

BabyOS设计和使用手册

V0.3.1

BabyOS V8.1.6

修订记录:

日期	记录	修订人
2022.03.18	编写初稿	notrynohigh
2022.06.05	增加功能模块详细介绍	notrynohigh
2022.06.06	修改按键模块的描述，文档对应代码的版本号	notrynohigh
2022.10.22	链接文件增加b_mod_state段 增加状态机介绍	notrynohigh
2022.12.07	更新驱动部分描述以及部分功能模块接口描述	notrynohigh
2023.01.16	更新算法模块描述和IAP接口描述	notrynohigh

目录

BabyOS设计和使用手册

目录

1. 项目介绍

2. 设计思路

3. 快速体验

- 3.1 虚拟机中体验BabyOS
- 3.2 基于STM32体验BabyOS
 - 3.2.1 准备基础工程
 - 3.2.2 添加BabyOS代码
 - 3.2.3 修改配置
 - 3.2.4 调用必要的函数
 - 3.2.5 快速体验结果

4. 进阶体验

- 4.1 补充MCU资源初始化
- 4.2 添加驱动文件
- 4.3 添加硬件接口
- 4.4 记录开机次数

5. 概要介绍

- 5.1 添加MCU
- 5.2 HAL层介绍
 - 5.2.1 心跳时钟
 - 5.2.2 延时函数
 - 5.2.3 通讯接口
- 5.3 驱动层介绍
 - 5.3.1 硬件接口
 - 5.3.2 注册设备
 - 5.3.3 操作设备
- 5.4 SECTION介绍
- 5.5 功能组件

6. 功能模块

- 6.1 b_mod_adchub
 - 6.1.1 数据结构
 - 6.1.2 接口介绍
 - 6.1.3 使用例子
- 6.2 b_mod_button
 - 6.2.1 数据结构
 - 6.2.2 接口介绍
 - 6.2.3 使用例子
- 6.3 b_mod_error
 - 6.3.1 数据结构
 - 6.3.2 接口介绍
 - 6.3.3 使用例子
- 6.4 b_mod_fs
 - 6.4.1 数据结构
 - 6.4.2 接口介绍
 - 6.4.3 使用例子
- 6.5 b_mod_gui
 - 6.5.1 数据结构
 - 6.5.2 接口介绍
 - 6.5.3 使用例子
- 6.6 b_mod_kv
 - 6.6.1 数据结构
 - 6.6.2 接口介绍
 - 6.6.3 使用例子

- 6.7 b_mod_menu
 - 6.7.1 数据结构
 - 6.7.2 接口介绍
 - 6.7.3 使用例子
- 6.8 b_mod_modbus
 - 6.8.1 数据结构
 - 6.8.2 接口介绍
 - 6.8.3 使用例子
- 6.9 b_mod_param
 - 6.9.1 数据结构
 - 6.9.2 接口介绍
 - 6.9.3 使用例子
- 6.10 b_mod_protocol
 - 6.10.1 数据结构
 - 6.10.2 接口介绍
 - 6.10.3 使用例子
- 6.11 b_mod_pwm
 - 6.11.1 数据结构
 - 6.11.2 接口介绍
 - 6.11.3 使用例子
- 6.12 b_mod_shell
 - 6.12.1 数据结构
 - 6.12.2 接口介绍
 - 6.12.3 使用例子
- 6.13 b_mod_timer
 - 6.13.1 数据结构
 - 6.13.2 接口介绍
 - 6.13.3 使用例子
- 6.14 b_mod_trace
 - 6.14.1 数据结构
 - 6.14.2 接口介绍
 - 6.14.3 使用例子
- 6.15 b_mod_xm128
 - 6.15.1 数据结构
 - 6.15.2 接口介绍
 - 6.15.3 使用例子
- 6.16 b_mod_ymodem
 - 6.16.1 数据结构
 - 6.16.2 接口介绍
 - 6.16.3 使用例子
- 6.17 b_mod_iap
 - 6.17.1 数据结构
 - 6.17.2 接口介绍
 - 6.17.3 使用例子
- 6.18 b_mod_state
 - 6.18.1 数据结构
 - 6.18.2 接口介绍
 - 6.18.3 使用例子

7.工具模块

- 7.1 b_util_at
 - 7.1.1 数据结构
 - 7.1.2 接口介绍
- 7.2 b_util_fifo
 - 7.2.1 数据结构
 - 7.2.2 接口介绍
- 7.3 b_util_i2c
 - 7.3.1 数据结构
 - 7.3.2 接口介绍

- 7.4 b_util_spi
 - 7.4.1 数据结构
 - 7.4.2 接口介绍
- 7.5 b_util_log
 - 7.5.1 接口介绍
- 7.6 b_util_lunar
 - 7.6.1 数据结构
 - 7.6.2 接口介绍
- 7.7 b_util_memp
 - 7.7.1 数据结构
 - 7.7.2 接口介绍
- 7.8 b_util_uart
 - 7.8.1 数据结构
 - 7.8.2 接口介绍
- 7.9 b_util_utc
 - 7.9.1 数据结构
 - 7.9.2 接口介绍

8. 参与开发

1. 项目介绍

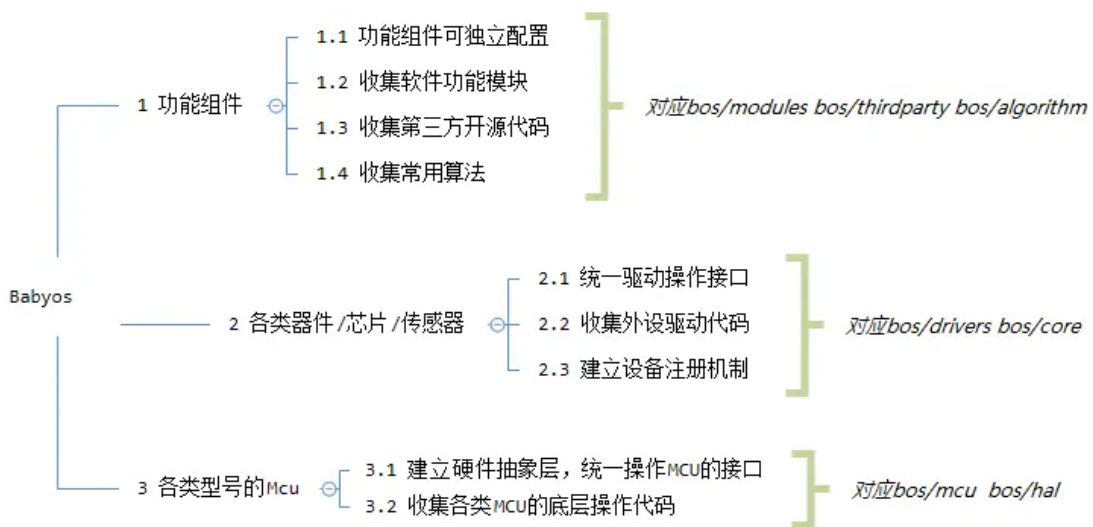
BabyOS构想是搭建一个货架存放软件CBB（CommonBuildingBlock）。

不同产品、系统之间有许多共用的模块，这些模块调试稳定后放入货架作为积累。久而久之，货架上便有许多成熟的CBB供开发人员使用。减少大量重复劳动或者研发已经存在的成果。另一方面，如果产品是基于这些成熟的CBB搭建而成，产品的质量、进度都会得到更好的控制和保证。

货架的搭建和CBB积累并不是一个人可以完成的，需要集合众人的力量。于是2019年底发起了BabyOS开源项目。开发人员以兴趣为动力，无任何薪资报酬。坚持开源互助，共同进步。

2. 设计思路

BabyOS是想搭建一个货架，那么货架上是怎么存放东西的呢？这便决定了代码的结构。




```

        RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA | RCC_APB2Periph_USART1 |
RCC_APB2Periph_AFIO, ENABLE);
    }

    static void _GpioInit()
    {
        GPIO_InitTypeDef GPIO_InitStructure;
        GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;
        GPIO_InitStructure.GPIO_Pin = GPIO_Pin_9;
        GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
        GPIO_Init(GPIOA, &GPIO_InitStructure);
        GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN_FLOATING;
        GPIO_InitStructure.GPIO_Pin = GPIO_Pin_10;
        GPIO_Init(GPIOA, &GPIO_InitStructure);
    }

    static void _UartInit()
    {
        NVIC_InitTypeDef NVIC_InitStructure;
        USART_InitTypeDef USART_InitStructure;
        USART_InitStructure.USART_BaudRate = 115200;
        USART_InitStructure.USART_WordLength = USART_WordLength_8b;
        USART_InitStructure.USART_StopBits = USART_StopBits_1;
        USART_InitStructure.USART_Parity = USART_Parity_No;
        USART_InitStructure.USART_HardwareFlowControl =
USART_HardwareFlowControl_None;
        USART_InitStructure.USART_Mode = USART_Mode_Rx |
USART_Mode_Tx;
        USART_Init(USART1, &USART_InitStructure);
        USART_ITConfig(USART1, USART_IT_RXNE, ENABLE);
        NVIC_InitStructure.NVIC_IRQChannel = USART1_IRQn;
        NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
        NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 1;
        NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;
        NVIC_Init(&NVIC_InitStructure);
        USART_Cmd(USART1, ENABLE);
    }

    void BoardInit()
    {
        _ClockInit();
        _GpioInit();
        _UartInit();
    }

```

```

//滴答定时器
SysTick_Config(SystemCoreClock / TICK_FRQ_HZ);
NVIC_SetPriority(SysTick_IRQn, 0x0);

```

3.2.2 添加BabyOS代码

路径	部分/全部	用于快速体验
bos/algorithm	全部添加	全部添加
bos/core	全部添加	全部添加
bos/drivers	根据需要添加	暂时不添加其中文件
bos/hal	全部添加	全部添加
bos/mcu	根据需要添加	添加bos/mcu/st/stm32f10x/路径代码
bos/modules	全部添加	全部添加
bos/thirdparty	根据需要添加	添加bos/thirdparty/nr_micro_shell/路径代码
bos/utils	全部添加	全部添加
bos/_config		b_config.h 全局 配置文件 b_device_list.h 注册设备的文件 b_hal_if.h 驱动接口文件

编译器添加两个路径即可：

bos/

_config/ 如果配置文件拷贝到其他路径了，则添加相应路径即可。

3.2.3 修改配置

配置项	说明	用于快速体验
Version Configuration	版本配置项，硬件和固件版本	无改动
Platform Configuration	平台配置项，指定心跳频率和MCU平台	MCU平台选择 STM32F10X_CL
Hal Configuration	硬件接口配置，可配置硬件接口参数是固定还是可变的	无改动
Utils Configuration	实用软件配置，部分软件代码的配置	无改动
Algorithm Configuration	算法模块配置，各个算法模块配置	无改动
Modules Configuration	模块配置项，各个功能模块的配置	无改动
Thirdparty Configuration	第三方开源代码配置项	勾选 NR Micro Shell Enable/Disable

b_hal_if.h 中指定DEBUG接口


```
#ifndef __B_HAL_IF_H__
#define __B_HAL_IF_H__
#include "b_config.h"
// debug
#define HAL_LOG_UART B_HAL_UART_1
#endif
```

3.2.4 调用必要的函数

包含头文件 `b_os.h`

①滴答定时器中断服务函数调用 `bHalIncSysTick()`;

```
void SysTick_Handler()
{
    bHalIncSysTick();
}
```

