BabyOS 设计和使用手册

源码地址

https://gitee.com/notrynohigh/BabyOS（主）

https://github.com/notrynohigh/BabyOS

开发者 QQ 群

管理员邮箱

notrynohigh@outlook.com

修订记录

时间	记录	修订人
2020.02.19	1.创建文档	notrynohigh
2020.02.24	1.更新功能模块描述	notrynohigh
2020.03.05	1.新增 gui，log,menu 的描述	notrynohigh
2020.03.11	1.改变文档结构	notrynohigh
2020.03.18	1.填补空白部分	notrynohigh
2020.04.03	1.更新设计部分内容	notrynohigh
2020.05.05	1.增加中断的描述	notrynohigh
2020.05.24	1.修改 ctl 描述	notrynohigh
2020.05.25	1.对功能模块介绍进行完善	notrynohigh
2020.05.26	1.设备注册原理描述	Alex
2020.05.31	1. 更新功能模块和驱动描述 2. 更新中断部分描述	notrynohigh
2020.06.04	1.增加文件系统描述	notrynohigh
2021.06.13	1.V6 版本各个版本描述	notrynohigh

目录

1. 引言

BabyOS 是为 MCU 裸机项目而生，主要有驱动和功能模块两个主要部分。V3.0.0 之后增加硬件抽象层，使代码更具框架性。本文档介绍 BabyOS 的设计以及使用方法，作为开发者优化代码框架和新增代码的参考。

2. 开发组成员

Notrynohigh

Alex

LiuWei

姜先生

不愿透漏姓名的王年年

超级布灵的小星星

Cloud

段仁胜

Illusion

绿色心晴

Lyping

Murphy

嵌入式_蓝莲花

思无邪

无诚无成

.

更多请加入开发者群进行查看....

3. 设计思路

BabyOS 的定位是做一个带框架的功能及设备驱动库。小项目推荐使用 BabyOS 作为整个项目的框架，以搭积木的方式缩短开发周期。复杂项目使用操作系统，也可以将 BabyOS 作为功能和驱动库使用。

工程师做几个项目后可以回顾一下，项目之间都存在着可以复用的功能，使用的驱动也有重复的。如果这些重复的部分可以放入一个框架中存放起来，再通过自身积累或者工程师之间资源共享，功能模块和驱动积累到一定程度后,在开始一个新项目时便可以通过搭积木的方式完成一部分的工作，缩短项目开发时间。BabyOS 便是要做这样的一个框架存放功能模块和驱动。

BabyOS 提供统一的驱动接口，应用中通过设备号来操作对应的设备。低功耗的应用场景中，驱动操作一般时唤醒，数据交互，休眠。这个模式和对文件的操作很像，打开，编辑，关闭。因此 BabyOS 选择使用类文件操作方式。下文进行具体的介绍。

