17](#_Toc3323187)

[6.3 set\_count(uint16\_t count); 17](#_Toc3323188)

[6.4 set\_polarity\_falling(); 18](#_Toc3323189)

[6.5 set\_polarity\_rising(); 18](#_Toc3323190)

[6.6 overflow\_event\_process(); 19](#_Toc3323191)

[6.7 get\_capture(); 19](#_Toc3323192)

[6.8 get\_overflow\_state(); 20](#_Toc3323193)

[6.9 attch\_ic\_interrupt(void(\*callback)(void)); 20](#_Toc3323194)

[6.10 attch\_update\_interrupt(void(\*callback)(void)); 21](#_Toc3323195)

[第7章 EXTI类 22](#_Toc3323196)

[7.1 EXTIx(GPIO \*EXTI\_pin,EXTITrigger\_TypeDef trigger); 22](#_Toc3323197)

[7.2 begin(); 23](#_Toc3323198)

[7.3 attach\_interrupt(void (\*callbackFun)(void)); 23](#_Toc3323199)

[7.4 interrupt(FunctionalState enable); 23](#_Toc3323200)

[第8章 USART类 25](#_Toc3323201)

[8.1 USART(USART\_TypeDef \* USARTx, 25](#_Toc3323202)

[8.2 begin(uint32\_t baud\_rate); 26](#_Toc3323203)

[8.3 begin(uint32\_t baud\_rate,uint8\_t data\_bit,uint8\_t parity,float stop\_bit); 26](#_Toc3323204)

[8.4 attach\_rx\_interrupt(void (\*callback\_fun)(void)); 27](#_Toc3323205)

[8.5 attach\_tx\_interrupt(void (\*callback\_fun)(void)); 27](#_Toc3323206)

[8.6 receive(); 28](#_Toc3323207)

[8.7 put\_char(char ch); 28](#_Toc3323208)

[8.8 put\_string(const char \*str); 29](#_Toc3323209)

[8.9 printf(const char\* fmt,...); 29](#_Toc3323210)

[8.10 printf\_length(const char \*str,uint16\_t length); 29](#_Toc3323211)

[8.11 wait\_busy() 30](#_Toc3323212)

[第9章 GTIMER类 31](#_Toc3323213)

[9.1 TIM(TIM\_TypeDef\* TIMx); 31](#_Toc3323214)

[9.2 begin(uint32\_t frq); 32](#_Toc3323215)

[9.3 attach\_interrupt(void(\*callback)(void)); 32](#_Toc3323216)

[9.4 interrupt(FunctionalState enable); 32](#_Toc3323217)

[9.5 start(void); 33](#_Toc3323218)

[9.6 stop(void); 33](#_Toc3323219)

[9.7 void reset\_frq(uint32\_t frq); 34](#_Toc3323220)

[9.8 set\_reload(uint16\_t autoreload); 34](#_Toc3323221)

[9.9 clear\_count(void); 34](#_Toc3323222)

[第10章 TIMERONE类 36](#_Toc3323223)

[第11章 SPI类 37](#_Toc3323224)

[11.1 SPI(SPI\_TypeDef \*SPIx,GPIO \*sck,GPIO \*miso,GPIO \*mosi); 37](#_Toc3323225)

[11.2 begin (SPI\_CONFIG\_TYPE\* spi\_config); 38](#_Toc3323226)

[11.3 config(SPI\_CONFIG\_TYPE\* spi\_config); 38](#_Toc3323227)

[11.4 read\_config(void); 39](#_Toc3323228)

[11.5 write(uint8\_t data); 39](#_Toc3323229)

[11.6 write(uint8\_t \*data,uint16\_t data\_length); 39](#_Toc3323230)

[11.7 read(); 40](#_Toc3323231)

[11.8 read(uint8\_t\* recv\_data); 40](#_Toc3323232)

[11.9 read(uint8\_t \*recv\_data,uint16\_t datalength); 41](#_Toc3323233)

[11.10 take\_spi\_right(SPI\_CONFIG\_TYPE\* spi\_config); 41](#_Toc3323234)

[11.11 release\_spi\_right(void); 41](#_Toc3323235)

[第12章 SOFTSPI类 42](#_Toc3323236)

[12.1 SOFTSPI(GPIO\* sck,GPIO\* miso,GPIO\* mosi); 43](#_Toc3323237)

[12.2 begin(SPI\_CONFIG\_TYPE\* spi\_config); 44](#_Toc3323238)

[12.3 config(SPI\_CONFIG\_TYPE\* spi\_config); 44](#_Toc3323239)

[12.4 read\_config(void); 44](#_Toc3323240)

[12.5 write(uint8\_t data); 44](#_Toc3323241)

[12.6 write(uint8\_t \*data,uint16\_t dataln); 44](#_Toc3323242)

[12.7 read(); 44](#_Toc3323243)

[12.8 read(uint8\_t\* data); 44](#_Toc3323244)

[12.9 read(uint8\_t \*rcvdata,uint16\_t dataln); 44](#_Toc3323245)

[12.10 take\_spi\_right(SPI\_CONFIG\_TYPE\* spi\_config); 44](#_Toc3323246)

[12.11 release\_spi\_right(void); 44](#_Toc3323247)

[第13章 I2C类 45](#_Toc3323248)

[13.1 I2C(I2C\_TypeDef\* I2Cx,GPIO\* scl\_pin,GPIO\* sda\_pin); 46](#_Toc3323249)

[13.2 begin(uint32\_t speed); 46](#_Toc3323250)

[13.3 config(uint32\_t speed); 47](#_Toc3323251)

[13.4 read\_config(); 47](#_Toc3323252)

[13.5 write\_byte(uint8\_t slave\_address, 47](#_Toc3323253)

[13.6 write\_byte(uint8\_t slave\_address, 48](#_Toc3323254)

[13.7 read\_byte (uint8\_t slave\_address, 48](#_Toc3323255)

[13.8 read\_byte (uint8\_t slave\_address, 49](#_Toc3323256)

[13.9 wait\_dev\_busy(uint8\_t slave\_address); 50](#_Toc3323257)

[13.10 take\_i2c\_right(uint32\_t speed); 50](#_Toc3323258)

[13.11 release\_i2c\_right(void); 50](#_Toc3323259)

[第14章 SOFTI2C类 52](#_Toc3323260)

[14.1 SOFTI2C(GPIO\* scl, GPIO\* sda); 53](#_Toc3323261)

[14.2 begin(uint32\_t speed); 53](#_Toc3323262)

[14.3 config(uint32\_t speed); 54](#_Toc3323263)

[14.4 其他公有成员函数 54](#_Toc3323264)

[第15章 RTC类 55](#_Toc3323265)

[15.1 begin(); 56](#_Toc3323266)

[15.2 attach\_interrupt(uint16\_t event, 56](#_Toc3323267)

[15.3 interrupt(uint32\_t bits,FunctionalState x); 57](#_Toc3323268)

[15.4 set\_counter(uint32\_t count); 57](#_Toc3323269)

[15.5 set\_alarm(uint32\_t count); 58](#_Toc3323270)

[15.6 get\_counter(); 58](#_Toc3323271)

[15.7 set\_time\_HMS(uint8\_t h,uint8\_t m,uint8\_t s); 58](#_Toc3323272)

[15.8 set\_alarm(uint8\_t h,uint8\_t m,uint8\_t s); 59](#_Toc3323273)

[15.9 get\_time\_HMS(uint8\_t\* h,uint8\_t\* m,uint8\_t\* s); 59](#_Toc3323274)

[15.10 RTC下的HHMMSS时钟变量 60](#_Toc3323275)

[第16章 FLASHCLASS类 61](#_Toc3323276)

