

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

目录

BabyOS 设计和使用手册.....	1
修订记录.....	2
目录.....	3
1. 引言.....	5
2. 开发组成员.....	5
3. 设计思路.....	6
3.1. 代码结构.....	7
3.2. 代码框图.....	8
3.2.1. 驱动接口.....	9
3.2.2. 设备接口.....	10
3.2.3. 设备注册.....	11
3.3. 功能模块设计.....	12
3.3.1. 电量检测.....	13
3.3.2. 错误管理.....	13
3.3.3. 事件管理.....	14
3.3.4. MODBUS RTU.....	14
3.3.5. 异步发送管理.....	15
3.3.6. 私有协议.....	16
3.3.7. 数据存储.....	17
3.3.8. UTC 转换.....	18
3.3.9. FIFO.....	18
3.3.10. 阳历阴历.....	18
3.3.11. KV 键值对存储.....	18
3.3.12. Xmodem128 和 Ymodem.....	19
3.3.13. 打印日志 b_log.....	20
3.3.14. 菜单程序.....	20

3.4. 中断处理.....	21
3.4.1. GPIO 外部中断.....	21
3.4.2. 串口接收中断.....	21
4. 使用教程.....	22
4.1. 详细教程.....	22
4.2. 概要描述.....	22
5. 期望.....	23

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

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

蜗牛

向日葵

.

更多请加入开发者群进行查看....

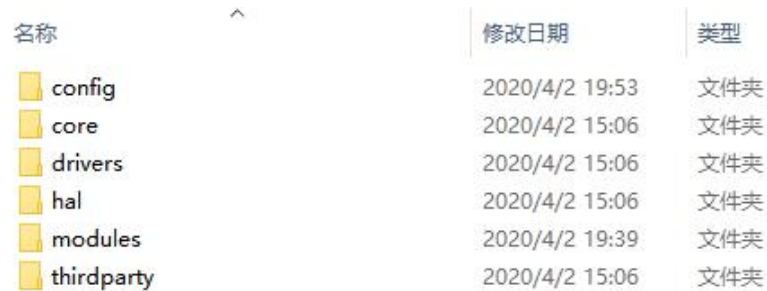
3. 设计思路

BabyOS 的定位在 MCU 裸机开发，做一个带框架的功能及设备库。裸机项目可以直接使用 BabyOS 为整个项目的框架。对于使用操作系统的用户，可以将 BabyOS 作为中间件使用。

一个公司开发的产品之间会有较多相同的功能，例如智能穿戴设备公司，以手环和智能跑鞋为例，除显示和算法不同，其他功能几乎都是可以复用的。可复用的功能以功能模块的形式存在于 BabyOS，新项目开始时可通过搭积木的方式选择已有的功能模块。以这种方式减少重复的工作加快项目开发。

物联网领域使用 MCU 进行裸机开发的产品非常多，物联网其中一个非常重要的特性是低功耗。为方便工程师控制功耗，BabyOS 的驱动操作设计为类似文件的操作，以打开和关闭文件对应设备的唤醒和睡眠。从功能模块的角度考虑，驱动使用了统一的接口也方便了功能模块的设计。

3.1. 代码结构



名称	修改日期	类型
config	2020/4/2 19:53	文件夹
core	2020/4/2 15:06	文件夹
drivers	2020/4/2 15:06	文件夹
hal	2020/4/2 15:06	文件夹
modules	2020/4/2 19:39	文件夹
thirdparty	2020/4/2 15:06	文件夹