3.1. 代码结构

图 3-1 BabyOS 代码目录

bos/algorithm 常用算法，将需要的添加至工程

bos/core/ 核心文件全部添加至工程

~~*bos/config/* 配置文件及设备列表文件，全部添加至工程~~

配置文件和设备注册文件一起作为单独仓库存放，BabyOS_Config

bos/driver/ 选择驱动添加，在 b_hal_if.h 内添加驱动的接口

~~*bos/hal/hal/* 硬件抽象层，将需要的文件添加至工程，根据平台进行修改~~

硬件抽象层已作为单独的仓库，方便收集各个平台的代码。

bos/utils/ 底层实用代码，全部添加至工程

bos/modules/ 功能模块，全部添加至工程

bos/thirdparty/ 第三方开源代码，将需要的添加至工程

BabyOS V6.0.0 后的版本，集成开发环境中指定头文件路径时，只需要添加 BabyOS/bos/ 路径即可。

BabyOS_Hal 仓库的代码，添加 BabyOS_Hal/inc/

BabyOS_Config 仓库的代码。添加 BabyOS_Config/

3.2. 代码框图

图 3-2 代码框图

3.3. 设备驱动

3.3.1. 驱动结构

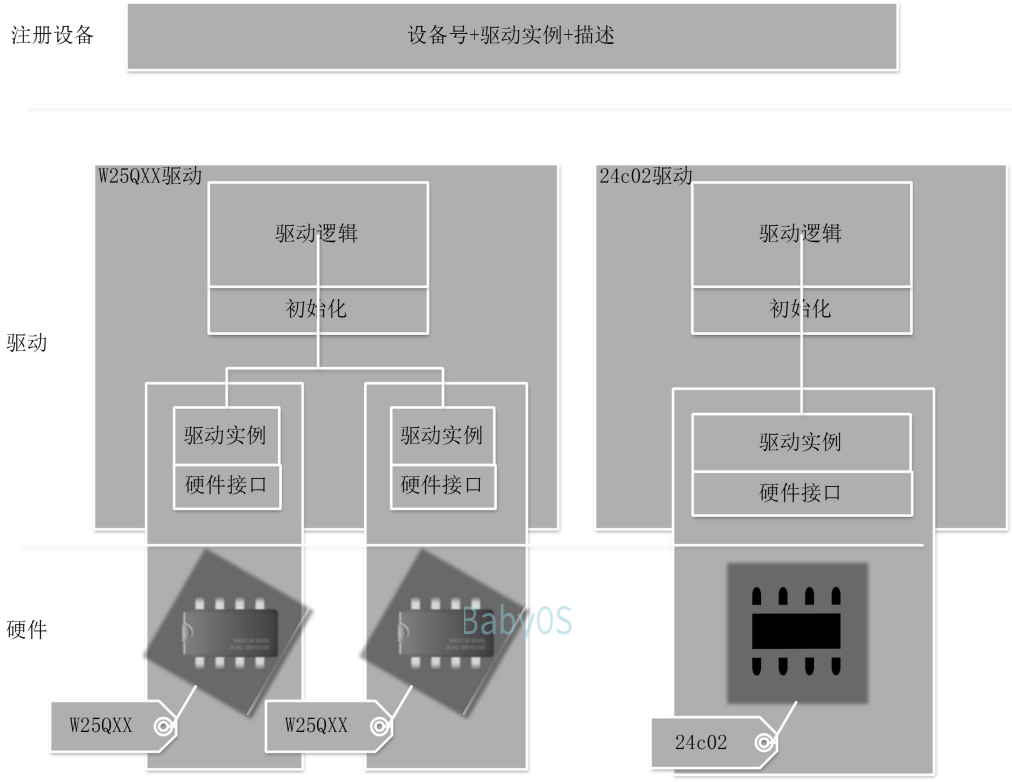

图 3-3 驱动结构

从图中可以看出整个驱动结构会涉及 4 个部分：

驱动逻辑	满足统一接口而编写的驱动代码：读、写、打开、关闭和控制
驱动实例	`bDriverInterface_t` 类型的全局变量，`b_driver.h` 中声明
初始化	填充驱动实例内的各个元素，以及完成必要的初始化操作
硬件接口	当前驱动使用到的硬件接口，如 `SPI`、`GPIO`、`IIC` 等

表 3-1 驱动各部分描述

3.3.2. 驱动逻辑

驱动逻辑同样是为满足统一结构而编写的驱动代码，驱动统一接口如下：

```
typedef struct bDriverIf
{
    int status;
    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,
uint16_t len);
    int (*read)(struct bDriverIf *pdrv, uint32_t offset, uint8_t *pbuf,
uint16_t len);
    void *_hal_if;
    union
    {
        uint32_t v;
        void *_p;
    }_private;
}bDriverInterface_t;
```


1）**status** 在驱动初始化时，若初始化碰到异常则将 status 设为-1 反之设为 0。 在操作设备是会检测此项，如果是-1 则不执行。

2）**open** 主要是负责唤醒的操作，在此处执行设备唤醒。如果设备没有休眠状态，open 赋值为 NULL 即可。

3）**close** 主要负责休眠的操作，在此处执行设备休眠。如果设备没有休眠状态，close 赋值为 NULL 即可。

4）**ctl** 主要负责对设备进行配置或者执行特定的操作，例如擦除，切换状态等等。ctl 的调用需要传入指令 cmd 和对应的参数*param。 因为不同类别的驱动有不同的指令，例如 flash 类驱动需要擦除，而 3 轴传感不需要， 所以驱动是分类存放，bos/driver/inc 目录 b_drv_class_xxx.h 文件是 xxx 类别的公用头文件，这里面有此类驱动支持的 cmd 以及对应参数的数据类型。 执行指令成功则返回 0，失败或者是不支持的指令返回-1。

5）**write** 主要负责传输数据至设备，例如存储设备，则传入要存储的数据，通讯模组则 传入要发送的数据。 执行成功则返回实际发送的数据长度，执行失败

则返回-1。

6）**read** 主要负责从设备获取数据，那么获取数据的最小单元依据设备的功能而定，例如，存储设备，最小可以获取 1 个字节；3 轴加速度设备，则最小单元为 3 个加速度值；温湿度传感器则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。

7）**_hal_if** 指向当前驱动对应的硬件接口的指针，驱动操作硬件时需要使用已绑定的硬件接口。 在 b_driver.h 里提供一个方便获取 pdrv 里硬件接口的宏： `#define bDRV_GET_HALIF(name, type, pdrv) type *name = (type *)(pdrv->_hal_if)`

8）**_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。

3.3.3. 硬件接口

驱动代码的 h 文件中定义了当前设备需要的接口情况。

统一的名命格式：bDRIVER_HalIf_t

根据硬件接口的数据结构，在 b_hal_if.h 中定义硬件接口实例，

统一的宏名称格式：HAL_DRIVER_IF

以上两个名命格式中的 DRIVER 指各类驱动的名称，例如 SPIFLASH

：bSPIFLASH_HalIf_t HAL_SPIFLASH_IF

硬件接口有两种情况，根据图 3-3 可以看出硬件接口有两种情况，

1）存在有相同设备的情况，例如接入 MCU 的有两个 Flash 芯片

2）当前驱动对应的设备不会存在多个，例如屏，一般只会接 1 块屏

以上两种情况的区别是定义硬件接口的方式。

第一种情况按照如下方式定义：

在驱动 h 文件定义硬件接口数据类型例如：

```
typedef struct
{
    union
    {
        bHalQSPINumber_t qspi;
        bHalSPINumber_t  spi;
    } _if;
    bHalGPIOInstance_t cs;
    uint8_t           is_spi;
} bSPIFLASH_HalIf_t;
```