[16.1 read(uint32\_t iAddress, uint8\_t \*buf, int32\_t iNbrToRead) ; 61](#_Toc3323277)

[16.2 write(uint32\_t iAddress, uint8\_t \*buf, uint32\_t iNbrToWrite); 62](#_Toc3323278)

[第17章 EBOX\_IWDG类 62](#_Toc3323279)

[17.1 EBOX\_IWDG(); 62](#_Toc3323280)

[17.2 begin(uint16\_t ms); 63](#_Toc3323281)

[17.3 feed(); 63](#_Toc3323282)

[第18章 Stream类 64](#_Toc3323283)

[18.1 available(void) 64](#_Toc3323284)

[18.2 read(void) 64](#_Toc3323285)

[18.3 peak(void) 64](#_Toc3323286)

[18.4 findUntil(char \*target, size\_t targetLen, char \*terminate, size\_t termLen); 65](#_Toc3323287)

[18.5 find(char \*target); 65](#_Toc3323288)

[18.6 find(char \*target, size\_t length); 65](#_Toc3323289)

[18.7 find(char target)； 65](#_Toc3323290)

[18.8 findUntil(char \*target, char \*terminator); 65](#_Toc3323291)

[18.9 parseInt(LookaheadMode lookahead = SKIP\_ALL, char ignore = NO\_IGNORE\_CHAR); 65](#_Toc3323292)

[18.10 parseFloat(LookaheadMode lookahead = SKIP\_ALL, char ignore = NO\_IGNORE\_CHAR); 66](#_Toc3323293)

[18.11 readBytes( char \*buffer, size\_t length); 66](#_Toc3323294)

[18.12 readBytesUntil( char terminator, char \*buffer, size\_t length); 66](#_Toc3323295)

# 公共基础接口

ebox\_init(void);

millis( void ) ;

delay\_ms(uint64\_t ms);

delay\_us(uint64\_t us);

analog\_read(GPIO\* pin);

analog\_read\_voltage(GPIO\* pin);

analog\_write(GPIO\* pwm\_pin, uint16\_t duty)

shift\_out(GPIO\* data\_pin, GPIO\* clock\_pin, uint8\_t bit\_order, uint8\_t val);

shift\_in(GPIO\* data\_pin, GPIO\* clock\_pin, uint8\_t bit\_order);

////////////////////宏定义/////////////////////////////

#define true 0x1

#define false 0x0

#define HIGH 0x1

#define LOW 0x0

#define LSB\_FIRST 0

#define MSB\_FIRST 1

#define PI 3.1415926535898

#define NVIC\_GROUP\_CONFIG NVIC\_PriorityGroup\_2

#define interrupts() \_\_enable\_irq()

#define no\_interrupts() \_\_disable\_irq()

MCU信息类

class SYSTEM

{

public:

uint32\_t chip\_id[3];

uint16\_t flash\_size;

void get\_chip\_info();

float get\_cpu\_temperature();

};

## ebox\_init(void);

### 描述：

初始化eBox运行需要的基础条件，包括ADC1，systicks，NVIC分组，PB4默认功能为GPIO。

### 参数：

### 返回值：

### 示例：

## millis( void ) ;

### 描述：

返回milli\_seconds当前值，milli\_seconds从eBox程序一开始运行便开始自动的每一毫秒加1，milli\_seconds计数到2^64后会溢出（变成0）。

### 参数：

空

### 返回值：

返回milli\_seconds当前值。

### 示例：

uint64\_t time;

time = millis();

## delay\_ms(uint64\_t ms);

### 描述：

精确的毫秒延时函数，使用滴答时钟完成的硬件计时。相当精确。

### 参数：

**ms**:延时的时间。最大值2^64。

### 返回值：

### 返回值：

### 示例：

delay\_ms(1000);

## delay\_us(uint64\_t us);

### 描述：

精确的微秒延时函数，使用滴答时钟完成的硬件计时。相当精确。

### 参数：

us:延时的时间。最大值2^64。

### 返回值：

空

### 示例：

delay\_us(1);

## shift\_out(GPIO\* data\_pin, GPIO\* clock\_pin,

**uint8\_t bit\_order, uint8\_t val)**

### 描述：

将一个8位的数据按bit顺序输出到IO引脚上。

### 参数：

**data\_pin：**GPIO的指针或地址

**clock\_pin：**GPIO的指针或地址

**bit\_order：**是高位先输出还是低位先输出。

**Val：**输出的值

### 返回值：

空

### 示例：

shift\_out(PA1，PA2，LSBFIRST,0X5A);

## shift\_in(GPIO\* data\_pin, GPIO\* clock\_pin, uint8\_t bit\_order)

### 描述：

读取一个IO上的8位的数据按bit顺序组合成一个8位数据

### 参数：

**data\_pin：**GPIO的指针或地址

**clock\_pin：**GPIO的指针或地址

**bit\_order：**是高位先输出还是低位先输出。

### 返回值：

**data\_pin**引脚读入的数据。

### 示例：

uin8\_t val；

val= shift\_in(PA1，PA2，LSBFIRST);

## uint32\_t chip\_id[3];

### 描述：

STM32的芯片ID；在eBox的初始化函数中已经被更新完成，用户可以直接使用。

## uint16\_t flash\_size;

### 描述：

STM32芯片的flash容量；在eBox的初始化函数中已经被更新完成，用户可以直接使用。

# GPIO类

class GPIO

{

public:

GPIO(GPIO\_TypeDef \*port, uint16\_t pin);

void mode(PIN\_MODE mode);

void set();

void reset();

void write(uint8\_t val);

void toggle();

void read(uint8\_t \*val);

uint8\_t read(void);

GPIO\_TypeDef\* port;

uint16\_t pin;

};

## GPIO(GPIO\_TypeDef \*port, uint16\_t pin);

### 描述：

GPIO的构造函数，实例化一个GPIO引脚（创建GPIO对象）。

### 参数：

**port：**GPIO所在的port，如GPIOA；

**pin：**GPIO引脚的掩码，如GPIO\_Pin\_1；

### 返回值：

空

### 示例：

GPIO PA1(GPIOA,GPIO\_Pin\_1);

#define PPA1 ((GPIO\*)&GPA1)

GPIO \*PPA1 = new GPIO(GPIOA,GPIO\_Pin\_1);

## mode();

### 描述：

GPIO引脚模式设置

### 参数：

**mode：PIN\_MODE 枚举变量**，如INPUT,OUTPUT,AF\_PP,AF\_OD等；

### 返回值：

### 示例：

PA1->mode(OUTPUT);

## set();

### 描述：

GPIO引脚输出高电平。执行效率最高。和官方固件库速度一样

### 参数：

空

### 返回值：

空

### 示例：

PA1.mode(OUTPUT\_PP);

PA1.set();

## reset();

### 描述：

GPIO引脚输出低电平。执行效率最高。和官方固件库速度一样

### 参数：

空

### 返回值：

空

### 示例：

PA1.mode(OUTPUT\_PP);

PA1.reset();

## write();

### 描述：

GPIO引脚输出控制。执行效率略低于set()和reset()。5个指令周期。

### 参数：

空

### 返回值：

空

### 示例：

PA1.mode(OUTPUT\_PP);

PA1.write(1);

PA1.write(0);

PA1.write(HIGH);

PA1.write(LOW);

## toggle();

### 描述：

GPIO引脚输出翻转。

### 参数：

空

### 返回值：

空

### 示例：

PA1.mode(OUTPUT\_PP);

PA1.toggle();

## read(uint8\_t\* val);

### 描述：

读取GPIO引脚输入电平。

### 参数：

**val:**输出uint8\_t指针

### 返回值：空

### 示例：

uint8\_t val;