②调用 `bInit()`; `bExec()`;

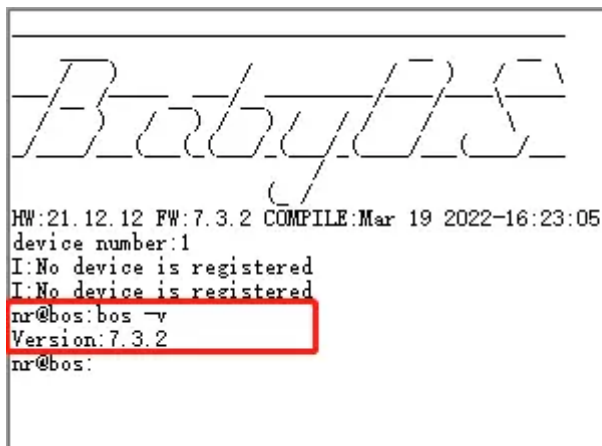
```
int main()
{
    BoardInit();
    SysTick_Config(SystemCoreClock / TICK_FRQ_HZ);
    NVIC_SetPriority(SysTick_IRQn, 0x0);

    bInit();           //bos初始化
    bShellInit();      //shell初始化
    while (1)
    {
        bExec();       //bos执行函数
    }
}
```

③由于勾选了shell功能模块，所以需要在串口接收中断服务函数里调用 `bShellParse`，将数据喂给模块。

3.2.5 快速体验结果

BabyOS的shell模块默认支持查询版本的指令，输入 `bos -v` 便可以查询到版本。

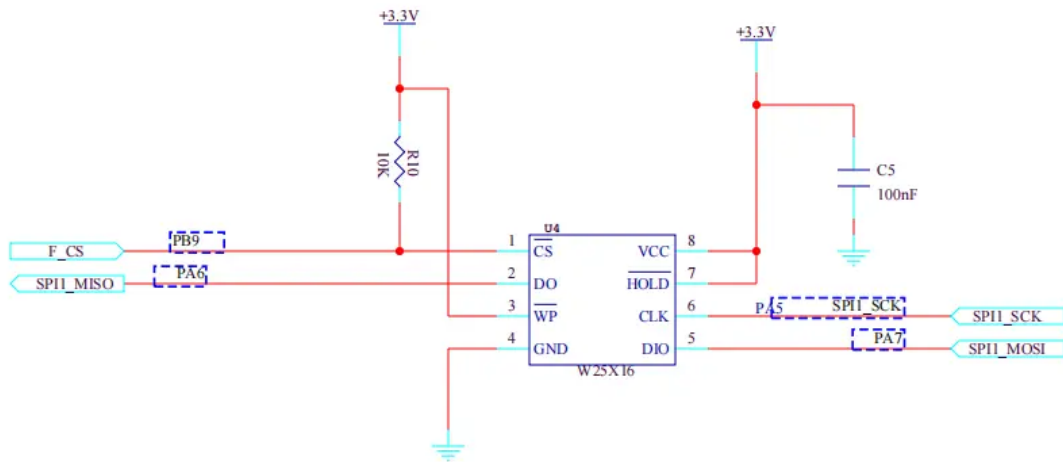


```

BabyOS
HW:21.12.12 FW:7.3.2 COMPILE:Mar 19 2022-16:23:05
device number:1
I:No device is registered
I:No device is registered
nr@bos:~$ bos -v
Version: 7.3.2
nr@bos:~$
```

4. 进阶体验

完成快速体验后，再体验设备的注册和相关操作。以SPIFlash为例进行说明。



SPI FLASH

4.1 补充MCU资源初始化

在快速体验工程的基础上，增加了硬件SPI，和F_CS引脚。增加SPI的初始化以及GPIO的初始化。

代码省略....

4.2 添加驱动文件

添加 `bos/drivers/b_drv_spiflash.c`

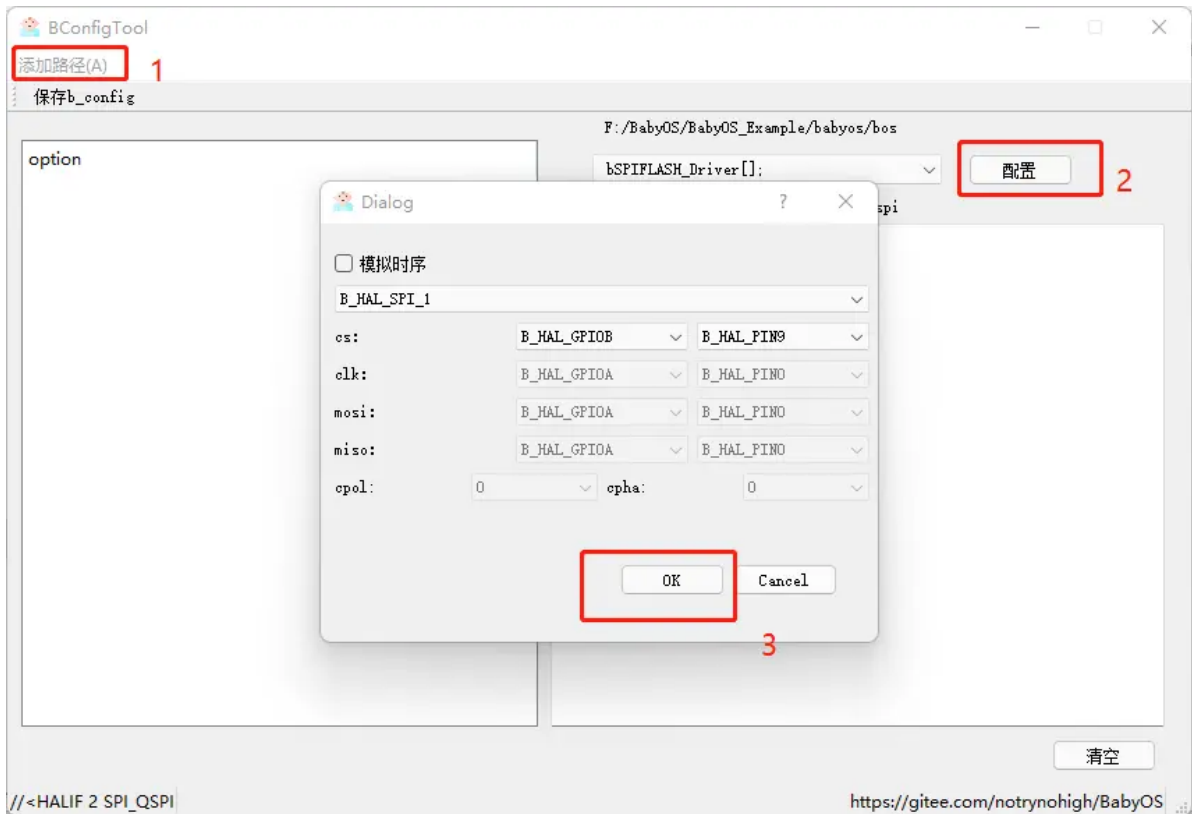
BabyOS里面SPIFLASH的驱动是基于sfud代码编写。因此也要添加sfud部分的代码。

添加 `bos/drivers/sfud/` 路径的代码。

4.3 添加硬件接口

在**hal_if.h**里面添加硬件接口。

可利用BabyOS配置工具生成代码。 <https://gitee.com/notrynohigh/bconfig-tool/releases/V0.0.2>



4.4 记录开机次数

在SPIFLASH的地址0x00000000记录开机次数，增加如下代码

```
int fd = -1;
uint32_t boot_count = 0;
fd = bopen(bSPIFLASH, BCORE_FLAG_RW);
bLseek(fd, 0);
bRead(fd, (uint8_t *)&boot_count, sizeof(boot_count));
b_log("boot:%d\r\n", boot_count);
boot_count += 1;
bFlashErase_t bFlashErase;
bFlashErase.addr = 0;
```

```
bFlashErase.num = 1;
bctl(fd, bCMD_ERASE_SECTOR, &bFlashErase);
blseek(fd, 0);
bwrite(fd, (uint8_t *)&boot_count, sizeof(boot_count));
bclose(fd);
```

BoobyDS

HW:21.12.12 FW:7.3.2 COMPILER:Mar 19 2022-19:01:14

device number:2

W:sfud_init 130 Start initialize Serial Flash Universal Driver(SFUD) V1.1.0.

W:sfud_init 131 You can get the latest version on <https://github.com/armink/SFUD>.

W:read_jedec_id 1026 The flash device manufacturer ID is 0xEF, memory type ID is 0x30, capacity ID is 0x15.

W:read_sfdp_header 134 Error: Check SFDP signature error. It's must be 50444653h('S' 'F' 'D' 'P').

Warning: Read SFDP parameter header information failed. The 000 is not support JEDEC SFDP.

Find a Winbond W25X16BV flash chip. Size is 2097152 bytes.

W:reset 1001 Flash device reset success.

000 flash device is initialize success.

nr@bos:boot:0

BoobyDS

HW:21.12.12 FW:7.3.2 COMPILER:Mar 19 2022-19:01:14

device number:2

W:sfud_init 130 Start initialize Serial Flash Universal Driver(SFUD) V1.1.0.

W:sfud_init 131 You can get the latest version on <https://github.com/armink/SFUD>.

W:read_jedec_id 1026 The flash device manufacturer ID is 0xEF, memory type ID is 0x30, capacity ID is 0x15.

W:read_sfdp_header 134 Error: Check SFDP signature error. It's must be 50444653h('S' 'F' 'D' 'P').

Warning: Read SFDP parameter header information failed. The 000 is not support JEDEC SFDP.

Find a Winbond W25X16BV flash chip. Size is 2097152 bytes.

W:reset 1001 Flash device reset success.

000 flash device is initialize success.

nr@bos:boot:1

5.概要介绍

5.1 添加MCU

`bos/mcu/` 路径是存放已调试过的MCU型号，命名规则是：*bos/mcu/厂商/型号/*。

`bos/hal/` 目录的文件及文件内定义的接口目前并不是很全，这部分的策略是：一点点添加，上层代码有需要时再添加。

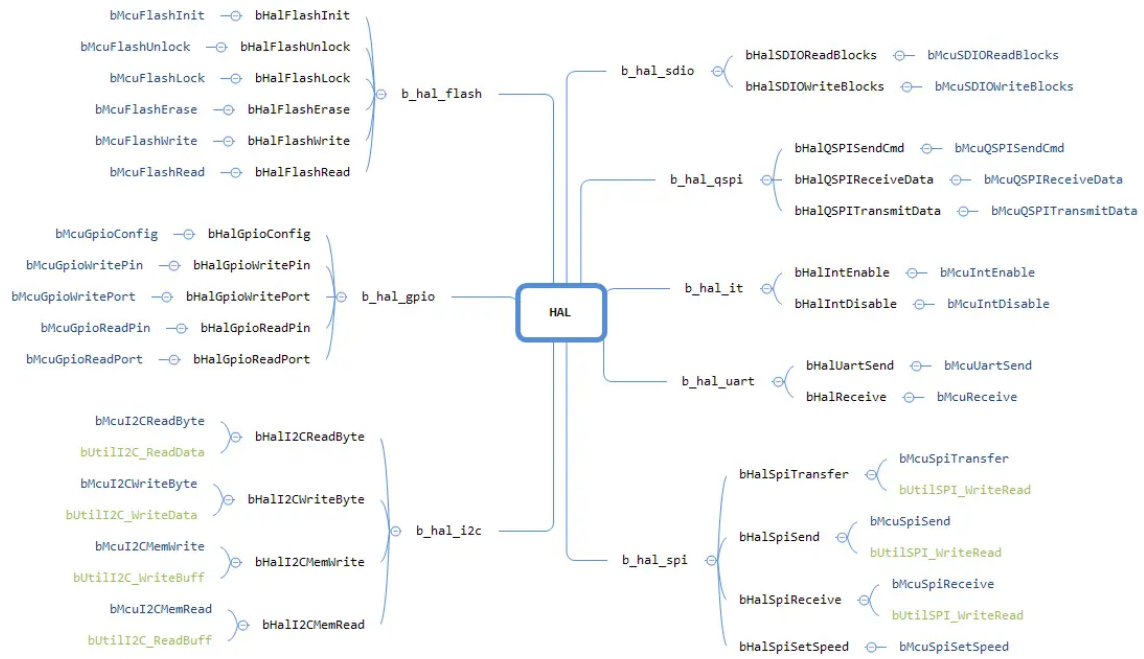
下图中黑色部分是HAL部分的内容，蓝色部分是MCU部分需要实现的，绿色部分是UTILS提供的模拟时序。

