BabyOS 设计和使用手册

源码地址

https://github.com/notrynohigh/BabyOS

https://gitee.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

目录

1. 引言

BabyOS 是为 MCU 裸机项目而生，分为驱动和功能模块两个部分。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。

希望在日后的某一天它能成为工程师喜爱的一份代码。

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

.

3. 设计思路

BabyOS 的定位在 MCU 裸机开发，而复杂的项目可以选择经过大量验证的操作系统 FreeRTOS ,uCOS-II/III 等。

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

图 3-1 BabyOS 代码目录

Core:核心以及功能模块

Drivers:驱动代码

Hal:硬件相关代码

移植过程：

1) Core 目录代码全部添加至工程，通过 b_config.h 文件配置功能模块
2) 选择 Drivers 目录下需要的驱动代码添加至工程
3) b_device_list.h 内注册设备
4) 将硬件相关的代码编写在 Hal 目录的 b_hal.c b_hal.h 中
5) 配置_TICK_FRQ_HZ 值，并根据对应频率调用 bHalIncSysTick

3.2. 设备和驱动

3.2.1. 驱动接口

BabyOS 驱动有统一的接口如下：

```
typedef struct
{
    int (*init)(void);
    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);
}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。

3.2.2. 设备接口

初始化阶段执行 *bInit*，*bInit* 会依次执行在 b_device_list.h 注册的所有设备驱动的 init。执行失败（返回值-1）则标记设备为异常设备。

执行 *bOpen bClose bCtl bWrite bRead* 操作，最终执行其驱动的 open close ctl write read。其中 *bCtl* 需要的参数 cmd 和 param 类型在 b_device.h 内定义。

执行 *bOpen* 后会得到一个句柄（≥0），默认限制是同时打开的设备为 10 个，在 b_core.h 中修改 BCORE_FD_MAX 值进行调整。执行 *bClose* 后句柄会被回收。当某个设备处于被打开的状态则不能再次被打开，执行 *bOpen* 后必须判断句柄是否有效。

每个句柄都对应一个 bCoreFd_t 类型的结构体，其中 uint32_t lseek 成员会在执行读写操作时自加，调用驱动的读写函数时这个值会当做 addr 参数传入，改变这个值是通过 *bLseek* 完成。