PA1.mode(INPUT);

PA1.read(&val);

## read(void);

### 描述：

读取GPIO引脚输入电平。

### 参数：空

### 空返回值：

**val:**GPIO引脚电平

### 示例：

uint8\_t val;

PA1.mode(INPUT);

val = PA1.read();

# ADC类

ADC类设计为DMA模式，软件触发，连续采样,分辨率12bit。

如何使用ADC类：

1 创建对象 Adc adc(ADC1);

2 添加通道 adc.add\_ch(&PA0);

adc.add\_ch(&PA1);

3 启动ADC adc.begin();

4 读取ADC adc.read(&PA0);

adc.read\_voltage(&PA0)

## add\_ch(Gpio \*io)

### 描述：

通过gpio添加1个adc通道

### 参数：

Gpio \*io GPIO对象指针或地址，如&PA7；

### 返回值：

空

## add\_temp\_senser()

添加temp通道

### 参数：

无

### 返回值：

空

## begin()

启动ADC

### 参数：

无

### 返回值：

空

# PARALLEL\_GPIO类

class PARALLEL\_GPIO

{

public:

GPIO \*bit[8];

public:

void mode(PIN\_MODE mode);

void write(uint8\_t data);

uint8\_t read();

};

## GPIO \*bit[8];

### 描述：

并行IO口指针，用于传递要使用的1-8位GPIO口

### 示例

PARALLEL\_GPIO P1;

P1.bit[0] = &PA1;

P1.bit[1]=&PB2;……

## mode(PIN\_MODE mode);

### 描述：

GPIO引脚模式设置

### 参数：

**mode：PIN\_MODE 枚举变量**，如INPUT,OUTPUT,AF\_PP,AF\_OD等；

### 返回值：

### 示例：

PARALLEL\_GPIO P1;

P1.mode(OUTPUT);

## write(uint8\_t data);

### 描述：

并行GPIO引脚输出控制。

### 参数：

data：并行GPIO端口的值。

### 返回值：

空

### 示例：

P1.write(0xff);

P1.write(0x55);

## read(void);

### 描述：

读取GPIO引脚输入电平。

### 参数：

空

### 空返回值：

**val:**GPIO引脚电平

### 示例：

uint8\_t val;

val = P1.read();

# PWM类

class PWM

{

public:

PWM(GPIO \*pwm\_pin);

void begin(uint32\_t frq,uint16\_t duty);

void set\_frq(uint32\_t frq);

void set\_duty(uint16\_t duty);

void set\_oc\_polarity(uint8\_t flag);

private:

GPIO \*pwm\_pin;

TIM\_TypeDef \*TIMx;

uint32\_t rcc;

uint8\_t ch;

uint16\_t period;//保存溢出值，用于计算占空比

uint16\_t duty;//保存占空比值

uint16\_t oc\_polarity;

void init\_info(GPIO \*pwm\_pin);

void base\_init(uint16\_t Period,uint16\_t Prescaler);

};

## PWM(GPIO\* pwm\_pin);

### 描述：

PWM的构造函数，实例化一个PWM（创建··PWM对象）。初始化某个引脚为PWM输出引脚。

### 参数：

**pwm\_pin：**GPIO对象指针或地址，如&PA7；

### 返回值：

空

### 示例：

PWM pwm(&PB8);

## begin(uint32\_t frq,uint16\_t duty);

### 描述：

PWM的初始化函数，自动开启相应的定时器，并设置相应的参数。

### 参数：

**frq：当前定时器**所有pwm引脚的频率。最小值1（1Hz）.最大值720K（当频率小于72000Hz时，pwm精度为0.1%；频率在72K~720K之间时，pwm精度为1%；频率再高将停止输出）；

**duty：当前pwm引脚**的占空比；

### 返回值：

空

### 示例：

初始化为10K频率，占空比为50%的PWM波。

PWM pwm(&PB8);

pwm.begin(10000,500);

## set\_frq(uint32\_t frq);

### 描述：

重新设定PWM波形频率，创建PWM对象后可调用，只改变波形频率不改变波形占空比。

### 参数：

**frq：当前定时器**所有pwm引脚的频率。最小值1（1Hz）.最大值720K（当频率小于72000Hz时，pwm精度为0.1%；频率在72K~720K之间时，pwm精度为1%；频率再高将停止输出）；

### 返回值：

空

### 示例：

pwm.set\_frq(1000);

## set\_duty(uint16\_t duty);

### 描述：

重新设定PWM波形占空比，创建PWM对象后可调用，只改变波形占空比不改变波形频率。

### 参数：

**duty：当前pwm引脚**的占空比；

### 返回值：

空

### 示例：

pwm.set\_duty(500);

## set\_oc\_polarity(uint8\_t flag);

### 描述：

设置PWM输出极性。

### 参数：

**flag：**1：比较后输出高电平，0：比较后输出低电平；

### 返回值：

空

### 示例：

pwm1.set\_oc\_polarity(1);//设置比较后输出高电平

# IN\_CAPTURE类

class IN\_CAPTURE

{

public:

IN\_CAPTURE(GPIO \*capture\_pin);

void begin(uint16\_t prescaler);

void set\_count(uint16\_t count);

void set\_polarity\_falling();

void set\_polarity\_rising();

void overflow\_event\_process();

uint32\_t get\_capture();

IC\_OVERFLOW\_STATE\_TYEP get\_overflow\_state();

void attch\_ic\_interrupt(void(\*callback)(void));

void attch\_update\_interrupt(void(\*callback)(void));

private:

GPIO \*capture\_pin;

TIM\_TypeDef \*TIMx;

uint8\_t ch;

uint16\_t period;//保存溢出值，用于计算占空比

uint16\_t prescaler;//保存时钟分频值，用于计算占空比

uint16\_t overflow\_times;//溢出次数

IC\_OVERFLOW\_STATE\_TYEP overflow\_state;//溢出的情况。如果发生16位溢出：IC\_OVERFLOW，如果发送32位溢出:IC\_FAULT

uint8\_t polarity;

void init\_info(GPIO \*capture\_pin);

void base\_init(uint16\_t Period,uint16\_t Prescaler);

uint16\_t (\*\_get\_capture)(TIM\_TypeDef \*TIMx);

void (\*\_set\_polarity)(TIM\_TypeDef \*TIMx,uint16\_t TIM\_OCPolarity);//设置为下降沿捕获

};

## IN\_CAPTURE(GPIO \*capture\_pin);

### 描述：

IN\_CAPTURE的构造函数，实例化一个IN\_CAPTURE对象。初始化某个引脚为IN\_CAPTURE输入捕获引脚。该引脚必须是定时器2,3,4的某个通道之一。

### 参数：

**capture \_pin：**GPIO对象指针或地址，如&PA7；

### 返回值：

空

### 示例：

IN\_CAPTURE capture (&PB8);

## begin(uint16\_t prescaler);

### 描述：

IN\_CAPTURE的初始化函数。将定时器的分频系数填入，计数器默认值为0xffff。

### 参数：

prescaler**：**分频系数，定时器频率的计算方法，72M/prescaler；

### 返回值：

空

### 示例：

capture .begin(1);//一分频

## set\_count(uint16\_t count);

### 描述：

设置定时器计数器的值，一般用于捕获中断中，清零计数器。

### 参数：

count**：**计数器的值：0~0xffff；

### 返回值：

空

### 示例：

capture.set\_count (0);

## set\_polarity\_falling();

### 描述：

设置捕获下降沿产生中断。

### 参数：

**空**；

### 返回值：

空

### 示例：

IN\_CAPTURE capture (&PB8);