图 3-1 BabyOS 代码目录

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

bos/driver/ 选择驱动添加至工程，将 b_hal.h 内定义的硬件接口取消注释

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

bos/hal/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

3.2. 代码框图

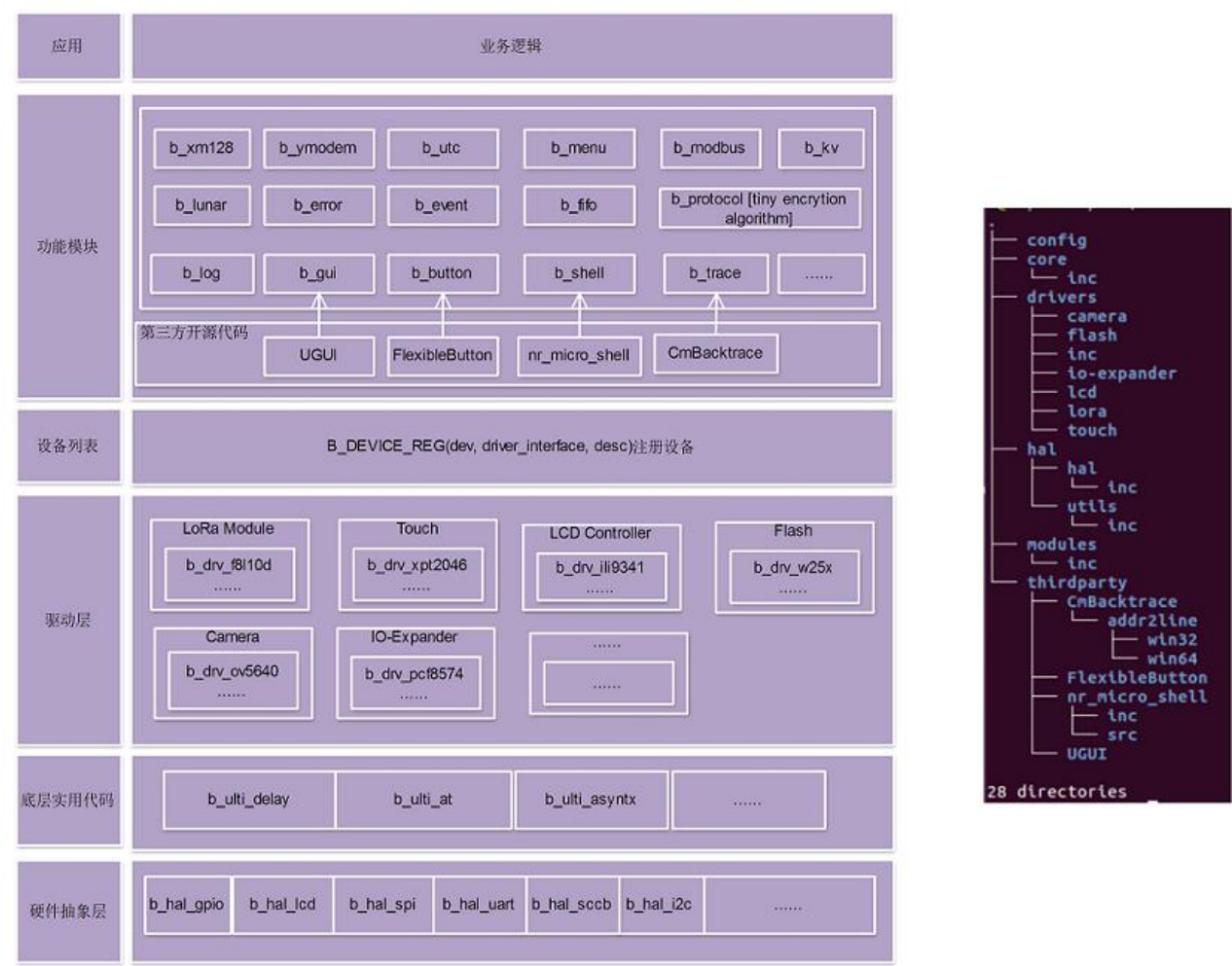


图 3-2 代码框图

3.2.1. 驱动接口

BabyOS 驱动有统一的接口如下：

```
typedef struct bDriverInterface
{
    int (*init)(struct bDriverInterface *);
    int (*open)(void);
    int (*close)(void);
    int (*ctl)(uint8_t cmd, void *param);
    int (*write)(uint32_t addr, uint8_t *pbuf, uint16_t len);
    int (*read)(uint32_t addr, uint8_t *pbuf, uint16_t len);
    void *_private;
}bDriverInterface_t;
```

1) 初始化 init，执行初始化设备的操作，成功返回 0，异常返回-1。如果初始化异常，设备会被标记为异常设备，其他操作都无法执行。

2) 打开/关闭 open/close，打开和关闭用于做设备唤醒和休眠的相关操作。

3) 控制 ctl，执行控制用于配置设备至特定模式等一些特定操作。执行函数需要带上参数 cmd 和 param，当前驱动支持的 cmd 以及 cmd 对应 param 的类型需要在 b_device.h 中定义。