SPI和I2C接口支持模拟时序，HAL层判断是否使用模拟时序，然后调用对应接口。

因此新增MCU型号：

- ①新建目录，添加文件
- ②实现蓝色部分的接口
- ③修改 `_config/b_config.h`，为MCU Platform增加一个选项

```
//<o> MCU Platform
//<1001=> STM32F10X_LD
//<1002=> STM32F10X_MD
//<1003=> STM32F10X_HD
//<1004=> STM32F10X_CL
//<1101=> STM32G0X0
//<2001=> NATION_F40X
//<3001=> MM32SPIN2X
//<3002=> MM32SPIN0X
//<4001=> HC32L13X
//<7001=> CH32F103
#define MCU_PLATFORM 1004
```



5.2 HAL层介绍

Hal层一方面是给MCU层提供统一的接口。还有如下几点作用：

- ①提供心跳时间的查询
- ②提供微秒级和毫秒级延时函数
- ③提供通讯接口的数据结构

5.2.1 心跳时钟

使用BabyOS,需要给予一个心跳时钟。心跳时钟的频率在 `_config/b_config.h` 里定义 `TICK_FRQ_HZ` ,使用者自行实现一个定时器,并定时调用 `bHalIncSysTick` 。应用代码中可根据 `bHalGetSysTick` 获取心跳时钟计数值。

5.2.2 延时函数

提供 `bHalDelayMs` 和 `bHalDelayUs` 两个阻塞型延时函数。毫秒级延时是通过心跳计算的。微妙级函数是通过for循环阻塞。`bHalInit` 中会计算微妙级延时所用到的参数,以此尽量保证微妙级函数的精准性。

5.2.3 通讯接口

HAL层提供通讯接口的数据结构：

```
//GPIO
typedef struct
{
    bHalGPIOPort_t port;
    bHalGPIOPin_t pin;
} bHalGPIOInstance_t;

//I2C
typedef struct
{
    uint8_t dev_addr;
    uint8_t is_simulation;
    union
    {
        bHalI2CNumber_t i2c;
        struct
        {
            bHalGPIOInstance_t clk;
            bHalGPIOInstance_t sda;
        } simulating_i2c;
    } _if;
} bHalI2CIf_t;

//SPI
typedef struct
{
    uint8_t is_simulation;
    union
    {
        bHalSPINumber_t spi;
```

```

        struct
        {
            bHalGPIOInstance_t miso;
            bHalGPIOInstance_t mosi;
            bHalGPIOInstance_t clk;
            uint8_t          CPOL;
            uint8_t          CPHA;
        } simulating_spi;
    } _if;
    bHalGPIOInstance_t cs;
} bHalSPIIf_t;

//UART
typedef enum
{
    B_HAL_UART_1,
    B_HAL_UART_2,
    ....
    B_HAL_UART_NUMBER
} bHalUartNumber_t;

//LCD
typedef struct
{
    union
    {
        uint32_t rw_addr;
        struct
        {
            bHalGPIOInstance_t data;
            bHalGPIOInstance_t rs;
            bHalGPIOInstance_t rd;
            bHalGPIOInstance_t wr;
            bHalGPIOInstance_t cs;
        } _io;
        struct
        {
            bHalGPIOInstance_t rs;
            bHalSPIIf_t        _spi;
        } _spi;
    } _if;
    uint8_t if_type; // 0: _io 1: rw_addr 2: _spi
} bLCD_HalIf_t;

```


5.3 驱动层介绍

BabyOS里面有几个重要的概念：设备、驱动和硬件接口。

设备：是一个实体，与MCU通过硬件接口相连接。

驱动，是针对设备的一份软件逻辑代码

硬件接口：是设备与MCU相连的数字接口

操作硬件接口，使用HAL层的接口，操作设备，使用bOpen bClose等接口

例如：SPIFLASH是一个设备，b_drv_spiflash是驱动，SPI是硬件接口

bos/driver/inc/b_driver.h 里定义了驱动的统一接口：

```
typedef struct bDriverIf
{
    int status;
    int (*init)(struct bDriverIf *pdrv);
    int (*open)(struct bDriverIf *pdrv);
    int (*close)(struct bDriverIf *pdrv);
    int (*ctl)(struct bDriverIf *pdrv, uint8_t cmd, void *param);
    int (*write)(struct bDriverIf *pdrv, uint32_t offset, uint8_t *pbuf,
uint32_t len);
    int (*read)(struct bDriverIf *pdrv, uint32_t offset, uint8_t *pbuf, uint32_t
len);
    void *hal_if;
    const char *pdes;
    uint32_t drv_no;
    union
    {
        uint32_t v;
        void *_p;
    } _private;
} bDriverInterface_t;
```

每个驱动文件的目标便是实现 bDriverInterface_t 里面的各个元素。

status 驱动初始化异常则将 status 设为-1 反之设为 0。操作设备时检测此项，如果是-1 则不执行。

init 负责执行初始化，用于运行过程中再次初始化。

open 负责唤醒操作，此处可执行设备唤醒。如果设备没有休眠状态，可以赋值为 NULL。

close 负责休眠的操作，此处可执行设备休眠。如果设备没有休眠状态，可以赋值为 NULL。

ctl 负责对设备进行配置或者执行特定的操作，例如擦除，切换状态等。ctl 的调用需要传入指令 cmd 和对应的参数。执行成功返回 0，失败或者不支持指令则返回-1。

write 负责传输数据至设备，执行成功返回实际发送的数据长度，执行失败则返回-1。

read 负责从设备获取数据，获取数据的最小单元依据设备功能而定，例如，存储设备，最小可以获取 1 个字节；3 轴加速度设备，最小单元为 3 个加速度值；温湿度传感器最小单元是一组温度湿度值。读取的最小单元需要在驱动的 h 文件进行说明让使用者能明白。

_hal_if 指向当前驱动对应的硬件接口。

pdes 指向设备的描述信息

drv_no 指同类设备中的序号，例如有3个SPIFLASH，那么它们的编号0，1，2便是存在drv_no。

private 当驱动需要携带私有参数时，则利用这个字段。例如 flash 的 id，可以放在 *private.v*。如果需要存放更多的信息，那么就利用private.p 指向一片数据区域。

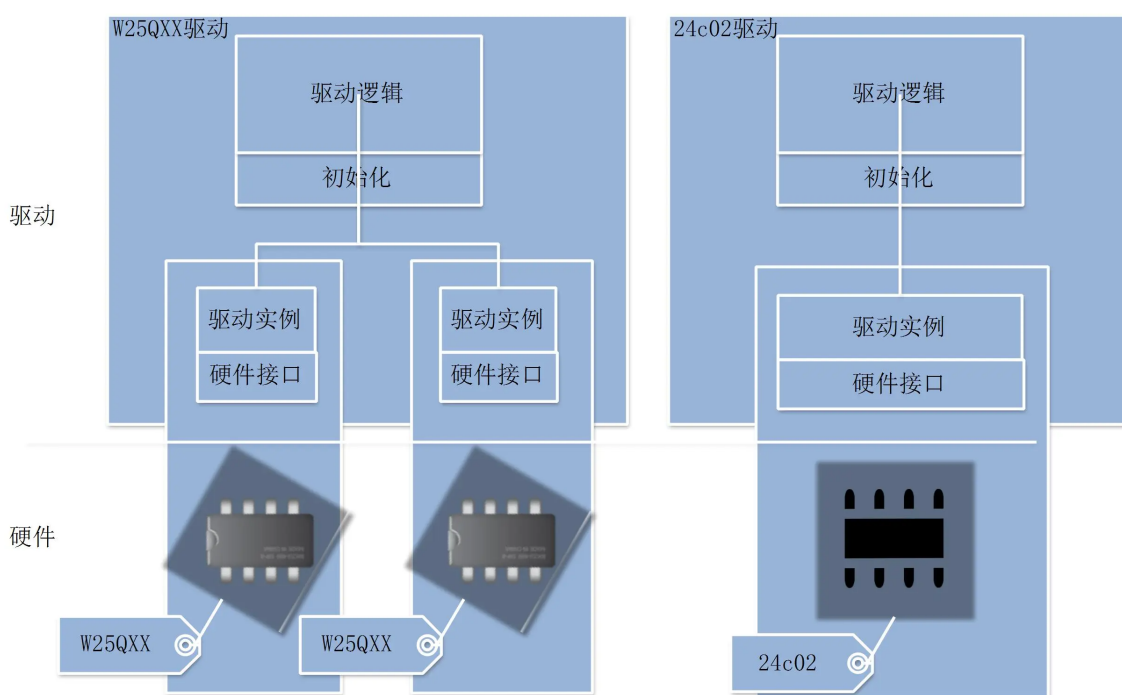
5.3.1 硬件接口

硬件接口通过 HAL_XXXX_IF 指定，具体可查看b_hal_if.h文件：

```
//-----  
//      驱动文件的命名规则 b_drv_<驱动名小写>.c .h  
//      每个驱动文件里有宏定义 #define DRIVER_NAME 驱动名大写  
//      例如： spiflash驱动  
//              驱动文件为 b_drv_spiflash.c .h  
//              c文件里面定义宏 #define DRIVER_NAME SPIFLASH  
//      驱动需要在此文件（b_hal_if.h）定义HAL层接口  
//      #define HAL_<DRIVER_NAME>_IF  
//      例如spiflash #define HAL_SPIFLASH_IF {具体的数据接口查看h文件}  
//-----  
//      如果有多个spiflash：  
//      #define HAL_SPIFLASH_IF {第一个SPIFLASH},{第二个SPIFLASH}  
//-----
```

注册设备

设备号+驱动实例+描述



例如2个SPIFLASH,1个24C02

```
//b_hal_if.h  
#define HAL_24CXX_IF  
  
{  
  
    .dev_addr = 0xa0,  

```

```

        .is_simulation = 1, \
        ._if.simulating_i2c.clk = {B_HAL_GPIOB, B_HAL_PIN6}, \
        ._if.simulating_i2c.sda = {B_HAL_GPIOB, B_HAL_PIN7}, \
    }

#define HAL_SPIFLASH_IF \
{ \
    .is_spi = 1, \
    ._if._spi = { \
        .is_simulation = 0, \
        .cs = {B_HAL_GPIOB, B_HAL_PIN9}, \
        ._if.spi = B_HAL_SPI_1, \
    }, \
}, \
{ \
    .is_spi = 1, \
    ._if._spi = { \
        .is_simulation = 0, \
        .cs = {B_HAL_GPIOB, B_HAL_PIN8}, \
        ._if.spi = B_HAL_SPI_1, \
    }, \
}

```

5.3.2 注册设备

注册设备，便是设备与驱动建立联系的过程：

`b_device_list.h` 中通过宏进行注册：

```

/**
typedef enum
{
    B_DRIVER_NULL = 0,
    B_DRIVER_24CXX,
    B_DRIVER_DS18B20,
    B_DRIVER_ESP12F,
    B_DRIVER_FM25CL,
    B_DRIVER_ILI9320,
    B_DRIVER_ILI9341,
    B_DRIVER_KEY,
    B_DRIVER_LIS3DH,
    B_DRIVER_MATRIXKEYS,
    B_DRIVER_MCUFLASH,
    B_DRIVER_OLED,
    B_DRIVER_PCF8574,
    B_DRIVER_SD,
    B_DRIVER_SPIFLASH,
    B_DRIVER_SSD1289,
    B_DRIVER_ST7789,
    B_DRIVER_TESTFLASH,
    B_DRIVER_XPT2046,
    B_DRIVER_NUMBER
} bDriverNumber_t;
*/
/**
    B_DEVICE_REG(dev_1, bDriverNumber_t, "description")
    .....