## set\_polarity\_rising();

### 描述：

设置捕获上升沿产生中断。

### 参数：

**空**；

### 返回值：

空

### 示例：

void mesure\_duty()//输入捕获中断事件

{

ic.set\_count(0);

if(ic.polarity == TIM\_ICPOLARITY\_FALLING)//测量高电平时间完成

{

value1 = ic.get\_capture() + 170;//校正值，查表可得

**ic.set\_polarity\_rising();//切换至测量低电平时间完成**

}

else//测量低电平时间完成

{

value2 = ic.get\_capture() + 170;//校正值，查表可得

**ic.set\_polarity\_falling();//切换至测量高电平时间完成**

}

}

## overflow\_event\_process();

### 描述：

溢出中断处理函数，为了使该模块能测量更长脉冲宽度的脉冲，增加溢出中断处理，此函数放到定时器溢出中断里面即可。

### 参数：

**空**；

### 返回值：

空

### 示例：

void update\_event()

{

ic.overflow\_event\_process();

}

## get\_capture();

### 描述：

获取在脉冲宽度之间的计数器的计数值，如果用户使用了溢出中断处理函数，会将溢出次数乘以0xffff再加上当前计数器的值返回。

### 参数：

**空**；

### 返回值：

空

### 示例：

void mesure\_frq()//  
{

ic.set\_count(0);

value1 = ic.get\_capture() + 170;

}

## get\_overflow\_state();

### 描述：

获取定时器的溢出状态。

### 参数：

**溢出状态：**0：没有溢出发生，1：代表溢出，2：代表溢出次数超过0xff；如果状态是2，则表示当前的分频系数过低，请调大分频次数。在测量时间不超过60s的情况下不用使用此函数。

### 返回值：

空

### 示例：

State = ic.get\_overflow\_state ();

If(state == 2)

Return err;

## attch\_ic\_interrupt(void(\*callback)(void));

### 描述：

绑定捕获中断函数。

### 参数：

callback**：**用户函数入口指针；

### 返回值：

空

### 示例：

ic.attch\_ic\_interrupt(mesure\_frq);

## attch\_update\_interrupt(void(\*callback)(void));

### 描述：

绑定捕获中断函数。

### 参数：

callback**：**用户函数入口指针；

### 返回值：

空

### 示例：

ic.attch\_update\_interrupt(update\_event);

# EXTI类

class EXTIx

{

public:

EXTIx(GPIO \*exti\_pin, EXTITrigger\_TypeDef trigger);

void begin();

void attach\_interrupt(void (\*callback\_fun)(void));

void interrupt(FunctionalState enable);

private:

GPIO \*exti\_pin;

EXTITrigger\_TypeDef trigger;

uint8\_t port\_source;

uint8\_t pin\_source;

uint32\_t exti\_line;

uint8\_t irq;

void init\_info(GPIO \*exti\_pin);

};

## EXTIx(GPIO \*EXTI\_pin,EXTITrigger\_TypeDef trigger);

### 描述：

EXTIx的构造函数，实例化一个EXTIx（创建EXTIx对象）。

### 参数：

**EXTI\_pin**：GPIO对象指针或地址，如&PA5；

**trigger**:EXTITrigger\_TypeDef类型枚举变量；如：EXTI\_Trigger\_Falling

### 返回值：

空；

### 示例：

EXTIx ex(&PA8,EXTI\_Trigger\_Falling);

## begin();

### 描述：

EXTIx初始化。

### 参数：

空

### 返回值：

空

### 示例：

EXTIx ex(&PA8,EXTI\_Trigger\_Falling);

ex.begin();

## attach\_interrupt(void (\*callbackFun)(void));

### 描述：

外部中断绑定回调函。

### 参数：

**(\*callbackFun)(void)：**中断回调函数指针（函数名），如void exti\_it\_event(void)的函数名exti\_it\_event。

### 返回值：

空

### 示例：

exti.attach\_interrupt(exti\_it\_event);

## interrupt(FunctionalState enable);

### 描述：

外部中断使能控制。

### 参数：

**enable**：ENABLE/DISABLE。

### 返回值：

空

### 示例：

ex.interrupt(ENABLE);

# USART类

class USART

{

public:

USART(USART\_TypeDef \*USARTx,GPIO \*tx\_pin,GPIO \*rx\_pin);

void begin(uint32\_t baud\_rate);

void begin(uint32\_t baud\_rate,uint8\_t data\_bit,uint8\_t parity,float stop\_bit);

void attach\_rx\_interrupt(void (\*callback\_fun)(void));

void attach\_tx\_interrupt(void (\*callback\_fun)(void));

int put\_char(uint16\_t ch);

void put\_string(const char \*str);

void printf\_length(const char \*str,uint16\_t length);

void printf(const char \*fmt,...);

void wait\_busy();

uint16\_t receive();

private:

USART\_TypeDef \*\_USARTx;

DMA\_Channel\_TypeDef \*\_DMA1\_Channelx;

char send\_buf[UART\_MAX\_SEND\_BUF];

uint16\_t dma\_send\_string(const char \*str,uint16\_t length);

void put\_string(const char \*str,uint16\_t length);

void set\_busy();

void interrupt(FunctionalState enable);

};

## USART(USART\_TypeDef \* USARTx,

**GPIO\* tx\_pin,GPIO\* rx\_pin);**

### 描述：

USART的构造函数，实例化一个串口（创建串口对象）。

### 参数：

**USARTx：**USART\_TypeDef类型值，如USART1、2、3、4、5；

**tx\_pin：**GPIO对象指针或地址，如PA9,PA2,PB10；

**rx\_pin：**GPIO对象指针或地址，如PA10,PA3,PB11；

### 返回值：

空

### 示例：

uart1(USART1,&PA9,&PA10);

## begin(uint32\_t baud\_rate);

### 描述：

串口初始化函数。

### 参数：

**baud\_rate：**串口波特率，如9600，115200等；

### 返回值：

空

### 示例：

uart1.begin(9600);

## begin(uint32\_t baud\_rate,uint8\_t data\_bit,uint8\_t parity,float stop\_bit);

### 描述：

串口初始化函数。

### 参数：

**baud\_rate：**串口波特率，如9600，115200等；

data\_bit：数据位，例如8

parity：校验位，例如0：无校验、1：奇校验、2：偶校验

stop\_bit：停止位，例如1、1.5、2

### 返回值：

空

### 示例：

uart1.begin(9600);

## attach\_rx\_interrupt(void (\*callback\_fun)(void));

### 描述：

串口中断绑定回调函。

### 参数：

**(\*callback\_fun)(void)：**中断回调函数指针（名），如void uar\_it\_rx(void)的函数名uar\_it\_rx。

### 返回值：

空

### 示例：

uart1.attach\_interrupt(uar\_it\_rx);

## attach\_tx\_interrupt(void (\*callback\_fun)(void));

### 描述：

串口中断绑定回调函。

### 参数：

**(\*callback\_fun)(void)：**发送完成中断回调函数指针（名），如void uar\_it\_tx(void)的函数名uar\_it\_tx。

### 返回值：

空

### 示例：

uart1.attach\_interrupt(uar\_it\_tx);

## receive();

### 描述：

读取串口接收到的数据，一般应用于串口接收中断事件中。

### 参数：

空；

### 返回值：

u16类型串口接收到的数据；(串口数据格式有9位数据的模式，所以使用u16类型，一般情况下使用的是u8类型的数据)

### 示例：

uint8\_t char;

char = receive();

## put\_char(char ch);

### 描述：

串口输出一个字节。

### 参数：

**ch：**char类型参数。

### 返回值：

空

### 示例：

uart1.put\_ char (‘1’);

uart1.put\_char(0x31);