4) 读/写 read/write，读写操作用于和设备进行数据交互。

5) _private 驱动私有部分

3.2.2. 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);
```

3.2.3. 设备注册

在文件 `b_device_list.h` 内通过如下宏进行设备注册：

```
B_DEVICE_REG(W25QXX, bW25X_Driver, "flash")
```

设备号：W25QXX

驱动：bW25X_Driver

描述：flash

`b_device.h` 中通过如下方式统计设备数量：

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

`b_device.c` 中通过如下方式建立驱动数组及描述数组：

```
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"
};
```

设备号就是各个驱动和描述的索引，由此实现设备号和驱动的对应。

3.3. 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

当功能模块需要使用硬件资源时，提供 API 给用户，让其指定设备号。用户指定的设备号是 3.2.3 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。

当功能模块有需要循环执行的操作时，例如检测超时等，将这些操作放入 `bExec()` 函数内执行。

使用功能模块是先将 `b_config.h` 内对应的宏打开。

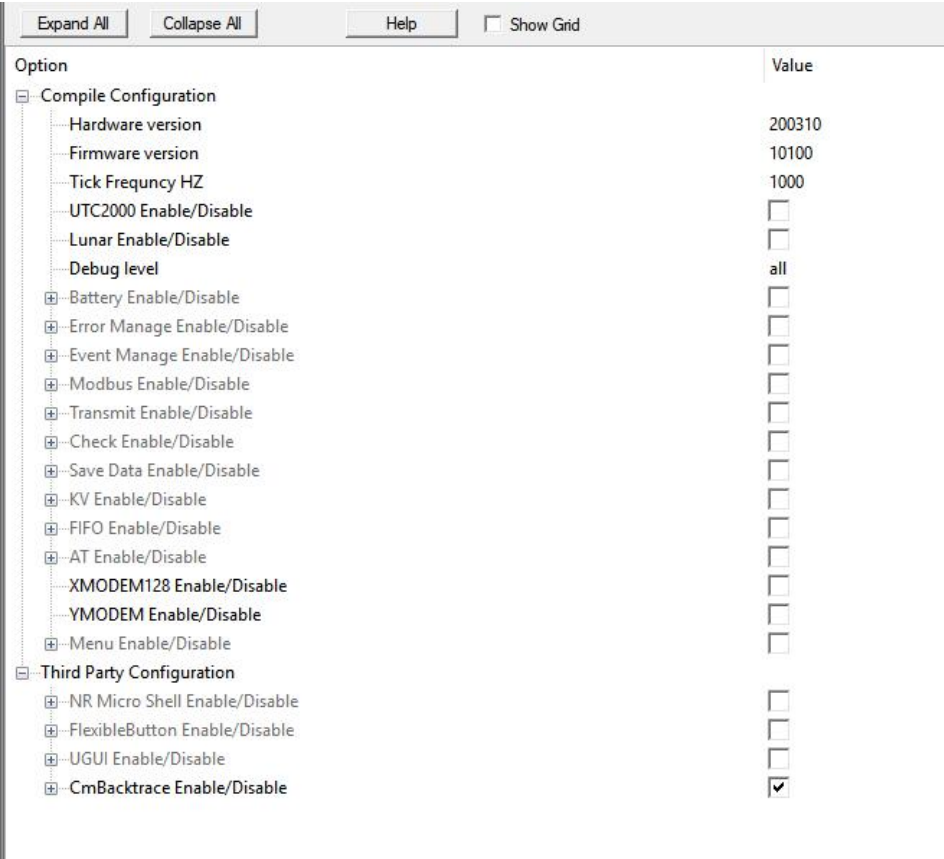


图 4-3 配置页面

3.3.1. 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 转换的函数是 weak 函数，用户根据自身的平台重新实现这个函数。

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。

3.3.2. 错误管理

使用错误管理时，首先指定错误发生后的回调函数，因此设计初始化函数如下 *int bErrorInit(pecb cb);*

系统遇到错误后主动将错误提交到错误管理，有些错误是持续性的，防止频繁的提交错误，所以再提交错误的 API 带上最小间隔时间，在间隔时间内相同的错误无法重复提交。

错误分了两个等级，等级 0 的错误只执行 1 次回调。等级 1 的错误在执行回调后如果没有主动清除错误，那么最小间隔时间过后还会继续调用回调。

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

清除错误的 API:

int bErrorClear(uint8_t e_no);

3.3.3. 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。

通过如下函数注册事件，参数 number 不能重复：