```

```

        B_DEVICE_REG(dev_n, bDriverNumber_t, "description")
    */

    B_DEVICE_REG(bTESTFLASH, B_DRIVER_TESTFLASH, "testflash")

```

设备注册后，便会自动生成如下数据结构以及定义的数组：

```

//b_device.h
//生成设备号
typedef enum
{
#define B_DEVICE_REG(dev, driver, desc) dev,
#include "b_device_list.h"
    B_REG_DEV_NULL,
    B_REG_DEV_NUMBER
} bDeviceName_t;
//b_device.c
//驱动号数组
static bDriverNumber_t bDriverNumberTable[B_REG_DEV_NUMBER] = {
#define B_DEVICE_REG(dev, driver, desc) driver,
#include "b_device_list.h"
    B_DRIVER_NULL,
};
//设备描述信息数组
static const char *bDeviceDescTable[B_REG_DEV_NUMBER] = {
#define B_DEVICE_REG(dev, driver, desc) desc,
#include "b_device_list.h"
    "null",
};
//驱动实例数组
static bDriverInterface_t bDriverInterfaceTable[B_REG_DEV_NUMBER];

```

根据注册的设备，定义相应数量的驱动实例。以宏的形式来实现，避免对动态内存的依赖。

BabyOS的代码中，在尽可能避免使用动态内存。

5.3.3 操作设备

```

int bOpen(uint32_t dev_no, uint8_t flag);
int bRead(int fd, uint8_t *pdata, uint32_t len);
int bwrite(int fd, uint8_t *pdata, uint32_t len);
int bCtl(int fd, uint8_t cmd, void *param);
int bLseek(int fd, uint32_t off);
int bclose(int fd);

int bInit(void);
int bExec(void);
int bReinit(uint32_t dev_no);

int bModifyHalIf(uint32_t dev_no, uint32_t type_size, uint32_t off, const
uint8_t *pval,
                uint8_t len);

```

dev_no 注册设备时指定的设备号

fd 打开设备后返回的句柄。

配置项 `_HALIF_VARIABLE_ENABLE` 用于配置是否允许硬件接口可以改动。

```
#if _HALIF_VARIABLE_ENABLE
#define HALIF_KEYWORD static
#else
#define HALIF_KEYWORD const static
#endif
```

`bModifyHalIf` 使用例子:

```
//oled硬件接口数据结构是 boLED_HalIf_t
typedef struct
{
    union
    {
        bHalI2CIf_t _i2c;
        bHalSPIIf_t _spi;
    } _if;
    uint8_t is_spi;
} boLED_HalIf_t;
// 修改IIC的设备地址 OLED是注册的设备号, dev_addr变量存放着新的指。
bModifyHalIf(OLED, sizeof(boLED_HalIf_t), (uint8_t)(&(((boLED_HalIf_t *)0)-
>_if._i2c.dev_addr)), &dev_addr, 1)
```

5.4 SECTION介绍

b_section.h 定义段的操作。现有的段有如下几个：

```
bSECTION_DEF_FLASH(bos_polling, pbPoling_t);
#define BOS_REG_POLLING_FUNC(func) //将func放入bos_polling段

#define bDRIVER_REG_INIT_0(drv_num, init_f) //将init_f放入driver_init_0段
#define bDRIVER_REG_INIT(drv_num, init_f) //将init_f放入driver_init段

bSECTION_DEF_FLASH(b_mod_shell, static_cmd_st);
#define bSHELL_REG_INSTANCE(cmd_name, cmd_handler) //将cmd信息放入b_mod_shell段

bSECTION_DEF_FLASH(b_mod_param, bParamInstance_t);
#define bPARAM_REG_INSTANCE(param, param_size) //将参数信息放入b_mod_param段

bSECTION_DEF_FLASH(b_mod_state, bStateInfo_t *);
#define bSTATE_REG_INSTANCE(state_info) //将状态信息放入b_mod_state段
```

驱动文件最后会有一行这样的代码：`bDRIVER_REG_INIT(B_DRIVER_XXX, bXXX_Init);`将初始化函数放入 `driver_init` 段。

```
//设备初始化时，将遍历driver_init_0和driver_init内的函数，并执行。
int bDeviceInit()
{
    uint32_t i = 0, j = 0;
    memset(bDriverInterfaceTable, 0, sizeof(bDriverInterfaceTable));
    bSECTION_FOR_EACH(driver_init_0, bDriverRegInit_t, pdriver_init_0)
    {
        j = 0;
        for (i = 0; i < B_REG_DEV_NUMBER; i++)
        {
            if (bDriverNumberTable[i] == pdriver_init_0->drv_number)
            {
                bDriverInterfaceTable[i].drv_no = j++;
                bDriverInterfaceTable[i].pdes = bDeviceDescTable[i];
                bDriverInterfaceTable[i].status = pdriver_init_0->
>init(&bDriverInterfaceTable[i]);
            }
        }
    }
    bSECTION_FOR_EACH(driver_init, bDriverRegInit_t, pdriver_init)
    {
        j = 0;
        for (i = 0; i < B_REG_DEV_NUMBER; i++)
        {
            if (bDriverNumberTable[i] == pdriver_init->drv_number)
            {
                bDriverInterfaceTable[i].drv_no = j++;
                bDriverInterfaceTable[i].pdes = bDeviceDescTable[i];
                bDriverInterfaceTable[i].status = pdriver_init->
>init(&bDriverInterfaceTable[i]);
            }
        }
    }
}
```

```

    return 0;
}

```

```

int bExec()
{
    //BabyOS的执行函数遍历需要轮询的函数即在bos_polling段的函数。
    bSECTION_FOR_EACH(bos_polling, pbPoling_t, polling)
    {
        (*polling)();
    }
    return 0;
}

```

当使用gcc编译时，需要编辑链接文件，在链接文件中补充这几个段，例如：

```

/* Define output sections */
SECTIONS
{
    .....
    /* BabyOS Section -----*/
    .driver_init :
    {
        . = ALIGN(4);
        PROVIDE(__start_driver_init = .);
        KEEP(*(SORT(.driver_init*)))
        PROVIDE(__stop_driver_init = .);
        . = ALIGN(4);
    } > FLASH

    .driver_init_0 :
    {
        . = ALIGN(4);
        PROVIDE(__start_driver_init_0 = .);
        KEEP(*(SORT(.driver_init_0*)))
        PROVIDE(__stop_driver_init_0 = .);
        . = ALIGN(4);
    } > FLASH

    .bos_polling :
    {
        . = ALIGN(4);
        PROVIDE(__start_bos_polling = .);
        KEEP(*(SORT(.bos_polling*)))
        PROVIDE(__stop_bos_polling = .);
        . = ALIGN(4);
    } > FLASH

    .b_mod_shell :
    {
        . = ALIGN(4);
        PROVIDE(__start_b_mod_shell = .);
        KEEP(*(SORT(.b_mod_shell*)))
        PROVIDE(__stop_b_mod_shell = .);
        . = ALIGN(4);
    } > FLASH
}

```

```
.b_mod_state :
{
    . = ALIGN(4);
    PROVIDE(__start_b_mod_state = .);
    KEEP(*(SORT(.b_mod_state*)))
    PROVIDE(__stop_b_mod_state = .);
    . = ALIGN(4);
} > FLASH

/* BabyOS Section -----end-----*/
.....
}
```


5.5 功能组件

功能组件包括: 功能模块、第三方开源代码，算法模块和工具模块。

组件	描述	代码
功能模块	收集BabyOS开发者编写的通用软件模块	b_mod_adchub b_mod_button b_mod_error b_mod_fs b_mod_gui b_mod_kv b_mod_menu b_mod_modbus b_mod_param b_mod_protocol b_mod_pwm b_mod_shell b_mod_timer b_mod_trace b_mod_xm128 b_mod_ymodem
第三方开源	收集第三方实用的开源代码	cjson cm_backtrace fatfs flexiblebutton littlefs nr_micro_shell ugui sfud
算法模块	收集常用的算法。	algo_base64 algo_crc algo_md5 algo_sort algo_utf8_unicode algo_hmac_sha1
工具模块	支持其他各模块的通用代码	b_util_at b_util_fifo b_util_i2c b_util_log b_util_lunar b_util_memp b_util_spi b_util_uart b_util_utc

组件的每个部分都可以通过全局配置文件使能以及配置参数。组件中的代码，操作MCU资源只能调用HAL层接口，操作设备只能基于设备号进行操作。

组件中每个c文件功能单一,提供的功能接口放在对应的h文件。尽量做到, 根据h文件的函数名便知道如何使用。

6. 功能模块

6.1 b_mod_adchub

6.1.1 数据结构

```
//回调 ad_val: ADC值      arg:用户指定传入的参数
typedef void (*pAdchubCb_t)(uint32_t ad_val, uint32_t arg);

typedef struct _AdcInfo
{
    uint8_t      seq;          //序号, 每个实例中序号不能一样
    uint8_t      filter;       //是否进行默认滤波处理
    uint8_t      flag;         //buf是否填满
    uint8_t      index;        //当前喂入的数据放入buf的索引
    pAdchubCb_t  callback;     //回调函数
    uint32_t      arg;          //指定回调传入的参数
    uint32_t      buf[FILTER_BUF_SIZE];
    struct _AdcInfo *next;
    struct _AdcInfo *prev;
} bAdcInfo_t;

typedef bAdcInfo_t bAdcInstance_t;

//快速创建实例的宏, name:实例名 ad_seq:序号 filter_en: 是否需要滤波 cb:回调 cb_arg:回调参数
#define bADC_INSTANCE(name, ad_seq, filter_en, cb, cb_arg) \
    bAdcInstance_t name = {                                \
        .seq      = ad_seq,                                \
        .filter   = filter_en,                              \
        .callback = cb,                                     \
        .arg      = cb_arg,                                  \
    }


```

6.1.2 接口介绍

```
//注册ADCHUB实例, 所有注册的实例将组成列表
int bAdchubRegist(bAdcInstance_t *pinstance);
//喂ADC数据, ad_seq:ADC的序号 ad_val:ADC的值
int bAdchubFeedValue(uint8_t ad_seq, uint32_t ad_val);