## put\_string(const char \*str);

### 描述：

串口输出一个字符串。遇到’\0’将会停止输出

### 参数：

**str：**char类型指针。

### 返回值：空

### 示例：

uart1.put\_string(“hellow world”);

## printf(const char\* fmt,...);

### 描述：

串口格式化输出字符串。使用方法和标准c中的一样。

### 参数：

fmt：字符串

...：变量参数

### 返回值：空

### 示例：

uart1.printf(“hellow world”);

uart1.printf(“intVal = %03d”,x);

uart1.printf(“floatVal = %f”,f);

## printf\_length(const char \*str,uint16\_t length);

### 描述：

串口输出指定长度字符串。这个主要针对发送特定长度的缓冲区，中间有‘/0’也可以继续发送。

### 参数：

**str：**char类型指针。

**length：**输出字符串长度。

### 返回值：

空

### 示例：

uart1.putString(“hellow world”,10);

## wait\_busy()

### 描述：

等待串口输出完成。这个函数的使用概率是非常低的，除非是你必须需要等待串口发送完成才能执行下面的函数才需要使用这个函数。

### 参数：

空

### 返回值：

空

### 示例：

uart1.put\_string(“hellow world”,10);

uart1.wait\_busy();

# GTIMER类

class TIM

{

public:

TIM(TIM\_TypeDef \*TIMx);

void begin(uint32\_t frq);

void attach\_interrupt(void(\*callback)(void));

void interrupt(FunctionalState enable);

void start(void);

void stop(void);

void reset\_frq(uint32\_t frq);

private:

void base\_init(uint16\_t period,uint16\_t prescaler);

void set\_reload(uint16\_t auto\_reload);

void clear\_count(void);

TIM\_TypeDef \*\_TIMx;

};

## TIM(TIM\_TypeDef\* TIMx);

### 描述：

TIM的构造函数，实例化一个TIM（创建TIM对象）。

### 参数：

**TIMx**：TIM\_TypeDef对象指针或地址，如TIM2、3、4、5、6、7,；

注意：TIMx目前只支持通用定时器，不支持高级定时器。如果使用了PWM请注意pwm引脚是否基于要使用的定时器，如果是的话最好避让该定时器，否则会改变PWM引脚的频率和占空比。

### 返回值：

空

### 示例：

TIM(TIM2);

## begin(uint32\_t frq);

### 描述：

初始化定时器参数，创建通用定时器对象后可调用；

### 参数：

**frq**：定时器的中断频率，取值范围1-1000000。

**返回值：空**

### 示例：

tim2.begin(1000);

## attach\_interrupt(void(\*callback)(void));

### 描述：

定时器溢出中断绑定回调函。

### 参数：

**(\*callbackFun)(void)：**中断回调函数指针（名），如void timer2It(void)的函数名timer2It。

### 返回值：

空

### 示例：

tim2.attach\_interrupt(timer2It);

## interrupt(FunctionalState enable);

### 描述：

通用定时器溢出中断使能。

### 参数：

**enable：**FunctionalState类型值，如ENABLE,DISABLE。

### 返回值：

空

### 示例：

tim2.interrupt(ENABLE);

## start(void);

### 描述：

启动通用定时器。初始化完成定时器，调用此函数才能启动定时器。

### 参数：

空；

### 返回值：

空;

### 示例：

tim2.start();

## stop(void);

### 描述：

停止通用定时器。停止定时器计时，可调用start再次启动。

### 参数：

空；

### 返回值：

空;

### 示例：

tim2.stop();

## void reset\_frq(uint32\_t frq);

### 描述：

重新初始化定时器参数。

### 参数：

**frq**：定时器的中断频率，取值范围1-1000000。

### 返回值：

空；

### 示例：

begin(1000);

## set\_reload(uint16\_t autoreload);

### 描述：

修改溢出值。

### 参数：

autoreload：定时器溢出值；

### 返回值：

空；

### 示例：

tim2.set\_reload(1024);

## clear\_count(void);

### 描述：

定时器计数器清理。

### 参数：

空；

### 返回值：

空；

### 示例：

tim2.clear\_count();

# TIMERONE类

class TIMERONE

{

public:

TIMERONE();

void begin(uint32\_t frq);

void attach\_interrupt(void(\*callback)(void));

void interrupt(FunctionalState x);

void start();

void stop();

void reset\_frq(uint32\_t frq);

private:

void base\_init(uint16\_t period,uint16\_t prescaler);

void set\_reload(uint16\_t autoreload);

void clear\_count(void);

};

注解:请参考通用定时器；

# SPI类

class SPIClass

{

public:

SPI(SPI\_TypeDef \*SPIx,GPIO \*sck,GPIO \*miso,GPIO \*mosi);

void begin (SPI\_CONFIG\_TYPE \*spi\_config);

void config(SPI\_CONFIG\_TYPE \*spi\_config);

uint8\_t read\_config(void);

int8\_t write(uint8\_t data);

int8\_t write(uint8\_t \*data,uint16\_t data\_length);

uint8\_t read();

int8\_t read(uint8\_t \*recv\_data);

int8\_t read(uint8\_t \*recv\_data,uint16\_t data\_length);

public:

int8\_t take\_spi\_right(SPI\_CONFIG\_TYPE \*spi\_config);

int8\_t release\_spi\_right(void);

private:

uint8\_t current\_dev\_num;

SPI\_TypeDef \*spi;

uint8\_t busy;

};

## SPI(SPI\_TypeDef \*SPIx,GPIO \*sck,GPIO \*miso,GPIO \*mosi);

### 描述：

### 参数：

**SPIx：**SPI\_TypeDef类型指针或地址，如SPI1，SPI2，SPI3；

**p\_sck\_pin：**GPIO对象指针或地址，如PA5；

**p\_miso\_pin：**GPIO对象指针或地址，如PA6；

**p\_mosi\_pin**:GPIO对象指针或地址，如PA7；

**注意**：如果使用SPI的冲映射引脚需要自己添加冲映射代码。硬件资源有限，只测试了SPI1。

### 返回值：

空

### 示例：

SPI spi1(SPI1,&PA5,&PA6,&PA7);

SPI spi2(SPI2,&PB13,&PB14,&PB15);

## begin (SPI\_CONFIG\_TYPE\* spi\_config);

### 描述：

SPI初始化。

### 参数：

**spi\_config：**SPICONFIG对象指针或地址；

### 返回值：

空

### 示例：

spi->begin(&spiDevW5500);

## config(SPI\_CONFIG\_TYPE\* spi\_config);

### 描述：

SPI更新配置。

### 参数：

spi\_config**：**SPI\_CONFIG\_TYPE对象指针或地址；

### 返回值：

空

### 示例：

spi1->config(SPI\_CONFIG);

## read\_config(void);

### 描述：

读取SPI配置号。

### 参数：

空；

### 返回值：

**currentDevNum**：spi处于某个芯片spi配置的编号；

### 示例：

spi->config(&SPIDevSDCard);

## write(uint8\_t data);

### 描述：

写一个字节的数据，发送并读取；

### 参数：

**data**:发送的数据；

### 返回值：

**spi**-**>dr**：接收到的数据；

### 示例：

spi1.write(data);

## write(uint8\_t \*data,uint16\_t data\_length);

### 描述：

写一个指定长度字节的数据；

### 参数：

**\*data**:传输的数据的指针；

**data\_length：**传输数据长度；

### 返回值：

**spi**-**>dr**：接收到的数据；

### 示例：

**spi1.write(data);**

## read();

### 描述：

读取一个指定长度字节的数据；

### 参数：

空

### 返回值：

**spi**-**>dr**：接收到的数据；

### 示例：

**spi1.read();**

## read(uint8\_t\* recv\_data);

### 描述：

读取一个指定长度字节的数据；

### 参数：

\*recv\_data：接收数据的指针。

### 返回值：

**0**：正常

### 示例：

spi1.read();