```
int bEventRegist(uint8_t number, pEventHandler_t phandler);
```

特定事件发生后调用如下函数：

```
int bEventTrigger(uint8_t number);
```

3.3.4. MODBUS RTU

3.3.5. 异步发送管理

数据输送给某个通信模块后需要等待发送完成的信号，数据输送完成和真正的发送完成中间有个间隔事件。这样的场景在这里称为异步发送。这个功能模块主要是确保在等待的这段时间没有新的数据进行发送。同时还需要负责等待发送完成信号的超时。

首先需要指定发送数据的函数及超时时间，注册完发送管理实例后会得到实例号，后续操作都是根据这个实例号进行。。

```
int bAsyntxRegist(pSendBytes f, uint32_t timeout_ms);
```

当有数据需要发送时调用发送请求，如果是紧急数据可以将发送等级置为1，那么可以不用管是否在等待完成信号，直接开始一次新的发送。

```
int bAsyntxRequest(int no, uint8_t *pbuf, uint16_t size, uint8_t flag);
```

当数据真正发送完成后调用如下函数：

```
int bAsyntxCplCallback(int no);
```

3.3.6. 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 bProtocolRegist(bProtoID_t id, pdispatch f);  
  
int bProtocolParse(int no, 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(int no, uint8_t cmd, uint8_t *param, bProtoLen_t  
param_size, uint8_t *pbuf);
```


3.3.7. 数据存储

数据存储功能模块，暂时提供了 3 种场景。

第一种场景：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。

第二中场景：在一块最小擦除单位连续存储相同大小的数据，同时支持统计已存数据的条数。

第三种场景：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。

第三种场景有较多参数需要给定，因此在注册实例时需要传入一个结构体数据：

```
typedef struct
{
    uint8_t min_unit;
    uint8_t min_number;
    uint32_t min_size;
    uint8_t total_unit;
    uint8_t total_number;

    uint32_t fbase_address;
    uint32_t fsize;
    uint32_t ferase_size;
}bSDA_Struct_t;

int bSDA_Regist(bSDA_Struct_t , uint8_t dev_no);
```

3.3.8. 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

3.3.9. FIFO

提供了一组 API 按照 FIFO 的特性去操作一块内存。由于代码里面不涉及动态内存的分配及释放，因此在注册 FIFO 实例时需要给定一个数组。

3.3.10. 阳历阴历

提供 API 给用户使用

3.3.11. KV 键值对存储

这个功能模块时基于 SPI Flash 进行，需要用到 4 个擦除最小单位。实际能用到的只有 1 个最小擦除单位。所以不支持大数据的存储，适用于存储系统参数。

在初始化时需要给定起始地址及给 KV 使用的存储区域大小，这个参数不得小于 $4 * e_size$ ，其中 e_size 是最小擦除大小。

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

3.3.12. Xmodem128 和 Ymodem

Xmodem 和 Ymodem 完成了接收部分，采用同样的套路，注册实例时指定发送字节的函数及收到数据后的回调函数。

```
int bXmodem128Init(pcb_t fcb, psend fs);  
int bXmodem128Parse(uint8_t *pbuf, uint8_t 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);
```

每成功收到一帧数据就调用回调，当回调收到的数据指针为空表示结束。

Ymodem 可以传送文件名，因此回调和 Xmodem 会不同。

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

这两个回调 number 都是从 0 开始依次增加。

3.3.13. 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。

3.3.14. 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的事了，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. 中断处理

3.4.1. GPIO 外部中断

首先注册外部中断发生时的回调：

```
int bHalGPIO_EXTI_Regist(bHalGPIO_EXTI_t *pexti);
```

当外部中断产生时，调用：

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

注册进来的 *pexti* 会放入单向链表内，*bHalGPIO_EXTI_IRQHandler* 会比较 *pin* 和 *pexti->pin*，如果相同则调用 *pexti->cb*。

3.4.2. 串口接收中断

串口接收提供给用户的时注册空闲回调：

```
int bHalUartRxRegist(bHalUartRxInfo_t *puart_rx);
```

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);
```

注册进来的 *puart_rx* 会放到单向链表内，注册回调时会指定阈值（已接收数量无变化的时间，用于判断空闲），当判断空闲后调用 *puart_rx->cb*。

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。