```

6.1.3 使用例子

```
//回调函数
void _AdcCallback(uint32_t ad_val, uint32_t arg)
{
    b_log("%d:%d\r\n", arg, ad_val);
    if (arg == 2) //可以根据arg来判断是哪个实例的回调
    {
        //.....
    }
}


```

```

//此处定义实例，序号分别填的是10和16，在喂数据时候要对应
//由于使用同一个回调函数，那么回调带入的参数要区分，分别是1 和 2
bADC_INSTANCE(ADTest, 10, 1, _AdcCallback, 1);
bADC_INSTANCE(ADTemp, 16, 1, _AdcCallback, 2);
int main()
{
    ...
    bInit();
    //注册实例
    bAdchubRegist(&ADTest);
    bAdchubRegist(&ADTemp);
    ...
}

//喂数据，中断里获取ADC值，然后喂数据
void ADC1_2_IRQHandler()
{
    uint32_t tmp = 0;
    if (ADC_GetITStatus(ADC1, ADC_IT_JEOC) == SET)
    {
        ADC_ClearITPendingBit(ADC1, ADC_IT_JEOC);
        tmp = ADC_GetInjectedConversionValue(ADC1, ADC_InjectedChannel_1);
        bAdchubFeedValue(10, tmp);
        tmp = ADC_GetInjectedConversionValue(ADC1, ADC_InjectedChannel_2);
        bAdchubFeedValue(16, tmp);
    }
}

```

6.2 b_mod_button

此功能模块是对第三方代码FlexibleButton的封装。支持独立按键和矩阵按键。

6.2.1 数据结构

```

typedef void (*pBtnEventHandler_t)(uint32_t dev_no, uint8_t sub_id, uint16_t
event, uint8_t param);

typedef struct bButtonInstance
{
    uint32_t          dev_no;
    uint16_t          event;
    pBtnEventHandler_t handler;
    struct bButtonInstance *next;
} bButtonInstance_t;

//按键事件，1个按键可以同时注册多个事件
#define BTN_EVENT_DOWN (0x001)
#define BTN_EVENT_CLICK (0x002)
#define BTN_EVENT_DOUBLE_CLICK (0x004)
#define BTN_EVENT_REPEAT_CLICK (0x008)
#define BTN_EVENT_SHORT (0x010)
#define BTN_EVENT_SHORT_UP (0x020)
#define BTN_EVENT_LONG (0x040)
#define BTN_EVENT_LONG_UP (0x080)
#define BTN_EVENT_LONGLONG (0x100)

```

```
#define BTN_EVENT_LONGLONG_UP (0x200)
```

6.2.2 接口介绍

```
#define bBUTTON_ADD_KEY(dev, e, e_handler)
#define bBUTTON_ADD_MATRIXKEYS(dev, e, e_handler)
// 请不要直接调用下面这两个函数。使用bBUTTON_ADD_KEY和bBUTTON_ADD_MATRIXKEYS代替
int bButtonAddKey(bButtonInstance_t *pbutton, flex_button_t *pflex);
int bButtonAddMatrixKeys(bButtonInstance_t *pbutton, flex_button_t *pflex);
```

6.2.3 使用例子

b_hal_if定义按键的硬件接口，b_dev_list.h注册按键设备。

```
#define HAL_KEY_IF

#define HAL_MATRIXKEYS_IF

void BtnEventHandler(uint32_t dev_no, uint8_t sub_id, uint16_t event, uint8_t
param)
{
    b_log("dev:%d id:%d event:%x param:%d\n", dev_no, sub_id, event, param);
}

int main()
{
    ...
    bInit();
    bBUTTON_ADD_KEY(bKEY1, BTN_EVENT_CLICK | BTN_EVENT_LONG, BtnEventHandler);
    bBUTTON_ADD_KEY(bKEY2, BTN_EVENT_CLICK | BTN_EVENT_LONG, BtnEventHandler);
    bBUTTON_ADD_KEY(bKEY3, BTN_EVENT_CLICK | BTN_EVENT_LONG, BtnEventHandler);
    bBUTTON_ADD_KEY(bKEY4, BTN_EVENT_CLICK | BTN_EVENT_LONG, BtnEventHandler);
    bBUTTON_ADD_MATRIXKEYS(bMATRIXKEYS, BTN_EVENT_CLICK | BTN_EVENT_LONG,
BtnEventHandler);
    ...
}
```

6.3 b_mod_error

6.3.1 数据结构

```
typedef void (*pecb)(uint8_t err);    //错误发生后的回调

#define INVALID_ERR ((uint8_t)0xFF)

#define BERROR_LEVEL_0 0x00    //错误等级0，调用回调后自动从队列中移除
#define BERROR_LEVEL_1 0x01    //错误等级1，需要手动从队列移除
```

6.3.2 接口介绍

```
//初始化并传入回调函数
int bErrorInit(pecb cb);
//注册错误, err:错误号 level:错误等级
//interval_ms:间隔时间, level为BERROR_LEVEL_1时有效。
//当错误发生后执行一次回调, 如果错误没有被清除, 则interval_ms时间后再次执行回调
int bErrorRegist(uint8_t err, uint32_t interval_ms, uint32_t level);
//清除指定的错误
int bErrorClear(uint8_t e_no);
//查询错误是否存在
int bErrorIsExist(uint8_t e_no);
//查询错误队列是否为空, 即没有错误发生或者发生的错误都已经被处理
int bErrorIsEmpty(void);
```

6.3.3 使用例子

```
#define BAT_LOW (0)
#define MEM_ERR (1)

void SystemErrCallback(uint8_t err)
{
    b_log_e("err:%d\r\n", err);
}

int main()
{
    ...
    bInit();
    bErrorInit(SystemErrCallback);
    bErrorRegist(BAT_LOW, 3000, BERROR_LEVEL_1); //当错误发生时调用
    bErrorRegist(MEM_ERR, 0, BERROR_LEVEL_0);    //当错误发生时调用
    ...
}
```

6.4 b_mod_fs

6.4.1 数据结构

```
//定义了两个物理盘, SPIFLASH和SDCARD
typedef enum
{
    #if _SPIFLASH_ENABLE
        E_DEV_SPIFLASH, /* Map SPIFLASH to physical drive*/
    #endif
    #if _SD_ENABLE
        E_DEV_SDCARD, /* Map MMC/SD card to physical drive*/
    #endif
    E_DEV_NUMBER,
} FS_DEV_Enum_t;
```

6.4.2 接口介绍

```
//b_mod_fs是对接fatfs和littlefs
//b_mod_fs主要是提供初始化函数，其他文件级操作使用fatfs或者littlefs提供的接口。
//初始化函数
int bFS_Init(void);
//提供的测试函数，主要是通过文件的方式记录开机次数
int bFS_Test(void);
```

6.4.3 使用例子

```
int main()
{
    ...
    bInit();
    bFS_Init();
    bFS_Test();
    ...
}
```

6.5 b_mod_gui

6.5.1 数据结构

```
typedef struct bGUIStruct
{
    const uint32_t    lcd_dev_no;
    const uint32_t    touch_dev_no;
    const int         touch_type;
    const uint16_t     lcd_x_size;
    const uint16_t     lcd_y_size;
    uint8_t           lcd_disp_dir;
    uint16_t           touch_ad_x[2];
    uint16_t           touch_ad_y[2];
    UG_GUI             gui_handle;
    struct bGUIStruct *pnext;
} bGUIStruct_t;

typedef bGUIStruct_t bGUIInstance_t;

#define TOUCH_TYPE_RES (0)
#define TOUCH_TYPE_CAP (1)

#define LCD_DISP_H (0)
#define LCD_DISP_V (1)

#define bGUI_ADD_DEVICE(_lcd_dev_no, _touch_dev_no, _x_size, _y_size, \
    _touch_type) \
    static bGUIInstance_t gui_##_lcd_dev_no = { \
        \
        .lcd_dev_no    = _lcd_dev_no, \
        \
        .touch_dev_no  = _touch_dev_no, \
        \
        .touch_type     = _touch_type, \
        \
        .lcd_x_size     = _x_size, \
        \
        .lcd_y_size     = _y_size, \
        \
        .lcd_disp_dir   = LCD_DISP_H, \
        \
        .touch_ad_x     = {0}, \
        \
        .touch_ad_y     = {0}, \
        \
        .gui_handle     = 0, \
        \
        .pnext          = 0 \
    }
```

```

        .touch_dev_no = _touch_dev_no,
    \
        .touch_type    = _touch_type,
    \
        .lcd_x_size    = _x_size,
    \
        .lcd_y_size    = _y_size,
    \
        .pnext         = NULL,
    \
    };
    \
    bGUIRegist(&gui_##_lcd_dev_no);

```

6.5.2 接口介绍

此模块支持多个屏使用uGUI。UI设计使用ugui.h文件提供的接口。

```

// 建议不要直接调用此接口，使用 bGUI_ADD_DEVICE 代替
int bGUIRegist(bGUIInstance_t *pInstance);

// 通过屏的dev_no获取GUI句柄
int bGUIGetHandle(uint32_t lcd_dev_no, UG_GUI **p_gui_handle);

// 选择当前操作的目标
int bGUISelect(uint32_t lcd_dev_no);

// 设置电阻屏触摸的AD值范围
int bGUITouchRange(uint32_t lcd_dev_no, uint16_t x_ad_min, uint16_t x_ad_max,
uint16_t y_ad_min,
uint16_t y_ad_max);

// 设置屏幕显示方向，默认是LCD_DISP_V
int bGUIDispDir(uint32_t lcd_dev_no, uint8_t dir);

// 如果将汉字字库存放在FLASH，通过此接口传入dev_no
int bGUISetFontDevice(uint32_t dev_no);

```

6.5.3 使用例子

```

void TouchTest()
{
    UG_GUI *p_gui = NULL;
    bGUIGetHandle(bTFT, &p_gui);
    if (p_gui)
    {
        if (p_gui->touch.state == TOUCH_STATE_PRESSED)
        {
            b_log("x:%d y:%d \r\n", p_gui->touch.xp, p_gui->touch.yp);
        }
    }
}

int main()

```



```

{
    .....
    bInit();
    bGUI_ADD_DEVICE(bTFT, bTOUCH, 240, 320, TOUCH_TYPE_RES);
    bGUI_ADD_DEVICE(bOLED, NULL, 128, 64, 0);

    bGUITouchRange(bTFT, 476, 3952, 338, 3592);

    bGUISelect(bTFT);
    UG_FillScreen(C_BLACK);
    UG_PutString(0, 0, "hello world");
    UG_PutString(0, 100, "babyos ssd1289");
    bGUISelect(bOLED);
    UG_FillScreen(0);
    UG_PutString(0, 0, "hello world");
    UG_PutString(0, 20, "babyos oled");
    bGUISelect(bTFT);
    bGUIDispDir(bTFT, LCD_DISP_H);
    UG_PutString(0, 20, "babyos oled tft");
    while (1)
    {
        bExec();
        //测试触摸
        BOS_PERIODIC_TASK(TouchTest, 500);
    }
}