## read(uint8\_t \*recv\_data,uint16\_t datalength);

### 描述：

接收指定长度字节的数据**。**

### 参数：

**\*recv\_data**：传输的数据的指针；

**datalength**:传输数据长度；

### 返回值：

空；

### 示例：

spi1.read(&data,10);

## take\_spi\_right(SPI\_CONFIG\_TYPE\* spi\_config);

### 描述：

获取SPI使用权，并设置速度；**获取不到一直等待**。

### 参数：

spi\_config：spi的配置；

### 返回值：

0：正常；

### 示例：看下文

## release\_spi\_right(void);

### 描述：

释放SPI使用权。

### 参数：空

### 返回值：

0：正常；

### 示例：

spi->take\_spi\_right(&spiDevW5500);

cs->reset(); // CS=0, SPI start

spi->write( (addrbsb & 0x00FF0000)>>16);// Address byte 1

spi->write( (addrbsb & 0x0000FF00)>> 8);// Address byte 2

spi->write( (addrbsb & 0x000000F8) + 4); //

spi->write(data); // Data write (write 1byte data)

cs->set(); // CS=1, SPI end

spi->release\_spi\_right();

# SOFTSPI类

class SOFTSPI

{

public:

SOFTSPI(GPIO \*sck,GPIO \*miso,GPIO \*mosi);

void begin(SPI\_CONFIG\_TYPE \*spi\_config);

void config(SPI\_CONFIG\_TYPE \*spi\_config);

uint8\_t read\_config(void);

int8\_t write(uint8\_t data);

int8\_t write(uint8\_t \*data,uint16\_t data\_length);

uint8\_t read();

int8\_t read(uint8\_t \*data);

int8\_t read(uint8\_t \*rcvdata,uint16\_t data\_length);

public:

int8\_t take\_spi\_right(SPI\_CONFIG\_TYPE \*spi\_config);

int8\_t release\_spi\_right(void);

private:

GPIO \*sck\_pin;

GPIO \*mosi\_pin;

GPIO \*miso\_pin;

uint8\_t mode;

uint8\_t bit\_order;

uint8\_t spi\_delay;

uint8\_t current\_dev\_num;

uint8\_t busy;

uint8\_t transfer0(uint8\_t data);

uint8\_t transfer1(uint8\_t data);

uint8\_t transfer2(uint8\_t data);

uint8\_t transfer3(uint8\_t data);

uint8\_t transfer(uint8\_t data);

};

## SOFTSPI(GPIO\* sck,GPIO\* miso,GPIO\* mosi);

### 描述：

SOFTSPI的构造函数，实例化一个SOFTSPI（创建SOFTSPI）对象。此对象应该在object.cpp中创建，被驱动层调用。

### 参数：

**sck：**GPIO对象指针或地址，如&PA5；

**miso：**GPIO对象指针或地址，如&PA6；

**mosi**:GPIO对象指针或地址，如&PA7；

### 返回值：

空

### 示例：

SOFTSPI sspi(&PA5,&PA6,&PA7);

## begin(SPI\_CONFIG\_TYPE\* spi\_config);

## config(SPI\_CONFIG\_TYPE\* spi\_config);

## read\_config(void);

## write(uint8\_t data);

## write(uint8\_t \*data,uint16\_t dataln);

## read();

## read(uint8\_t\* data);

## read(uint8\_t \*rcvdata,uint16\_t dataln);

## take\_spi\_right(SPI\_CONFIG\_TYPE\* spi\_config);

## release\_spi\_right(void);

以上函数的方法和硬件SPI兼容

# I2C类

class I2C

{

public:

I2C(I2C\_TypeDef \*I2Cx,GPIO \*scl\_pin,GPIO \*sda\_pin);

void begin(uint32\_t speed);

void config(uint32\_t speed);

uint32\_t read\_config();

int8\_t write\_byte(uint8\_t slave\_address,uint8\_t reg\_address,uint8\_t data);

int8\_t write\_byte(uint8\_t slave\_address,uint8\_t reg\_address,uint8\_t \*data,uint16\_t num\_to\_write);

int8\_t read\_byte (uint8\_t slave\_address,uint8\_t reg\_address,uint8\_t \*data);

int8\_t read\_byte (uint8\_t slave\_address,uint8\_t reg\_address,uint8\_t \*data,uint16\_t num\_to\_read);

int8\_t wait\_dev\_busy(uint8\_t slave\_address);

public:

int8\_t take\_i2c\_right(uint32\_t speed);

int8\_t release\_i2c\_right(void);

private:

int8\_t start();

int8\_t stop();

int8\_t send\_no\_ack();

int8\_t send\_ack();

int8\_t send\_byte(uint8\_t regData);

int8\_t send\_7bits\_address(uint8\_t slave\_address);

int8\_t receive\_byte(uint8\_t \*data);

private:

I2C\_TypeDef \*I2Cx;

GPIO \*sda\_pin;

GPIO \*scl\_pin;

uint32\_t speed;

uint8\_t busy;

};

## I2C(I2C\_TypeDef\* I2Cx,GPIO\* scl\_pin,GPIO\* sda\_pin);

### 描述：

I2C的构造函数，实例化一个I2C（创建I2C对象）。此对象一般eBox默认提供两个对象i2c1和i2c2，在object.cpp中定义。

### 参数：

**I2Cx：**I2C\_TypeDef类型值，如I2C1，I2C2；

**sda\_pin：**GPIO对象指针或地址，如PB7；

**scl\_pin：**GPIO对象指针或地址，如PB6；

### 返回值：空

### 示例：

I2C i2c1(I2C1,&PB6,&PB7);

I2C i2c2(I2C2,&PB10,&PB11);

## begin(uint32\_t speed);

### 描述：

I2C初始化函数；

### 参数：

**speed：**I2C速度设置，如；100000,200000,300000,400000等；

### 返回值：

空

### 示例：

i2c.begin(100000);

## config(uint32\_t speed);

### 描述：

I2C速度设置函数；

### 参数：

**speed：**I2C速度设置，如；100000,200000,300000,400000；

### 返回值：

空

### 示例：

I2c1.config(100000);

## read\_config();

### 描述：

返回当前I2C的速度配置；此函数

### 参数：

**空**；

### 返回值：

i2c的速度配置：speed；

### 示例：

I2c1.read\_config();

## write\_byte(uint8\_t slave\_address,

**uint8\_t reg\_address,uint8\_t data);**

### 描述：

I2C器件写入一个字节；

### 参数：

slave\_address：I2C器件地址；

reg\_address：器件寄存器地址；

data：要写入的值；

### 返回值：

err：0表示正常；

### 示例：

i2c->write\_byte(SLAVEADDRESS,GYRO\_CONFIG,0x18);

## write\_byte(uint8\_t slave\_address,

**uint8\_t reg\_address,uint8\_t\* data,uint16\_t num\_to\_write);**

### 描述：

I2C器件写入指定长度字节；

### 参数：

slave\_address：I2C器件地址；

reg\_address：器件寄存器地址；

data：uint8\_t数据指针；

num\_to\_write：写入的数据长度；

### 返回值：

err：0表示正常；

### 示例：

uint8\_t tmp[10];

i2c->write\_byte(SLAVEADDRESS,SMPLRT\_DIV,tmp，10);

## read\_byte (uint8\_t slave\_address,

uint8\_t reg\_address,uint8\_t\* data);