1）在驱动 C 文件定义硬件接口的数组，例如：

```
static const bSPIFLASH_HalIf_t
bSPIFLASH_HalIfTable[HAL_SPIFLASH_TOTAL_NUMBER] = HAL_SPIFLASH_IF;
```

第二种情况只需要将硬件接口在 b_hal_if.h 内定义一份即可。

```
#define HAL_XPT2046_IF              \
    {                              \
        B_HAL_SPI_3,                \
        {                          \
            B_HAL_GPIOC, B_HAL_PIN9 \
        }                          \
    }
```

3.3.4. 驱动实例

如果需要支持多个相同芯片，那么驱动实例要定义为数组例如：

```
bSPIFLASH_Driver_t bSPIFLASH_Driver[HAL_SPIFLASH_TOTAL_NUMBER];
```

根据用户定义的硬件接口数量来定驱动实例数组的大小。驱动实例在 b_driver.h 进行声明：

```
extern bDriverInterface_t   bSPIFLASH_Driver[];
```

这种情况注册设备时按照如下方式：

```
B_DEVICE_REG(SPIFLASH0, bSPIFLASH_Driver[0], "flash0")
B_DEVICE_REG(SPIFLASH1, bSPIFLASH_Driver[1], "flash1")
```

3.3.5. 初始化

初始化函数主要干如下几件事：

1）将驱动实例和硬件接口绑定（多个相同芯片的情况）

2）执行必要的初始化程序

3）填充驱动实例的元素 4

4）更新状态：正常或者异常

以 SPIFLASH 为例：

```
int bSPIFLASH_Init()
{
    size_t i = 0, number = sizeof(bSPIFLASH_HalIfTable) /
sizeof(bSPIFLASH_HalIf_t);
    int retval = 0;
    for(i = 0;i < number;i++)
    {
        sprintf(bSPIFlashName[i], "%03d", i);
        flash_table[i].name = bSPIFlashName[i];
        flash_table[i].spi._hal_if =
                 (void *)&bSPIFLASH_HalIfTable[i];

        bSPIFLASH_Driver[i]._hal_if =
               (void *)&bSPIFLASH_HalIfTable[i];
        bSPIFLASH_Driver[i].open = _bSPIFLASH_Open;
        bSPIFLASH_Driver[i].close = _bSPIFLASH_Close;
        bSPIFLASH_Driver[i].ctl = _bSPIFLASH_Ctl;
        bSPIFLASH_Driver[i].read = _bSPIFLASH_ReadBuf;
        bSPIFLASH_Driver[i].write = _bSPIFLASH_WriteBuf;
        bSPIFLASH_Driver[i].status = 0;
        bSPIFLASH_Driver[i]._private._p = &flash_table[i];
    }

    for(i = 0;i < number;i++)
    {
        _bSPIFLASH_Open(&bSPIFLASH_Driver[i]);    //wakeup flash
    }

    if(sfud_init() != SFUD_SUCCESS)
    {
        for(i = 0;i < number;i++)
        {
```

```
            bSPIFLASH_Driver[i].status = -1;
        }
        retval = -1;
    }

    for(i = 0;i < number;i++)
    {
        _bSPIFLASH_Close(&bSPIFLASH_Driver[i]); //powerdown
    }

    return retval;
}
```

最后注册驱动初始化函数，即在驱动 C 文件添加如下代码：

```
bDRIVER_REG_INIT(bSPIFLASH_Init);
```

3.3.6. 设备接口

```
int bOpen(uint8_t dev_no, uint8_t flag);
int bRead(int fd, uint8_t *pdata, uint16_t len);
int bWrite(int fd, uint8_t *pdata, uint16_t len);
int bCtl(int fd, uint8_t cmd, void *param);
int bLseek(int fd, uint32_t off);
int bClose(int fd);
```

上面这组 API 是对设备的操作。每个设备必须先打开后再进行读写及控制，操作完成后关闭设备。

打开设备后返回一个句柄，余下的操作根据句柄进行。打开设备时使用的 dev_no 是在 b_device_list.h 中注册的设备。

提供 *bCoreIsIdle* 供用户使用查看当前是否所有设备处于空闲状态。

```
int bCoreIsIdle(void);
```

3.3.7. 设备注册

b_device_list.h 内通过如下宏进行设备注册，三个参数分别为设备名（也称设备号），设备驱动，设备描述，以 SPIFLASH 为例：

*B_DEVICE_REG(*SPIFLASH*, bSPIFLASH_Driver[0], "flash")*

下面通过预处理分析设备注册的原理：

设备管理有以下几个重要结构：

【设备号】

```
typedef enum
{
    #define B_DEVICE_REG(dev, driver, desc)    dev,
    #include "test.h"
    bDEV_MAX_NUM
}bDeviceName_t;
```