```

6.6 b_mod_kv

6.6.1 数据结构

```

//bkv的状态
#define bKV_IDLE 0
#define bKV_BUSY 1
#define bKV_ERROR 2
//bkv区域至少是有4个最小可擦除单位。
//【数据索引1】【数据1】 【数据索引2】【数据2】
#define bKV_SECTOR_T1 0X01
#define bKV_SECTOR_T2 0X02
#define bKV_SECTOR_D1 0X04
#define bKV_SECTOR_D2 0X08
#define bKV_SECTOR_ALL 0X0F
//kv区域的标志字符串
#define bKV_HEAD_STR "B_KV"

#define bKV_ALIGN_4BYTES(n) (((n) + 3) / 4 * 4)

```

6.6.2 接口介绍

```

//初始化, dev_no: 存储数据的设备号 s_addr:起始地址 size:存储区域尺寸 e_size:最小擦除单位大小
int bKV_Init(int dev_no, uint32_t s_addr, uint32_t size, uint32_t e_size);
//设置KV的数据
int bKV_Set(const char *key, uint8_t *pvalue, uint16_t len);
//读取KV数据
int bKV_Get(const char *key, uint8_t *pvalue);
//删除KV的KEY
int bKV_Delete(const char *key);

```

6.6.3 使用例子

```

int main()
{
    ...
    bInit();
    bKV_Init(BSPIFLASH, 0x0, 40960, 4096);
    if(0 > bKV_Get("boot", (uint8_t *)&boot_count))
    {
        boot_count = 0;
    }
    b_log("boot : %d\r\n", boot_count);
    boot_count += 1;
    bKV_Set("boot", (uint8_t *)&boot_count, sizeof(boot_count));
    ...
}

```

6.7 b_mod_menu

6.7.1 数据结构

```

//更新UI的函数, pre_id: 当前界面是从pre_id的界面切换过来
typedef void (*pCreateUI)(uint32_t pre_id);

//切换菜单的操作
#define MENU_UP 1
#define MENU_DOWN 2
#define MENU_BACK 3
#define MENU_ENTER 4

```

6.7.2 接口介绍

```

//增加同等级的菜单。创建第一个节点时，参考ID和界面ID值相同。
int      bMenuAddSibling(uint32_t ref_id, uint32_t id, pCreateUI f);
//增加子级菜单
int      bMenuAddChild(uint32_t ref_id, uint32_t id, pCreateUI f);
//菜单切换操作
void     bMenuAction(uint8_t cmd);
//直接跳转到ID界面
void     bMenuJump(uint32_t id);
//获取当前显示界面的ID
uint32_t bMenuCurrentID(void);
//设置ID界面的可视化状态，用于隐藏和显示界面
int      bMenuSetVisible(uint32_t id, uint8_t s);

```

6.7.3 使用例子

```

//定义4个按键进行菜单切换操作
void BtnEventHandler0(uint16_t event, uint8_t param)
{
    bMenuAction(MENU_UP);
}
void BtnEventHandler1(uint16_t event, uint8_t param)
{
    bMenuAction(MENU_DOWN);
}
void BtnEventHandler2(uint16_t event, uint8_t param)
{
    bMenuAction(MENU_BACK);
}
void BtnEventHandler3(uint16_t event, uint8_t param)
{
    bMenuAction(MENU_ENTER);
}

//创建菜单。更多的代码，参考example仓库的例程。
int bMenuInit()
{
    bMenuAddSibling(LEVEL0_MENU0_ID, LEVEL0_MENU0_ID, Level0Menu0F);
    bMenuAddSibling(LEVEL0_MENU0_ID, LEVEL0_MENU1_ID, Level0Menu1F);
    bMenuAddSibling(LEVEL0_MENU1_ID, LEVEL0_MENU2_ID, Level0Menu2F);

    bMenuAddChild(LEVEL0_MENU0_ID, LEVEL1_MENU0_ID, Level1Menu0F);
    bMenuAddChild(LEVEL0_MENU1_ID, LEVEL1_MENU1_ID, Level1Menu1F);
    bMenuAddChild(LEVEL0_MENU2_ID, LEVEL1_MENU2_ID, Level1Menu2F);

    return 0;
}

```

6.8 b_mod_modbus

6.8.1 数据结构

```

//这部分代码主要是提供RTU模式的主机读写功能
//从机返回读数据结果的数据结构
typedef struct
{

```

```

    uint8_t    func;
    uint8_t    reg_num;
    uint16_t *reg_value;
} bMB_ReadResult_t;
//从机返回写数据结果的数据结构
typedef struct
{
    uint8_t    func;
    uint16_t reg;
    uint16_t reg_num;
} bMB_WriteResult_t;
//传入回调函数的数据结构
typedef struct
{
    uint8_t type; // 0: read    1:write
    union
    {
        bMB_ReadResult_t r_result;
        bMB_WriteResult_t w_result;
    } result;
} bMB_SlaveDeviceData_t;

typedef void (*pMB_Send_t)(uint8_t *pbuf, uint16_t len);
typedef void (*pMB_Callback_t)(bMB_SlaveDeviceData_t *pdata);
//指定发送函数和回调函数
typedef struct
{
    pMB_Send_t      f;
    pMB_Callback_t cb;
} bMB_Info_t;

typedef bMB_Info_t bModbusInstance_t;

//可以通过这个宏快速创建实例，创建实例的时候指定发送和回调函数
#define bMODBUS_INSTANCE(name, pSendData, pCallback) \
    bModbusInstance_t name = {.f = pSendData, .cb = pCallback};

```

6.8.2 接口介绍

```

//读取寄存器的值
int bMB_ReadRegs(bModbusInstance_t *pModbusInstance, uint8_t addr, uint8_t func,
uint16_t reg, uint16_t num);
//写寄存器的值
int bMB_WriteRegs(bModbusInstance_t *pModbusInstance, uint8_t addr, uint8_t
func, uint16_t reg, uint16_t num, uint16_t *reg_value);
//将接收到的数据喂给模块，让模块进行解析。解析正确后执行回调
int bMB_FeedReceivedData(bModbusInstance_t *pModbusInstance, uint8_t *pbuf,
uint16_t len);

```

6.8.3 使用例子

```
//...待添加
```

6.9 b_mod_param

6.9.1 数据结构

注意：使用此功能模块，需要同时使能shell功能模块

```
//size:变量的大小Byte name: 变量名 addr:变量地址
typedef struct
{
    uint8_t size;
    char*   name;
    void*   addr;
} bParamStruct_t;

typedef bParamStruct_t bParamInstance_t;

#define _PARAM2STR(n) (#n)
//注册实例，指定需要调整的变量名和变量大小
#define bPARAM_REG_INSTANCE(param, param_size)
    \
    bSECTION_ITEM_REGISTER_FLASH(b_mod_param, bParamInstance_t, CONCAT_2(do_,
param)) = { \
    .size = param_size, .name = _PARAM2STR(param), .addr = &(param)};
```

6.9.2 接口介绍

```
//通过 #define bPARAM_REG_INSTANCE(param, param_size) 注册实例后，通过shell操作
//例如变量名 i
//param i  查询变量值
//param i 8 设置变量名值为8
```

6.9.3 使用例子

```
static uint32_t TestTick = 0;
//为了测试，变量值每秒增加1
void TestParamF()
{
    TestTick += 1;
}
//通过指令查询和调整TestTick的值
bPARAM_REG_INSTANCE(TestTick, 4);

int main()
{
    ...
    bInit();
    bShellInit();
}
```

```

...
while (1)
{
    bExec();
    BOS_PERIODIC_TASK(TestParamF, 1000);
}
}
/*
nr@bos:bos -v
Version:7.4.0
nr@bos:param TestTick    //查询变量值
TestTick:19
nr@bosparam TestTick    //查询变量值
TestTick:23
nr@bosparam TestTick 0    //设置变量值为0
nr@bosparam TestTick    //再次查询
TestTick:4
nr@bos:*/

```

6.10 b_mod_protocol

此模块提供通用协议格式，测试软件

(https://gitee.com/notrynohigh/BabyOS_Protocol/tree/master) :

```

/**
|      |      |      |      |      |      |
| :--- | ----- | ----- | ---- | ----- | ---- |
| Head | Device ID | Len (cmd+param) | Cmd | Param | Check |
| 0xFE | sizeof(bProtoID_t) | sizeof(bProtoLen_t) | 1Byte | 0~nBytes | 1Byte |
*/

```

设备ID的长度以及len字段的长度可以在b_config文件进行配置。

设备ID：发送数据时，该字段是**目标设备的ID**，如果设备ID为0xFFFFFFFF表示广播。

接收数据时，**判断ID字段与自身的ID是否匹配**。或者ID是否为0xFFFFFFFF。

6.10.1 数据结构

```

#if PROTO_FID_SIZE == 1
typedef uint8_t bProtoID_t;
#define INVALID_ID 0xFF
#elif PROTO_FID_SIZE == 2
typedef uint16_t bProtoID_t;
#define INVALID_ID 0xFFFF
#else
typedef uint32_t bProtoID_t;
#define INVALID_ID 0xFFFFFFFF
#endif

#if PROTO_FLEN_SIZE == 1
typedef uint8_t bProtoLen_t;
#else
typedef uint16_t bProtoLen_t;

```

```

#endif

#pragma pack(1)
typedef struct
{
    uint8_t    head;
    bProtoID_t device_id;
    bProtoLen_t len;
    uint8_t    cmd;
} bProtocolHead_t;
#pragma pack()
//分发函数，当接收的数据按照协议解析成功，则调用分发函数
typedef int (*pdispatch)(uint8_t cmd, uint8_t *param, bProtoLen_t param_len);

#define PROTOCOL_HEAD 0xFE

```

6.10.2 接口介绍

```

//初始化，指定设备自身的ID和分发函数
int bProtocolInit(bProtoID_t id, pdispatch f);
//修改设备ID
int bProtocolSetID(bProtoID_t id);
//将接收到的数据喂给模块进行解析
int bProtocolParse(uint8_t *pbuf, bProtoLen_t len);
//将数据根据协议打包。打包完成的数据放在pbuf,同时返回数据长度
int bProtocolPack(uint8_t cmd, uint8_t *param, bProtoLen_t param_size, uint8_t *pbuf);

```

6.10.3 使用例子

```

//协议分发函数 cmd:指令 param:参数 param_len:参数长度
int ProtocolDispatch(uint8_t cmd, uint8_t *param, bProtoLen_t param_len)
{
    b_log("cmd:%d param_len: %d\r\n", cmd, param_len);
    // 添加指令对应的执行代码
    return 0;
}

//接收空闲
int ProtocolRecCallback(uint8_t *pbuf, uint16_t len)
{
    //接收完一段数据后，将数据给模块进行解析
    bProtocolParse(pbuf, len);
    return 0;
}

BUTIL_UART_INSTANCE(protocol, 128, 100, ProtocolRecCallback);

int main()
{
    ...
    bInit();
    bProtocolInit(0x520, ProtocolDispatch);
    ...
}

```

6.11 b_mod_pwm

6.11.1 数据结构

```
#define PWM_HANDLER_CCR (0)
#define PWM_HANDLER_PERIOD (1)
//PWM回调函数, type: PWM_HANDLER_CCR or PWM_HANDLER_PERIOD
typedef void (*PwmHandler)(uint8_t type);

typedef struct bSoftPwmStruct
{
    uint32_t          repeat;    //指定重复次数, 为0则一直重复
    uint32_t          tick;      //用于计时
    uint32_t          period;    //周期, 单位ms
    uint32_t          ccr;       //CCR, 单位ms
    PwmHandler        handler;   //回调执行函数
    uint32_t          flag;      //执行回调标志
    struct bSoftPwmStruct *next;
} bSoftPwmStruct_t;

typedef bSoftPwmStruct_t bSoftPwmInstance_t;

// 创建PWM实例, 指定PWM的参数
#define bPWM_INSTANCE(name, _period, _ccr, _repeat) \
    bSoftPwmInstance_t name = {.period = _period, .ccr = _ccr, .repeat = \
    _repeat};
```