### 描述：

I2C器件读取一个字节；

### 参数：

slave\_address：I2C器件地址；

reg\_address：器件寄存器地址；

data：输出数据的指针；

### 返回值：

err：0表示正常；

### 示例：

uint8\_t tmp;

i2c->read\_byte(SLAVEADDRESS,SMPLRT\_DIV,&tmp);

## read\_byte (uint8\_t slave\_address,

uint8\_t reg\_address,uint8\_t\* data,uint16\_t num\_to\_read);

### 描述：

I2C器件读取指定长度数据；

### 参数：

slave\_address：I2C器件地址；

reg\_address：器件寄存器地址；

data：输出数据的指针；

num\_to\_read：读取的数据长度；

### 返回值：

err：0表示正常；

### 示例：

uint8\_t tmp[10];

i2c1->read\_byte(SlaveAddress,SMPLRT\_DIV,tmp,10);

## wait\_dev\_busy(uint8\_t slave\_address);

### 描述：

I2C等待器件响应；

### 参数：

slave\_address：I2C器件地址；

### 返回值：

0：正常；

-1：超时，器件无响应。

### 示例：

for(uint16\_t i = 0; i < num\_to\_write; i++)

{

write\_byte(byte\_addr++,buf[i]);

ret = i2c->wait\_ dev \_busy(SLAVE\_ADDR);

}

## take\_i2c\_right(uint32\_t speed);

### 描述：

获取I2C使用权，并设置速度；**获取不到一直等待**。

### 参数：

**speed：**I2C速度；

### 返回值：

0：正常；

### 示例：看下文

## release\_i2c\_right(void);

### 描述：

释放I2C使用权。

### 参数：

空

### 返回值：

0：正常；

### 示例：

i2c->take\_i2c\_right(speed);

i2c->write\_byte(SLAVE\_ADDR,byte\_addr,byte);

i2c->release\_i2c\_right();

# SOFTI2C类

class SOFTI2C

{

public:

SOFTI2C(GPIO \*scl, GPIO \*sda);

void begin(uint32\_t speed);

int8\_t config(uint32\_t speed);

uint32\_t read\_config();

int8\_t write\_byte(uint8\_t slave\_address,

uint8\_t reg\_address,uint8\_t data);

int8\_t write\_byte(uint8\_t slave\_address,

uint8\_t reg\_address,uint8\_t \*data,uint16\_t num\_to\_write);

int8\_t read\_byte (uint8\_t slave\_address,

uint8\_t reg\_address,uint8\_t \*data);

int8\_t read\_byte (uint8\_t slave\_address,

uint8\_t reg\_address,uint8\_t \*data,uint16\_t num\_to\_read);

int8\_t wait\_dev\_busy (uint8\_t slave\_address);

public:

int8\_t take\_i2c\_right(uint32\_t speed);

int8\_t release\_i2c\_right(void);

private:

void start();

void stop();

int8\_t send\_ack();

int8\_t send\_no\_ack();

int8\_t send\_byte(uint8\_t Byte);

int8\_t send\_7bits\_address(uint8\_t slave\_address);

uint8\_t receive\_byte();

int8\_t wait\_ack();

private:

GPIO \*sda\_pin;

GPIO \*scl\_pin;

uint32\_t speed;

uint16\_t delay\_times;

uint8\_t busy;

};

## SOFTI2C(GPIO\* scl, GPIO\* sda);

### 描述：

SOFTI2C的构造函数，实例化一个SOFTI2C（创建SOFTI2C）对象。此对象一般在object.cpp中定义。

### 参数：

**scl\_pin：**GPIO对象指针或地址，如&PA4；

**sda\_pin：**GPIO对象指针或地址，如&PA5；

### 返回值：空

### 示例：

SOFTI2C si2c(&PA4,&PA5);

SOFTI2C si2c1(&PB6,&PB7);

## begin(uint32\_t speed);

### 描述：

SOFTI2C初始化函数；初始化I2C，并设置速度为\_speed。

### 参数：

**speed：**I2C速度设置，如；100000,200000,300000,400000；注意：如果不是以上四个参数，\_speed的值将会直接用于delay\_us()函数。delay\_us(\_speed)。

### 返回值：

空

### 示例：

si2c1->begin(100000);

## config(uint32\_t speed);

### 描述：

SOFTI2C速度设置函数；

### 参数：

**speed：**I2C速度设置，如；100000,200000,300000,400000；注意：如果不是以上四个参数，\_speed的值将会直接用于delay\_us()函数。delay\_us(\_speed)。

### 返回值：

空

### 示例：

si2c1->config(100000);

## 其他公有成员函数

和硬件I2C的完全兼容，请参考上一章节。

# RTC类

class RTCCLASS

{

public:

uint8\_t sec;

uint8\_t min;

uint8\_t hour;

public:

void begin();

void attach\_interrupt(uint16\_t event, void (\*callbackFun)(void));

void interrupt(uint32\_t bits,FunctionalState x);

void set\_counter(uint32\_t count);

void set\_alarm(uint32\_t count);

uint32\_t get\_counter();

void set\_time\_HMS(uint8\_t h,uint8\_t m,uint8\_t s);

void set\_alarm (uint8\_t h,uint8\_t m,uint8\_t s);

void get\_time\_HMS(uint8\_t \*h,uint8\_t \*m,uint8\_t \*s);

private:

void config();

uint8\_t is\_config(uint16\_t configFlag);

void set\_config\_flag(uint16\_t configFlag);

};

注意：该类有库本身默认实例化对象rtc，不需要用户自己创建对象，

#define RTC\_CLOCK\_LSI 0

#define RTC\_CLOCK\_LSE 1

#define RTC\_CLOCK\_HSI 2//²»Ö§³Ö£¬ÔÝÊ±²»¿ÉÓÃ

//中断源

#define RTC\_EVENT\_OW ((uint16\_t)0x0004) /\*!< Overflow interrupt \*/

#define RTC\_EVENT\_ALR ((uint16\_t)0x0002) /\*!< Alarm interrupt \*/

#define RTC\_EVENT\_SEC ((uint16\_t)0x0001) /\*!< Second interrupt \*/

//用户配置

#define RTC\_CFG\_FLAG 0XA5A5

#define RTC\_CLOCK\_SOURCE RTC\_CLOCK\_LSI

//#define RTC\_CLOCK\_SOURCE RTC\_CLOCK\_LSE

## begin();

### 描述：

初始化rtc；

### 参数：

空；

### 返回值：

空

### 示例：

rtc.begin();

## attach\_interrupt(uint16\_t event,

**void (\*callbackFun)(void));**

### 描述：

串口中断绑定回调函。

### 参数：

**event:**中断事件类型，如RTC\_EVENT\_OW，RTC\_EVENT\_ALR，RTC\_EVENT\_SEC

**(\*callbackFun)(void)：**中断回调函数指针（名），如void rtcSECIt(void)的函数名rtcSECIt。

### 返回值：

空

### 示例：

rtc.attach\_interrupt(RTC\_EVENT\_SEC,rtcSECIt);

## interrupt(uint32\_t bits,FunctionalState x);

### 描述：

rtc中断使能。

### 参数：

**bits:**中断事件类型，如RTC\_EVENT\_OW，RTC\_EVENT\_ALR，RTC\_EVENT\_SEC

**x：**FunctionalState类型值，如ENABLE,DISABLE。

### 返回值：

空

### 示例：

rtc.interrupt(RTC\_EVENT\_SEC,ENABLE);

## set\_counter(uint32\_t count);

### 描述：

设定rtc计数器的值；

### 参数：

**count：**计数器设定值

注解：最大2^32；开始运行136年后溢出；

### 返回值：

空

### 示例：

rtc.set\_counter(100);