【驱动接口】

```
static bDriverInterface_t* bDriverTable[bDEV_MAX_NUM] = {
    #define B_DEVICE_REG(dev, driver, desc)    &driver,
    #include "b_device_list.h"
};
```

【设备描述】

```
static const char *bDeviceDescTable[bDEV_MAX_NUM] = {
    #define B_DEVICE_REG(dev, driver, desc)    desc,
    #include "test.h"
};
```

【测试代码 test.c】

```
typedef struct bDriverInterface
{
    ....
}bDriverInterface_t;

typedef enum
{
    #define B_DEVICE_REG(dev, driver, desc)    dev,
    #include "b_device_list.h"
    bDEV_MAX_NUM
}bDeviceName_t;

static bDriverInterface_t   bNullDriver;
static bDriverInterface_t   F8l10dDriver;

static bDriverInterface_t* bDriverTable[bDEV_MAX_NUM] = {
    #define B_DEVICE_REG(dev, driver, desc)    &driver,
    #include "b_device_list.h"
};

static const char *bDeviceDescTable[bDEV_MAX_NUM] = {
    #define B_DEVICE_REG(dev, driver, desc)    desc,
    #include "b_device_list.h"
};
```

【测试代码 b_device_list.h】

```
B_DEVICE_REG(SPIFLASH, bSPIFLASH_Driver[0], "flash")
B_DEVICE_REG(LoRaModule, F8l10dDriver, "lora")
#undef B_DEVICE_REG
```

【GCC 进行预处理】

```
gcc -E test.c -o test.i
```

【宏展开后代码】

```
# 1 "test.c"
# 1 "<built-in>"
# 1 "<command-line>"
# 1 "test.c"
typedef struct bDriverInterface
{
    ....
}bDriverInterface_t;

typedef enum
{
# 1 "b_device_list.h" 1
SPIFLASH,
LoRaModule,
# 16 "test.c" 2
    bDEV_MAX_NUM
}bDeviceName_t;

static bDriverInterface_t bSPIFLASH_Driver[0];
static bDriverInterface_t F8l10dDriver;

static bDriverInterface_t* bDriverTable[bDEV_MAX_NUM] = {
# 1 "b_device_list.h" 1
&bSPIFLASH_Driver[0],
&F8l10dDriver,
# 25 "test.c" 2
};

static const char *bDeviceDescTable[bDEV_MAX_NUM] = {
# 1 "b_device_list.h" 1
"flash",
"lora",
# 30 "test.c" 2
};
```

对比宏展开前后的代码可以得到如下结论：

注册的设备被宏展开为 enum 类型(bDeviceName_t)，使用 bDEV_MAX_NUM 可以获取设备数量， 驱动接口表(bDriverTable)里面宏展开为设备对应的驱动结构体指针，设备描述数组(bDeviceDescTable)中宏展开为字符串.

所以使用了 B_DEVICE_REG(dev, driver, desc) 注册的驱动会自动在这三个数据结构中填充内容，三个数据结构中的顺序是一一对应的，最后 BOS 中的设备接口(比如 bOpen) 就可以使用 enum 设备号引用设备名和调用设备对应的驱动函数接口。

3.4. 功能模块设计

每个功能模块只做成一个功能，可通过配置文件对其进行 ENABLE/DISABLE，增加一项功能模块需要在 b_config.h 中增加一项开关。大致会有如下几种特性的功能模块：

1) 用户主动调用功能模块提供的 API

2) 用户提供回调函数，由功能模块调用回调

当功能模块要使用硬件资源时，提供 API 给用户，让其指定设备号。用户指定的设备号是 3.3.7 章节提到的在 b_device_list.h 中注册的设备号。由于操作设备的接口是统一的，那么知道设备号后，功能模块便知道怎么操作设备了。

在编写功能模块时，如果模块中有需要轮询的函数时，通过如下宏处理轮询函数，其中 xxxxxx 是轮询的函数：

```
BOS_REG_POLLING_FUNC(xxxxxx);
```