6.11.2 接口介绍

```
//启动PWM, 并指定回调
int bSoftPwmStart(bSoftPwmInstance_t *pPwmInstance, PwmHandler handler);
int bSoftPwmStop(bSoftPwmInstance_t *pPwmInstance);
int bSoftPwmReset(bSoftPwmInstance_t *pPwmInstance);
int bSoftPwmSetPeriod(bSoftPwmInstance_t *pPwmInstance, uint32_t ms);
int bSoftPwmSetCcr(bSoftPwmInstance_t *pPwmInstance, uint32_t ms);
```

6.11.3 使用例子

```
bPWM_INSTANCE(led1_pwm, 20, 5, 0);
bPWM_INSTANCE(led2_pwm, 20, 18, 0);

void PwmHandler1(uint8_t type)
{
    if(type == PWM_HANDLER_CCR)
    {
        bHalGpioWritePin(B_HAL_GPIOD, B_HAL_PIN7, 0);
    }
    else
    {
        bHalGpioWritePin(B_HAL_GPIOD, B_HAL_PIN7, 1);
    }
}
```



```

void PwmHandler2(uint8_t type)
{
    if(type == PWM_HANDLER_CCR)
    {
        bHalGpioWritePin(B_HAL_GPIOD, B_HAL_PIN3, 0);
    }
    else
    {
        bHalGpioWritePin(B_HAL_GPIOD, B_HAL_PIN3, 1);
    }
}

int main()
{
    ...
    bInit();
    bSoftPwmStart(&led1_pwm, PwmHandler1);
    bSoftPwmStart(&led2_pwm, PwmHandler2);
    ...
}

```

6.12 b_mod_shell

此软件模块对接nr_micro_shell

6.12.1 数据结构

```

typedef void (*pCmdHandler)(char argc, char *argv);

//注册指令和指令的执行函数
#define bSHELL_REG_INSTANCE(cmd_name, cmd_handler)

```

6.12.2 接口介绍

```

//shell模块初始化
//初始化后，添加了默认指令，bos -v 查询版本
void bShellInit(void);
//解析函数，接收的数据放入此处解析
int bShellParse(uint8_t *pbuf, uint16_t len);

```

6.12.3 使用例子

```

int main()
{
    ...
    bInit();
    bShellInit();
    ...
}

void USART1_IRQHandler()
{
    uint8_t uart_dat = 0;
}

```

```

    if (USART_GetITStatus(USART1, USART_IT_RXNE) == SET)
    {
        USART_ClearITPendingBit(USART1, USART_IT_RXNE);
        uart_dat = USART_ReceiveData(USART1);
        bshellParse(&uart_dat, 1);    //shell 解析
    }
}

```

6.13 b_mod_timer

6.13.1 数据结构

```

//定时器回调
typedef void (*pTimerHandler)(void);

typedef struct bSoftTimerStruct
{
    uint8_t          repeat;    //单次定时还是重复, 0: 单次    1: 重复
    uint32_t         tick;
    uint32_t         period;
    pTimerHandler    handler;
    struct bSoftTimerStruct *next;
} bSoftTimerStruct_t;

typedef bSoftTimerStruct_t bSoftTimerInstance_t;
//创建实例的宏
#define bTIMER_INSTANCE(name, _period, _repeat) \
    bSoftTimerInstance_t name = {.period = _period, .repeat = _repeat};

```

6.13.2 接口介绍

```

int bSoftTimerStart(bSoftTimerInstance_t *pTimerInstance, pTimerHandler
handler);
int bSoftTimerStop(bSoftTimerInstance_t *pTimerInstance);
int bSoftTimerReset(bSoftTimerInstance_t *pTimerInstance);
int bSoftTimerSetPeriod(bSoftTimerInstance_t *pTimerInstance, uint32_t ms);

```

6.13.3 使用例子

```

bTIMER_INSTANCE(timer1, 1000, 1);
bTIMER_INSTANCE(timer2, 2000, 1);
void Timer1Handler()
{
    b_log("babyos\r\n");
}
void Timer2Handler()
{
    b_log("hello \r\n");
}
int main()
{
    ...
}

```

```

    bInit();
    bSoftTimerStart(&timer1, Timer1Handler);
    bSoftTimerStart(&timer2, Timer2Handler);
    ...
}

```

6.14 b_mod_trace

当前软件模块对接的是CmBacktrace

6.14.1 数据结构

```
//...
```

6.14.2 接口介绍

```
int bTraceInit(const char *pfw_name); //初始化并指定固件名
```

6.14.3 使用例子

参考<https://gitee.com/Armink/CmBacktrace/tree/master>

6.15 b_mod_xm128

6.15.1 数据结构

```

//XMODEM回调，number是序号，pbuf是指向数据的指针，当pbuf为NULL时，表示接收完毕
typedef void (*pcb_t)(uint16_t number, uint8_t *pbuf);
//发送函数，用于发送指令
typedef void (*psend)(uint8_t cmd);

```

6.15.2 接口介绍

```

//初始化，指定回调和发送函数
int bxmodem128Init(pcb_t fcb, psend fs);
//将接收的数据喂给模块进行解析
int bxmodem128Parse(uint8_t *pbuf, uint8_t len);
//XModem开始和停止
int bxmodem128Start(void);
int bxmodem128Stop(void);

```

6.15.3 使用例子

```

uint8_t FileBuf[1024];
uint16_t FileLen = 0;
//XModem回调
void XModemCallback(uint16_t number, uint8_t *pbuf)
{

```

```

    if(pbuf != NULL)
    {
        memcpy(&FileBuf[FileLen], pbuf, 128);
        FileLen += 128;
    }
}

//xModem 发送接口
void XmodemSend(uint8_t cmd)
{
    bHalUartSend(HAL_LOG_UART, &cmd, 1);
}

//串口接收空闲，需要接收空闲后喂数据
int UartIdleCallback(uint8_t *pbuf, uint16_t len)
{
    bXmodem128Parse(pbuf, len);
    return 0;
}

//建立串口接收实例
BUTIL_UART_INSTANCE(XmodemRec, 200, 50, UartIdleCallback);

int main()
{
    ...
    bInit();
    bXmodem128Init(XModemCallback, XmodemSend);

    //开始传输
    bXmodem128Start();
    ...
}

void USART1_IRQHandler()
{
    uint8_t uart_dat = 0;
    if (USART_GetITStatus(USART1, USART_IT_RXNE) == SET)
    {
        USART_ClearITPendingBit(USART1, USART_IT_RXNE);
        uart_dat = USART_ReceiveData(USART1);
        butilUartRxHandler(&XmodemRec, uart_dat);
    }
}

```

6.16 b_mod_ymodem

6.16.1 数据结构

```

//ymodem回调。t:标题或者数据 pbuf:数据 len:数据长度
typedef void (*pymcb_t)(uint8_t t, uint8_t *pbuf, uint16_t len);
//发送接口
typedef void (*pymsend)(uint8_t cmd);

```

6.16.2 接口介绍

```
//初始化，提供回调和发送接口
int bYmodemInit(pymcb_t fcb, pymsend fs);
//解析函数，收到的数据喂入进行解析
int bYmodemParse(uint8_t *pbuf, uint16_t len);
//YModem的开始和停止
int bYmodemStart(void);
int bYmodemStop(void);
```

6.16.3 使用例子

```
uint8_t FileBuf[1024];
uint16_t FileLen = 0;
//回调函数，t可以为文件名也可以是文件数据 pbuf是数据，当pbuf为NULL时结束 len是数据的长度
void YModemCallback(uint8_t t, uint8_t *pbuf, uint16_t len)
{
    if(pbuf != NULL && (t == YMODEM_FILEDATA))
    {
        memcpy(&FileBuf[FileLen], pbuf, len);
        FileLen += len;
    }
}
//YModem发送接口
void YmodemSend(uint8_t cmd)
{
    bHalUartSend(HAL_LOG_UART, &cmd, 1);
}
//串口接收空闲
int UartIdleCallback(uint8_t *pbuf, uint16_t len)
{
    bYmodemParse(pbuf, len);
    return 0;
}
//串口接收实例
BUTIL_UART_INSTANCE(YmodemRec, 1128, 50, UartIdleCallback);

int main()
{
    ...
    bInit();
    bYmodemInit(YModemCallback, YmodemSend);

    //启动传输
    bYmodemStart();
    ...
}

void USART1_IRQHandler()
{
    uint8_t uart_dat = 0;
    if (USART_GetITStatus(USART1, USART_IT_RXNE) == SET)
    {
        USART_ClearITPendingBit(USART1, USART_IT_RXNE);
        uart_dat = USART_ReceiveData(USART1);
        butilUartRxHandler(&YmodemRec, uart_dat);
    }
}
```

```
}  
}
```

6.17 b_mod_iap

详细介绍:

<https://gitee.com/notrynohigh/BabyOS/wikis/BabyOS固件升级功能>

6.17.1 数据结构

```
/**  
 * IAP状态，IAP介绍文档中有状态的切换路径  
 */  
#define B_IAP_STA_NULL (0)  
#define B_IAP_STA_START (1)  
#define B_IAP_STA_READY (2)  
#define B_IAP_STA_FINISHED (3)  
#define IS_IAP_STA(s)  
    \  
    ((s) == B_IAP_STA_NULL) || ((s) == B_IAP_STA_START) || ((s) ==  
B_IAP_STA_READY) || \  
    ((s) == B_IAP_STA_FINISHED))  
  
#define B_IAP_FILENAME_LEN (64)    // 固件名的长度限制  
#define B_IAP_FAIL_COUNT (3)      // 固件失败的次数限制  
#define B_IAP_BACKUP_EN (0x55)    // 备份固件功能启用的标志  
#define B_IAP_BACKUP_VALID (0xAA) // 存在有效备份固件的标志  
  
typedef struct  
{  
    char    name[B_IAP_FILENAME_LEN]; // 固件名，限制在64个字符  
    uint32_t version;                 // 固件版本号  
    uint32_t len;                     // 固件长度  
    uint32_t crc;                     // 固件数据的校验值  
    uint32_t crc_type;                // CRC类型，参考algorithm/inc/algo_crc.h  
} bIapFwInfo_t;  
  
typedef struct  
{  
    uint8_t flag; // 备份标志，0xAA表示存在有效备份  
    uint32_t fcrc; // 备份区固件的crc32校验值  
    uint32_t second; // 运行多少秒后进行备份  
} bIapBackupInof_t;  
  
typedef struct  
{  
    int    stat;  
    int    fail_count;  
    uint32_t cache_dev;  
    uint32_t backup_dev;  
    bIapFwInfo_t info;  
    bIapBackupInof_t backup;  
    uint32_t percentage;  
    uint32_t fcrc; //crc
```

```
} bIapFlag_t;
```