## set\_alarm(uint32\_t count);

### 描述：

设定rtc闹钟值，如果rtc的counter中的值和alarm中的值一样，将置位alarm中断标志位；

### 参数：

count：闹钟设定值

注解：最大2^32；

### 返回值：

空

### 示例：

rtc.set\_alarm(110);

## get\_counter();

### 描述：

获取rtc计数器的值；

### 参数：

空；

### 返回值：

**counter：**counter当前值；

### 示例：

val = rtc.get\_counter();

## set\_time\_HMS(uint8\_t h,uint8\_t m,uint8\_t s);

### 描述：

以HHMMSS格式设定rtc闹钟值；

### 参数：

**h：**闹钟时；

**m：**闹钟分；

**s：**闹钟秒；

### 返回值：

空

### 示例：

rtc.set\_time\_HMS(11,59,50);

## set\_alarm(uint8\_t h,uint8\_t m,uint8\_t s);

### 描述：

设定rtc闹钟值，如果rtc的counter中的值和alarm中的值一样，将置位alarm中断标志位；

### 参数：

**h：**闹钟时；

**m：**闹钟分；

**s：**闹钟秒；

注意：如果想使用内部简易的的hhmmss闹钟功能，因为counter默认不清零。请在秒中断中判断counter是不是0x15179,如果是请将counter设置为0。或者在闹钟中断中重新设定新的闹钟值。否则闹钟将失去作用。

### 返回值：

空

### 示例：

rtc.set\_alarm(11,59,59);

## get\_time\_HMS(uint8\_t\* h,uint8\_t\* m,uint8\_t\* s);

### 描述：

获取HHMMSS格式的时间。如果不想使用rtc中断，可通过此函数获得当前时间。

### 参数：

**h：**闹钟时的指针；

**m**：闹钟分的指针；

**s：**闹钟秒的指针；

### 返回值：

空

### 示例：

uint8\_t h,m,s;

rtc.get\_time\_HMS(&h,&m,&s);

## RTC下的HHMMSS时钟变量

uint8\_t sec;

uint8\_t min;

uint8\_t hour;

这三个值在允许秒中断后系统会自动的更新。如果想获取当前的系统的时间，可直接读取这三个值即可。

# FLASHCLASS类

class FLASHCLASS

{

public:

int read(uint32\_t iAddress, uint8\_t \*buf, int32\_t iNbrToRead) ;

int write(uint32\_t iAddress, uint8\_t \*buf, uint32\_t iNbrToWrite);

private:

uint16\_t write\_without\_check(uint32\_t iAddress, uint8\_t \*buf, uint16\_t iNumByteToWrite);

};

#if defined (STM32F10X\_HD) || defined (STM32F10X\_HD\_VL) || defined (STM32F10X\_CL) || defined (STM32F10X\_XL)

#define FLASH\_PAGE\_SIZE ((uint16\_t)0x800)

#else

#define FLASH\_PAGE\_SIZE ((uint16\_t)0x400)

#endif

## read(uint32\_t iAddress, uint8\_t \*buf, int32\_t iNbrToRead) ;

### 描述：

读取内部flash。

### 参数：

**iAddress：**flash绝对地址；

**buf：**缓冲区；

**iNbrToRead：**读取字节数量；

### 返回值：

**i:**返回读取数据长度

### 示例：

flash.read(0x08000000,buf,100);

## write(uint32\_t iAddress, uint8\_t \*buf, uint32\_t iNbrToWrite);

### 描述：

写入内部flash。

### 参数：

**iAddress：**flash绝对地址；

**buf：**缓冲区；

**iNbrToWrite：**读取字节数量；

### 返回值：

iNbrToWrite：返回写入数据长度

### 示例：

Flash.write(0x08000000,buf,100);

# EBOX\_IWDG类

class EBOX\_IWDG

{

public:

EBOX\_IWDG(){};

void begin(uint16\_t ms);

void feed();

};#endif

## EBOX\_IWDG();

### 描述：

创建一个独立看门狗对象。

### 参数：

**空**；

### 返回值：

**空**

### 示例：

EBOX\_IWDG wdg;

## begin(uint16\_t ms);

### 描述：

初始化看门狗。

### 参数：

**ms：**看门狗的最长喂狗时间；

### 返回值：

**空**

### 示例：

wdg. begin (100);

## feed();

### 描述：

喂狗函数，在定时器中调用，或者在主函数，或者在操作系统中的某个任务中。必须在规定的时间内调用一次此函数，否则MCU将复位

### 参数：

**空**；

### 返回值：

**空**

### 示例：

wdg.feed();

# Stream类

### 描述：

Stream类继承了Print，添加了数据的读取和转换功能, 由于不知道底层实现，跟Print类似，将底层的实现申明为虚函数。

Stream申明了3个虚函数：

virtual int available() = 0; //判断当前是否有可读取的数据

virtual int read() = 0; //读取一个字节并丢弃

virtual int peek() = 0; //读取这个字节但不丢弃，此时如果在读取的话还是同样的数据

读取数据时，可能当前没有数据可读取但数据很快就传送过来，也可能根本就没有数据，为了适应这种不确定性，设定了超时读取时间，超过这个时间还没有读取到数据则读取失败，单位ms，默认1000ms

void setTimeout(unsigned long timeout);

unsigned long getTimeout(void)

## available(void)

描述： 该函数 available() 获取数据流中接收到的字节数

返回值： int类型

## read(void)

描述： 获取数据流中第一个字节数据，获取数据后会清除当前字节数据，与peek()函数有区别

返回值： 取数据字符的第一个字节（8bit）,无数据时返回 -1

## peak(void)

描述： 从数据流中读取当前的一个字节，不会清除数据流中当前字节数据，与 read() 函数有区别

返回值： 取数据字符的第一个字节（8bit）,无数据时返回 -1

## findUntil(char \*target, size\_t targetLen, char \*terminate, size\_t termLen);

描述： 在接受到的数据中查找字符串target，遇到terminator结束：

参数：

char \*target 指向查找的目标

size\_t targetLen 目标字符串长度

char \*terminate 指向结束字符串

size\_t termLen 结束字符串长度

返回值：找到目标字符串返回True，超时返回 false

## find(char \*target);

## find(char \*target, size\_t length);

## find(char target)；

## findUntil(char \*target, char \*terminator);

17.2->17.5 都是通过调用17.1实现的，相关说明参考17.1

## parseInt(LookaheadMode lookahead = SKIP\_ALL, char ignore = NO\_IGNORE\_CHAR);

描述： 读取数据并转换成整型，转换从第一个遇到的数字或者负号开始，到第一个非数字结束

参数：

LookaheadMode lookahead 忽略

SKIP\_ALL, 忽略所有无效字符

SKIP\_NONE 不跳过任何字符

SKIP\_WHITESPACE 跳过空格，tab，换行，回车

char ignore = NO\_IGNORE\_CHAR

返回值：Long

## parseFloat(LookaheadMode lookahead = SKIP\_ALL, char ignore = NO\_IGNORE\_CHAR);

描述： 读取数据并转换成浮点型，转换从第一个遇到的数字或者负号开始，到第一个非数字结束

参数： 参考parseInt（）

返回值： Float (**注意，超时会返回0**，建议在调用此函数前先调用available()判断已经收到数据)

## readBytes( char \*buffer, size\_t length);

描述：读取指定长度的数据到buffer

参数：

char \*buffer 缓存

size\_t length 要读取的长度

返回值：返回放置在缓存中的字符数

## readBytesUntil( char terminator, char \*buffer, size\_t length);

描述： 读取指定长度的字符到数组buffer，遇到终止字符terminator后停止。

参数：

char terminator, 终止字符

char \*buffer 缓存

size\_t length 要读取的长度

返回值： 返回存入缓存的字符数，0表示没有有效数据