使用功能模块是先将 b_config.h 内对应的宏打开。

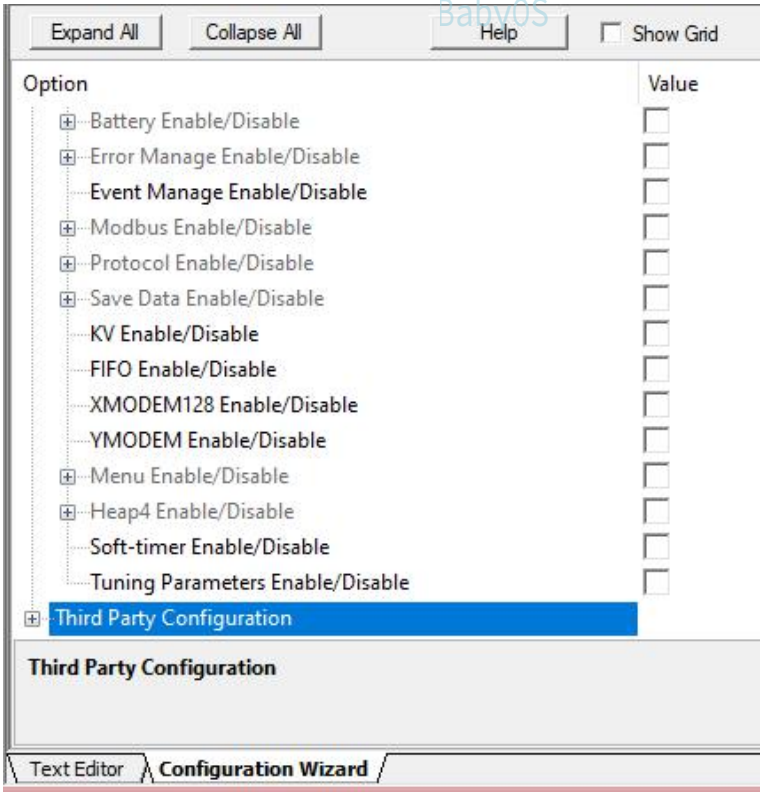

图 4-4 配置页面

3.4.1. 电量检测

电量检测核心是 ADC 的测量，这部分及其依赖于硬件，所以 AD 转换获取电压的函数需要用户提供。

int bBatteryInit(pBatteryGetmV_t f);

根据配置的检测间隔时间，每次测量采样 5 次，去掉最大最小再取平均值得到最后的结果。根据阈值更新电池状态，正常或者低电量。

获取状态：*uint8_t bBatGetStatus(void);*

获取电压值：*uint16_t bBatGetVoltageValue(void);*

3.4.2. 按键功能模块

按键功能模块是基于第三方代码 FlexibleButton 完成，在其之上封装了一层，让用户使用起来更加简单。

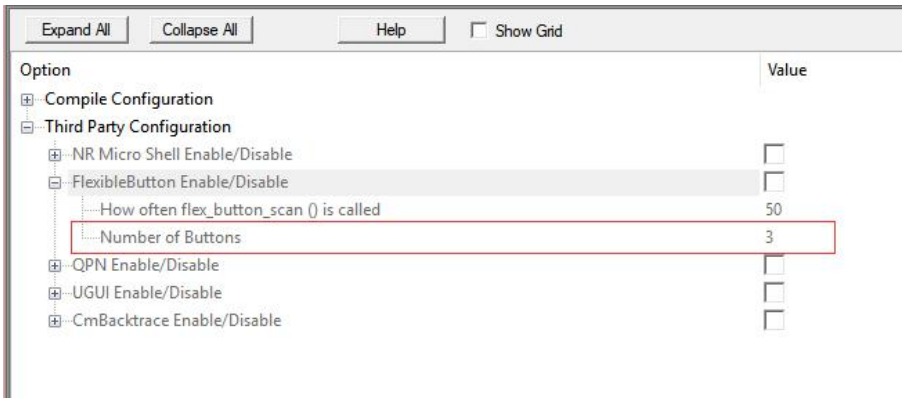

图 3-5 Button 配置

首先配置时填写按键数量。

在 b_hal_if.h 里是硬件的定义，里面有如下宏是用于定义按键引脚的：

```
#if _FLEXIBLEBUTTON_ENABLE
#define HAL_B_BUTTON_GPIO                \
    {                                    \
        {B_HAL_GPIOA, B_HAL_PIN0, 0}, \
    }
#endif
```


调用初始化函数,指定短按、长按、超长按分别对应的时间 ms：

```
int   bButtonInit(uint16_t short_xms, uint16_t long_xms, uint16_t llong_xms);
```

为按键注册事件：

```
void bButtonRegEvent(uint8_t id, uint16_t event, pBtnEventHandler_t handler);
```

按键的 ID 则是定义引脚时的顺序，从 0 开始，按键的事件则参考如下枚举类型：

```
#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)
```

根据事件对应的值便可以看出，一个按键可以注册多个事件。例如单击和短按，注册时传入 *BTN_EVENT_CLICK | BTN_EVENT_SHORT*

3.4.3. 错误管理

错误管理功能模块，当系统产生故障后将错误提交到错误管理，参数分为是故障码，故障产生的时间，处理相同故障的最小间隔时间，故障等级。

处理相同故障最小间隔时间，举例说明，当电池电量低时，更换电池之前，每次检测电量都会判断为低电量，如果检测电量的周期是 1 分钟，处理相同故障最小间隔时间是 1 小时，处理故障的方式是上报服务器。这种情况下，低电量故障信息时每小时上报一次而不是每分钟。

注册故障码至错误管理单元的 API 如下所示：

```
int bErrorRegist(uint8_t err,
                 uint32_t utc,
                 uint32_t interval,
                 uint32_t level);
```