6.17.2 接口介绍

```
/**
 * 跳转是弱函数，用户可自己实现
 */
void bIapJump2Boot(void);
void bIapJump2App(void);

/**
 * \brief 初始化函数
 * \param cache_dev_no 缓存固件的设备号，若不需要缓存则传入0
 * \param backup_dev_no 备份固件的设备号，若不需要备份固件则传入0
 * \param backup_time_s 运行多少s后备份固件，若不需要备份固件则忽略
 */
int bIapInit(uint32_t cache_dev_no, uint32_t backup_dev_no, uint32_t
backup_time_s);

/**
 * \brief 事件处理函数
 * \param event \ref bIapEvent_t
 * \param arg
 */
int bIapEventHandler(bIapEvent_t event, void *arg);

/**
 * 查询当前IAP的状态
 * 应用程序，查询到是B_IAP_STA_READY状态，则跳转至启动程序
 */
uint8_t bIapGetStatus(void);

/**
 * 查询备份固件是否有效
 */
uint8_t bIapBackupIsValid(void);

/**
 * 查询升级进度
 */
uint8_t bIapPercentage(void);
```

6.17.3 使用例子

https://gitee.com/notrynohigh/BabyOS_Example/tree/BearPi/

例程仓库小熊派分支，利用BabyOS通用协议专用上位机进行固件升级

https://gitee.com/notrynohigh/BabyOS_Protocol

6.18 b_mod_state

此功能模块提供给用户进行应用的开发：

状态：进入状态时执行的函数、在此状态下执行的函数、离开状态时执行的函数

事件：每个状态可附加一个事件表：事件&事件处理函数

6.18.1 数据结构

```
typedef void (*pStateEventHandler_t)(uint32_t event, void *arg);
typedef void (*pStateEnterHandler_t)(uint32_t pre_state);
typedef void (*pStateExitHandler_t)(void);
typedef void (*pStateHandler_t)(void);

typedef struct
{
    uint32_t          event;
    pStateEventHandler_t handler;
} bStateEvent_t;

typedef struct
{
    bStateEvent_t *p_event_table;
    uint32_t      number;
} bStateEventTable_t;

typedef struct
{
    uint32_t          state;
    pStateEnterHandler_t enter;
    pStateExitHandler_t exit;
    pStateHandler_t     handler;
    bStateEventTable_t  event_table;
} bStateInfo_t;
```

6.18.2 接口介绍

```
#define bSTATE_REG_INSTANCE(state_info)    //注册状态信息

int bStateTransfer(uint32_t state);          //切换状态
int bStateInvokeEvent(uint32_t event, void *arg); //触发事件
int bGetCurrentState(void);                 //获取当前状态
```

6.18.3 使用例子

https://gitee.com/notrynohigh/BabyOS_Example

7.工具模块

7.1 b_util_at

7.1.1 数据结构

```
//at的回调, id: 调用AT发送后返回的id result:运行的结果
typedef void (*bAtCallback_t)(uint8_t id, uint8_t result);

#define AT_INVALID_ID (0xFF)

#define AT_STA_NULL (0)
#define AT_STA_OK (1)
#define AT_STA_ERR (2)
#define AT_STA_ID_INVALID (3)
```

7.1.2 接口介绍

```
int bAtGetStat(uint8_t id);
int bAtRegistCallback(bAtCallback_t cb);
//将接收的数据喂给模块
int bAtFeedRespData(uint8_t *pbuf, uint16_t len);
//AT发送指令, 发送的指令会放入队列, 并返回id。
//pcmd: at指令 cmd_len:指令长度 presp:期待的回复内容 resp_len:回复内容的长度
//uart:串口号 timeout:允许的超时时间
int bAtCmdSend(const char *pcmd, uint16_t cmd_len, const char *presp, uint16_t resp_len, uint8_t uart, uint32_t timeout);
```

7.2 b_util_fifo

7.2.1 数据结构

```
typedef struct
{
    uint8_t *      pbuf;
    uint16_t       size;
    volatile uint16_t r_index;
    volatile uint16_t w_index;
} bFIFO_Info_t;
typedef bFIFO_Info_t bFIFO_Instance_t;
//创建fifo实例
#define bFIFO_INSTANCE(name, _fifo_size) \
    static uint8_t  fifo##name[_fifo_size]; \
    bFIFO_Instance_t name = {.pbuf = fifo##name, .size = _fifo_size, .r_index = \
    0, .w_index = 0};
```

7.2.2 接口介绍

```
//FIFO的常用操作
int bFIFO_Length(bFIFO_Instance_t *pFIFO_Instance, uint16_t *plen);
int bFIFO_Flush(bFIFO_Instance_t *pFIFO_Instance);
int bFIFO_Write(bFIFO_Instance_t *pFIFO_Instance, uint8_t *pbuf, uint16_t size);
int bFIFO_Read(bFIFO_Instance_t *pFIFO_Instance, uint8_t *pbuf, uint16_t size);
```

7.3 b_util_i2c

7.3.1 数据结构

```
//模拟I2C的GPIO定义
typedef struct
{
    bHalGPIOInstance_t sda;
    bHalGPIOInstance_t clk;
} bUtilI2C_t;
```

7.3.2 接口介绍

```
//模拟I2C的常用操作
void bUtilI2C_Start(bUtilI2C_t i2c);
void bUtilI2C_Stop(bUtilI2C_t i2c);
int bUtilI2C_ACK(bUtilI2C_t i2c);
void bUtilI2C_mACK(bUtilI2C_t i2c);

void bUtilI2C_WriteByte(bUtilI2C_t i2c, uint8_t dat);
uint8_t bUtilI2C_ReadByte(bUtilI2C_t i2c);

int bUtilI2C_WriteData(bUtilI2C_t i2c, uint8_t dev, uint8_t dat);
uint8_t bUtilI2C_ReadData(bUtilI2C_t i2c, uint8_t dev);

int bUtilI2C_ReadBuff(bUtilI2C_t i2c, uint8_t dev, uint8_t addr, uint8_t *pdat,
uint8_t len);
int bUtilI2C_WriteBuff(bUtilI2C_t i2c, uint8_t dev, uint8_t addr, const uint8_t
*pdat, uint8_t len);
```

7.4 b_util_spi

7.4.1 数据结构

```
//模拟SPI的GPIO定义和SPI参数
typedef struct
{
    bHalGPIOInstance_t miso;
    bHalGPIOInstance_t mosi;
    bHalGPIOInstance_t clk;
    uint8_t          CPOL;
    uint8_t          CPHA;
} bUtilSPI_t;
```

7.4.2 接口介绍

```
//模拟SPI的读写操作
uint8_t bUtilSPI_WriteRead(bUtilSPI_t spi, uint8_t dat);
```

7.5 b_util_log

在b_hal_if定义log输出的串口号。

7.5.1 接口介绍

```
#define b_log_i(...)
#define b_log_w(...)
#define b_log_e(...)
#define b_log(...)
```

7.6 b_util_lunar

7.6.1 数据结构

```
//阴历数据结构
typedef struct
{
    uint16_t year;
    uint8_t  month;
    uint8_t  day;
} bLunarInfo_t;
```

7.6.2 接口介绍

```
//阳历转阴历
int bSolar2Lunar(uint16_t syear, uint8_t smonth, uint8_t sday, bLunarInfo_t
*pLunar);
```

7.7 b_util_memp

7.7.1 数据结构

```
//需要监控的信息，unused_unit 统计最小未使用量
typedef struct
{
    uint16_t unused_unit;
} bMempMonitorInfo_t;
//内存链表
typedef struct bMempList
{
    uint8_t *p;
    uint32_t total_size;
    uint32_t size;
    struct bMempList *next;
    struct bMempList *prev;
} bMempList_t;
```

7.7.2 接口介绍

```
//申请和释放空间
void *bMalloc(uint32_t size);
void bFree(void *paddr);

#if _MEM_MONITOR_ENABLE
void bMempGetMonitorInfo(bMempMonitorInfo_t *pinfo);
#endif
//内存链表初始化
int bMempListInit(bMempList_t *phead);
//申请空间存p指向的数据，再将此次申请的空间放入链表
int bMempListAdd(bMempList_t *phead, uint8_t *p, uint32_t len);
//释放链表中所有动态申请的内存
int bMempListFree(bMempList_t *phead);
//内存链表里存储的数据转为连续内存存储
uint8_t * bMempList2Array(const bMempList_t *phead);
```

7.8 b_util_uart

7.8.1 数据结构

```
//串口接收空闲的回调
typedef int (*pbUartIdleCallback_t)(uint8_t *pbuf, uint16_t len, void *arg);

typedef struct UtilUart
{
    uint8_t *pbuf;
    uint16_t buf_size;
    volatile uint16_t index;
    uint32_t idle_thd_ms;
    pbUartIdleCallback_t callback;
    void *cb_arg;
    uint32_t l_tick;
    uint32_t l_index;
    struct UtilUart *next;
```

```

    struct UtilUart    *prev;
} butilUart_t;

typedef butilUart_t butilUartInstance_t;

//用于创建串口接收实例
#define BUTIL_UART_INSTANCE(name, buf_len, idle_ms, cb, arg) \
    static uint8_t      Buf##name[buf_len];                \
    butilUartInstance_t name = {                            \
        .pbuf           = Buf##name,                      \
        .buf_size       = buf_len,                        \
        .idle_thd_ms    = idle_ms,                       \
        .callback       = cb,                             \
        .cb_arg         = arg,                            \
        .index          = 0,                              \
        .l_tick         = 0,                              \
        .l_index        = 0,                              \
        .prev           = NULL,                           \
        .next           = NULL,                           \
    }

```

7.8.2 接口介绍

```

//    初始化用户定义的butilUartInstance_t（没有使用BUTIL_UART_INSTANCE去定义实例的情况）
void butilUartInitStruct(butilUartInstance_t *pinstance, uint8_t *pbuf, uint16_t
size,
                        uint32_t idle_ms, pbUartIdleCallback_t cb, void *arg);

//    将串口号绑定到已有实例，绑定后可以用 butilUartRxHandler2和butilUartReceivedSize2
void butilUartBind(uint8_t uart_no, butilUartInstance_t *pinstance);

//    butilUartRxHandler 和 butilUartRxHandler2 效果是一样
//    但是，只有通过butilUartBind绑定串口号，才能调用butilUartRxHandler2
void butilUartRxHandler(butilUartInstance_t *pinstance, uint8_t dat);
void butilUartRxHandler2(uint8_t uart_no, uint8_t dat);

//    获取当前BUF中已经收到的数据长度
uint16_t butilUartReceivedSize(butilUartInstance_t *pinstance);
uint16_t butilUartReceivedSize2(uint8_t uart_no);

```

7.9 b_util_utilc

7.9.1 数据结构

```
typedef struct
{
    uint16_t year;
    uint8_t month;
    uint8_t day;
    uint8_t week;
    uint8_t hour;
    uint8_t minute;
    uint8_t second;
} bUTC_DateTime_t;

typedef uint32_t bUTC_t;
```

7.9.2 接口介绍

```
//UTC与时间结构相互转换
//UTC的起始时间是2000年1月1日0时0分0秒
void    bUTC2Struct(bUTC_DateTime_t *tm, bUTC_t utc);
bUTC_t  bStruct2UTC(bUTC_DateTime_t tm);
```

8. 参与开发

目前还需要广大开源爱好者的加入，将货架做稳固，再填充高质量的货物。

<https://gitee.com/notrynohigh/BabyOS>（主仓库）

<https://github.com/notrynohigh/BabyOS>（自动同步）

管理员邮箱：notrynohigh@outlook.com

开发者基于<https://gitee.com/notrynohigh/BabyOS>仓库dev分支进行。

有贡献的开发者（不局限于提交代码），记录到<http://babyos.cn/>网站Team页面。

有意者随时私信联系！