处理故障的方式，在初始化错误管理的时候指定：

```
int bErrorInit(pecb cb);
```


错误分了两个等级，等级 0 和等级 1。其区别说明如下：

如果处理故障的方式是上报服务器。故障等级为 0 时，那么上报一次后则不再处理，直到下一次注册故障。如果故障等级为 1 时，上报服务器后，如果服务器没有回复，那么过了最小间隔时间后会再次上传。

上报故障后服务器回复，则可以调用如下 API：

```
int bErrorAck(uint8_t e_no);
```

更多关于错误管理的 API 可以查看 b_mod_error.h

3.4.4. 事件管理

防止全局变量满天飞，设计了这个事件管理功能模块。将某个特定事件需要执行的函数注册至事件管理，当事件发生后调用触发函数即可。

首先定义一个事件实例：

```
bEVENT_INSTANCE(name)
```

通过如下函数注册事件，注册主要是为了将实例放入事件链表，注册是携带参数实例指针以及事件产生后的处理函数：

```
int bEventInit(bEventInstance_t *pInstance,
               pEventHandler_t handler);
```

事件产生后调用如下函数：

```
int bEventTrigger(bEventInstance_t *pInstance);
```

3.4.5. GUI 功能模块

GUI 功能模块是基于 uGUI 完成，首先进行配置：

图 3-6 GUI 配置

其中 X，Y 的物理尺寸按照竖屏时的尺寸进行配置。触屏的 AD 值同样是按照竖屏的尺寸配置。LCD_DISPLAY_MODE 配置显示方式：横屏或者竖屏。

调用初始化，指定 LCD 和触屏芯片的设备号（b_device_list.h）

```
int bGUI_Init(int lcd, int touch);
```

初始化后便可以使用 uGUI 的 API，具体 API 可以查看 ugui.h。

uGUI 使用教程查看：

bos/thirdparty/UGUI/Reference Guide ugui v0.3.pdf

3.4.6. MODBUS RTU

目前的 Modbus-rtu 协议 不是完整的，只是完成了 Modbus 主机协议的功能码 03 和 16。提供的 API 中，2 个字节的数据都是小端模式，更利于 MCU 的代码编写。

使用前先定义一个 Modbus 实例，定义实例是指定发送数据的函数以及回调函数，回调函数主要是处理从机设备的回复：

```
bMODBUS_INSTANCE(name, pSendData, pCallback)
```

读取从机寄存器数据，参数需要实例，从机地址，功能码，寄存器地址以及寄存器数量，具体 API 如下：

```
int bMB_ReadRegs(bModbusInstance_t *pModbusInstance,
                 uint8_t addr,
                 uint8_t func,
                 uint16_t reg,
                 uint16_t num);
```


写从机寄存器，参数需要实例，从机地址，功能码，寄存器地址，寄存器数量，要写入寄存器的数据，具体 API 如下：

```
int bMB_WriteRegs(bModbusInstance_t *pModbusInstance,
            uint8_t addr,
            uint8_t func,
            uint16_t reg,
            uint16_t num,
            uint16_t *reg_value);
```

当 MCU 接收到从机回复后将数据提交给功能模块进行解析，当解析成功后会将解析后的结果传入回调，提交数据的 API 如下：

```
int bMB_FeedReceivedData(bModbusInstance_t *pModbusInstance,
                         uint8_t *pbuf,
                        uint16_t len);
```

3.4.7. 私有协议

协议的格式：

头部	设备 ID	长度	指令	参数	校验
0xFE	2/4 字节可配	1/2 字节可配	1 字节	0~n 字节	1 字节

这部分的通用协议可配的部分在 b_config.h 里面进行配置。主要根据每次通信能发送的最长数据来选择长度字段的大小，根据一个网络内设备数量选择设备 ID 字段的大小。

将收到的数据给协议模块解析，解析完成后执行分发函数，因此最开始需要指定分发函数和设备 ID：

```
int bProtocolInit(bProtoID_t id, pdispatch f);
int bProtocolParse(uint8_t *pbuf, bProtoLen_t len);
```

如果有数据需要按照协议发送，当前的协议模块只提供组包的功能，不执行发送，因此在调用组包时需要提供一个 buffer 给功能模块使用，最后由用户自行发送 buffer 里面的数据。

```
int bProtocolPack(uint8_t cmd, uint8_t *param, bProtoLen_t param_size, uint8_t *pbuf);
```

3.4.8. 数据存储

数据存储功能模块，暂时提供了 2 种场景：

第一种场景（sda）：依靠时间存储相同大小的数据，例如每小时存储 20 个字节，存储 1 年。这种情况下会根据时间计算每个时间点存储的地址。

第二种场景(sdb)：在一块最小擦除单位存放一笔数据。存储时会额外加上校验，读取时会判断校验。

首先定义实例：

SDA 需要指定存储的间隔时间，总时间，每次存储的数据大小以及起始地址和最小擦除单位，设备号。根据上面场景介绍举例，间隔时间是 3600s,总时间是（366 * 24 * 60 * 60）s：

```
bSDA_INSTANCE(name,  _time_interval,  _total_time,  _data_size, _fbase_addr, _fsector_size, _dev_no)
```

SDA 是依赖于时间的，所以提供的 API 都是基于时间：

```
int bSDA_Write(bSDA_Instance_t *pSDA_Instance,
               uint32_t utc,
               uint8_t *pbuf);
int bSDA_Read(bSDA_Instance_t *pSDA_Instance,
              uint32_t utc,
              uint8_t *pbuf);
int bSDA_TimeChanged(bSDA_Instance_t *pSDA_Instance,
                     uint32_t o_utc,
                     uint32_t n_utc);
```

SDB 需要指定存储地址，存储数据大小以及设备号：

```
bSDB_INSTANCE(name, addr, _usize, _dev_no)
```

SDB 提供的是读写 API，当读取的数据校验不正确时返回-1

```
int bSDB_Write(bSDB_Instance_t *pSDB_Instance, uint8_t *pbuf);
int bSDB_Read(bSDB_Instance_t *pSDB_Instance, uint8_t *pbuf);
```

3.4.9. UTC 转换

UTC 时间时比较常用的，代码里面提供的 UTC 时间时基于 2000 年 1 月 1 日 0 点 0 分 0 秒。提供转换用的 API:

```
void bUTC2Struct( bUTC_DateTime_t *tm, bUTC_t utc );
bUTC_t bStruct2UTC( bUTC_DateTime_t tm);
```

3.4.10. FIFO

FIFO 功能模块提供了一组 API 按照 FIFO 的特性去操作一块内存。

先定义 FIFO 实例，指定实例名和 fifo 的大小：

```
bFIFO_INSTANCE(name, _fifo_size)
```

根据实例可以调用如下几个 API 完成 FIFO 的操作：

```
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);
```

3.4.11. 阳历阴历

提供 API 给用户使用，传入阳历的年月日得到阴历信息。

```
int bSolar2Lunar(uint16_t syear,
                 uint8_t smonth,
                 uint8_t sday,
                 bLunarInfo_t *plunar);
```

3.4.12. KV 键值对存储

键值对存储，已考虑 flash 寿命问题。将分配给 KV 的存储区域分配为 2 个区，索引区和数据区。通过索引区的信息去查找数据区存储的数据。

使用时首先进行初始化，指定设备号，起始地址，分配给 KV 的存储空间大小，最小擦除单位大小（如果不需要擦除的设备填 0）：

```
int bKV_Init(int dev_no,
        uint32_t s_addr,
        uint32_t size,
        uint32_t e_size);
```

初始化后便可以方便的进行键值对存储，提供如下 API：

```
int bKV_Set(const char *key, uint8_t *pvalue, uint16_t len);
int bKV_Get(const char *key, uint8_t *pvalue);
int bKV_Delete(const char *key);
```

新增和修改都是使用 bKV_Set。

BabyOS V6.0.0 开始对 KV 存储增加了 4 字节对齐，同时提供了使用 MCU 内部 FLASH 的驱动。因此可以基于 MCU 内部 FLASH 使用 KV 存储。

3.4.13. Xmodem128 和 Ymodem

Xmodem 和 Ymodem 完成了接收部分，采用同样的套路。

首先初始化，指定发送字节的函数以及回调函数。每收到一帧数据，解析成功后都会调用回调，回调参数 pbuf 为 NULL 时表示结束。初始化 API：

```
int bXmodem128Init(pcb_t fcb, psend fs);
int bYmodemInit(pymcb_t fcb, pymsend fs);
```

回调函数原型如下所示：

```
typedef void (*pcb_t)(uint16_t number, uint8_t *pbuf);
```

Xmodem128 每次传输 128 个字节，number 从 0 开始计数，通过 number 可以算出已接收多少个字节。

```
typedef void (*pymcb_t)(uint8_t t, uint8_t *pbuf, uint16_t len);
```

Ymodem 每次传输的数据 1k 或者 128 字节，同时 Ymodem 可以传输文件信息和文件数据。通过 t 可以分辨是文件信息（文件名和大小）还是文件数据。pbuf 对应的数据长度为 len。

这两种协议都需要接收端主动发起，所以设计启动函数：

```
int bXmodem128Start(void);
int bYmodemStart(void);
```

3.4.14. 打印日志 b_log

打印日志分等级，当调试的时候多点信息，开发的后期可以仅打印错误或者警告信息。打印数据的大小在 b_log.h 里面配置。

错误级别和警告级别都会额外打印函数名，行号。错误级别还多一项文件名。

3.4.15. 菜单程序

菜单的构建是基于页面与页面的关系，页面与页面之间是兄弟关系或者是父子关系。根据这两个关系便可以构建出整个多级菜单。因此设计两个 API：

```
int bMenuAddSibling(uint32_t ref_id, uint32_t id, pCreateUI f);
int bMenuAddChild(uint32_t ref_id, uint32_t id, pCreateUI f);
```

菜单构建完成后便是跳转，菜单模块提供了上、下、确定、返回四个动作，同时增加了跳转指定页面的 API，完成基本的需求。


```
void bMenuAction(uint8_t cmd);
void bMenuJump(uint32_t id);
```

每个页面都有唯一的 ID，因此直接跳转和设置页面是否可见都是通过 ID 对应到页面：

```
uint32_t bMenuCurrentID(void);
int bMenuSetVisible(uint32_t id, uint8_t s);
```

3.4.16. Shell 功能模块

Shell 功能模块是基于第三方代码完成。首先在配置文件中使能。

注册指令，注册时携带处理函数：

```
#define bSHELL_REG_INSTANCE(cmd_name, cmd_handler)
```

当串口收到数据后将数据给 Shell 解析，使用如下 API：

```
int bShellParse(uint8_t *pbuf, uint16_t len);
```

3.4.17. 硬件错误跟踪

硬件错误跟踪基于第三方代码完成，首先在配置文件中使能功能模块，并选择内核类型以及语言：

图 3-7 配置项

配置后调用初始化函数，参数为当前工程生成执行文件的文件名：

```
int bTraceInit(const char *pfw_name);
```

在硬件错误中断服务函数内调用如下 API：

```
void bHardfaultCallback(void);
```


当硬件错误发生后，会打印出一串指令，执行指令会有 addr2line.exe，这个工具在 *bos/thirdparty/CmBacktrace/addr2line/*

3.4.18. 动态内存

BabyOS 代码中避免使用动态内存，因为 MCU 本身内存是有限的，而且使用动态内存会有碎片的风险。有用户的应用场景可能需要用到动态内存，所以 BabyOS 将 FreeRTOS 的 heap_4 拿过来使用，以 b_mod_heap 的形式提供给用户使用。

首先配置使能 heap，并指定 heap 的大小，同时如果是使用了外部的 SDRAM，那么配置内存的地址。

申请和释放内存的 API：

```
void *bMalloc(uint32_t xWantedSize);
void bFree( void *pv );
```

3.4.19. QPN

QPN 是移植第三方代码 QP_nano 的一套事件驱动的框架，有其自身的一套原理，简单的一页 A4 纸无法描述清楚。需要进一步了解的可以加入开发者群，在群共享里有《嵌入式系统的事件驱动型编程技术》，此份代码是有开发者 Alex 提供，有疑问的用户可以咨询 Alex。

3.4.20. 软定时器

软定时器是基于 tick 完成的，在移植 BabyOS 时需要提供 tick 时钟。那么软定时器便是基于此。这个不是一个严格的定时器，因为它计数满了后不能无法抢占 CPU 去执行定时器的函数

首先定义实例，指定周期和是否重复：

```
bTIMER_INSTANCE(name, _period, _repeat)
```

定义实例后便可以通过使用如下 API 去操作定时器：

```
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);
```

3.4.21. 调节参数

调节参数是通过命令行去设置程序内的变量值。

注册需要调节的参数：

```
#define bPARAM_REG_INSTANCE(param, param_size)
```

通过命令控制，命令介绍如下：

`param` 查看所有已注册的变量

`param [变量名]` 查看变量的值

`param [变量名] [数值]` 修改变量的值

3.4.22. 文件系统

BabyOS 中以及加入了 FatFS 和 LittleFS,通过配置选择使用哪种文件系统，如果基于 SPIFLASH 使用文件系统，则注册设备时设备号为 SPIFLASH，如果基于 SD 卡，则注册设备时设备号为 SD。

调用 bFS_Init()对设备进行挂载。使用 FatFS 情况下，SPIFLASH 的路径为 0：SD 卡的路径为 1:

由于 FatFS 没有擦写均衡，如果是基于 SPIFLASH 使用则选择 LittleFS。

3.5. 中断处理

3.5.1. GPIO 外部中断

首先注册外部中断发生时的回调，指定引脚号和处理函数：

```
bHAL_REG_GPIO_EXTI(_pin, _handler)
```

当外部中断产生时，调用硬件抽象层如下函数：

```
void bHalGPIO_EXTI_IRQHandler(uint8_t pin);
```

3.5.2. 串口接收中断

串口接收提供给用户的时注册空闲回调，指定串口号，接收 buf 大小，空闲判断的时间阈值，空闲的处理函数：

```
bHAL_REG_UART_RX(_uart, _buf_len, _idle_ms, _idle_handler)
```


当串口接收中断产生，将接收的单字节通过如下函数传入：

```
void bHalUartRxIRQ_Handler(uint8_t no, uint8_t dat);
```

4. 使用教程

4.1. 详细教程

代码仓库:https://gitee.com/notrynohigh/BabyOS_Example

不同分支对应不同教程，根据需要选择不同的分支查看。

4.2. 概要描述

https://gitee.com/notrynohigh/BabyOS/wikis

https://github.com/notrynohigh/BabyOS/wiki

5. 期望

一份代码的成长离不开网络的大环境，希望能够在众多网友的支持下，将她不断的扩充不断的完善。让她成为 MCU 裸机开发中不可缺少的一部分。也希望各位开发者一起努力优化她。