```
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);
```

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。

4. 使用方法

4.1. 功能模块

使用功能模块是先将 b_config.h 内对应的宏打开。

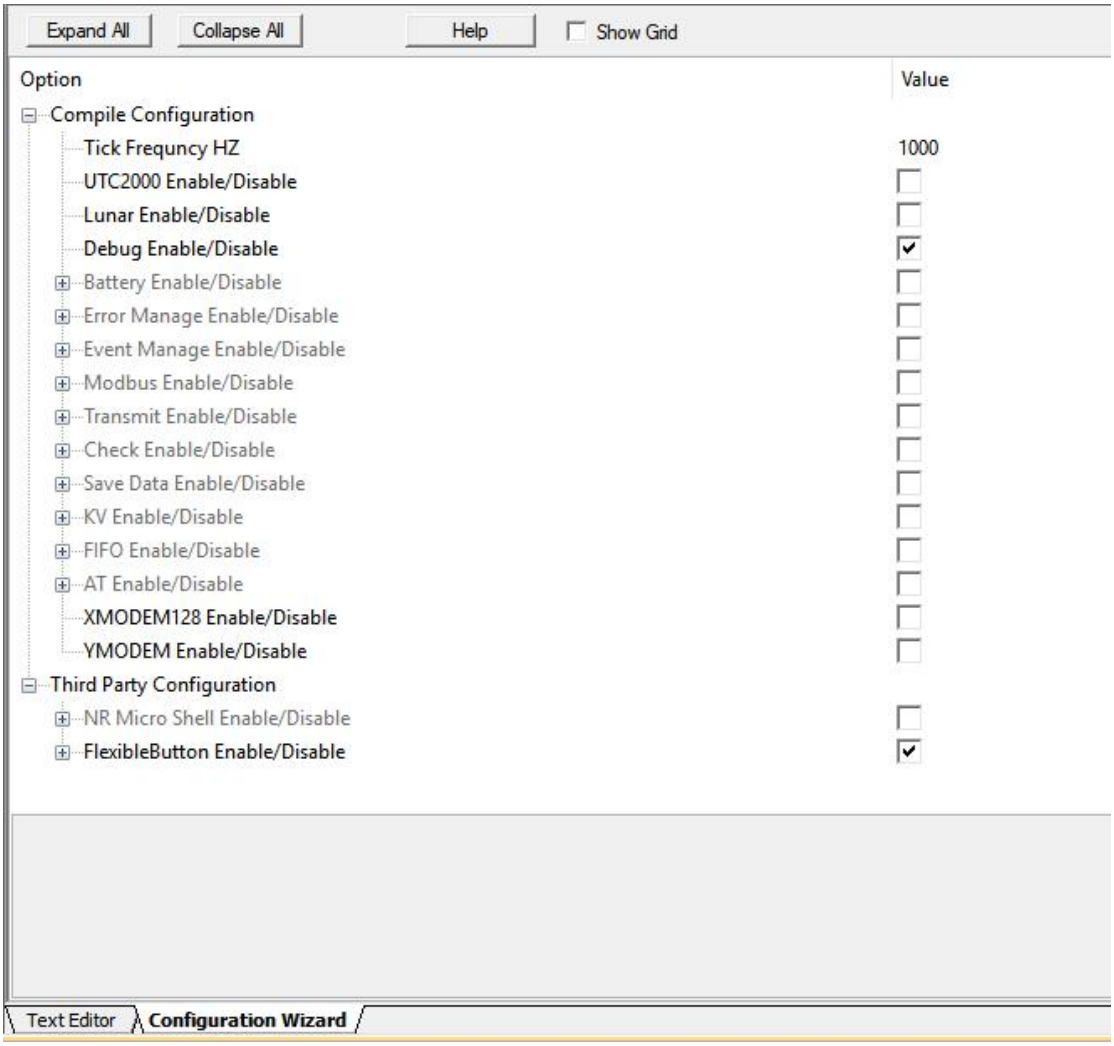

图 4-1 配置页面

4.1.1. 电量检测

电量检测需要用户完成 ADC 采样的函数：

*int bBatteryGetmV(uint16_t *pmV)*

用户在配置文件中会配置电量的阈值以及检测周期，那么 *bExec()*会根据用户配置的周期对电池电量进行检测并判断当前电量状态。

用户通过调用

bBatGetStatus() 获取状态

bBatGetVoltageValue() 获取电压值。

4.1.2. 校验 CRC/累加和/异或

这三种都是属于用户主动调用 API 的功能模块。

*uint32_t bCRC32(uint32_t crc32_in, uint8_t *pbuf, uint32_t len)*
*uint8_t bSUM(uint8_t *pbuf, uint8_t len)*
*uint8_t bXOR(uint8_t *pbuf, uint8_t len)*

4.1.3. 错误管理

错误管理分为两种错误级别 L1 L2，*bExec()*会每秒检测一次是否有错误发生，如果有则调用处理函数。

首先用户需要先给定一个处理错误的回调函数

int bErrorInit(pecb cb)

当有错误发生时调用

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

参数分别为：

错误号

当前时间

间隔时间（间隔时间表示在此间隔时间内，忽略相同错误）

错误等级

L1 等级是调用一次处理函数后就自动失效

L2 等级是调用处理函数后不会失效，超过间隔时间会再次调用执行函数，直到 *int bErrorClear(uint8_t e_no)*主动清除错误。

4.1.4. 事件管理

首先需要注册事件

int bEventRegist(uint8_t number, pEventHandler_t phandler)

其中 number 数值不能重复，第二个参数是事件发生后的处理函数。

当某个事件的触发条件发生，执行

int bEventTrigger(uint8_t number)

4.1.5. MODBUS RTU

这个功能模块目前只做到了主设备的发送及数据的解析。

首先提供设备号 *int bMB_Regist(uint8_t dev_no)*

数据的发送，参数：功能模块实例 ID，功能码，寄存器，数量

int bMB_WriteCmd(int no, uint8_t addr, uint8_t func, uint16_t reg, uint16_t num)

接收到从设备的数据通过 *int bMB_CheckRTUS_ACK(uint8_t *psrc, uint16_t len)*判断校验是否正确。

4.1.6. 发送管理

发送管理是保证当前系统中对外发送数据的执行函数只有一个。

首先注册一个发送管理的实例：

*int bTX_Regist(uint8_t *pbuf, uint32_t size, uint8_t td_mode, uint8_t dev_no)*

参数分别是：

暂存空间（用于存储待发送数据）

暂存空间大小

发送模式（执行完即发送完，执行完需等待发送完成的信号）

设备号

需要发送数据时调用

*bTX_Request(int no, uint8_t *pbuf, uint16_t size, uint8_t flag)*，

参数分别是：

b_tx 实例

数据指针

大小

标志

如果发送模式是属于需要等待完成信号，则在发送完成后调用

int bTX_CplCallback(int no)。

4.1.7. 私有协议

私有协议格式：

头（1Byte）|设备 ID(4Bytes)|数据长度(1Byte)|命令（1Byte）|参数（0~nBytes）|校验和(1Byte)|

首先注册一个功能模块实例：

bProtocolRegist(uint32_t id, uint8_t tx_no, pdispatch f)

参数：

本机 ID

b_tx 功能模块实例 ID

分发函数

使用协议通讯时，发送的操作统一由 b_tx 进行管理。bProtocolParse()

接收到数据后调用

*int bProtocolParse()(int no, uint8_t *pbuf, uint8_t len)*

匹配 ID、长度检测、数据校验通过后调用分发函数。

发送数据是调用

*int bProtocolSend (int no, uint8_t cmd, uint8_t *param, uint8_t param_size)*

参数分别为

私有协议实例 ID

指令

参数

参数的长度。

4.1.8. 数据存储

数据存储基于 SPIFlash 类型存储器。有 A B C 三种存储场景。

A 场景是间隔固定的时间存储固定大小的数据，例如每 1 小时存储 30 个字节。

B 场景是固定地址存储固定大小的数据

C 场景是连续存储固定大小的数据

三种存储场景的使用方式相同，先注册一个实例，再调用存储和读取的 API。

A 场景存储需要提供一下信息，例如每小时存储 1 次，总共存储 1 年的数据

```
typedef struct
{
    uint8_t min_unit;    //最小单位，小时
    uint8_t min_number;  // 1 表示 1 小时
    uint32_t min_size;   //每次存储的数据大小
    uint8_t total_unit;  //最大的单位，年
    uint8_t total_number; //1 表示 1 年

    uint32_t fbase_address; //起始地址
    uint32_t fsize;         //分配的存储大小
    uint32_t ferase_size;   //最小擦除单位的大小
}bSDA_Struct_t;
```

4.1.9. UTC 转换

UTC 转换是提供 API 给用户调用。

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

4.1.10. FIFO

FIFO 是提供 API 给用户调用

*int bFIFO_Regist(uint8_t *pbuf, uint16_t size);*
*int bFIFO_Length(int no, uint16_t *plen);*
int bFIFO_Flush(int no);
*int bFIFO_Write(int no, uint8_t *pbuf, uint16_t size);*
*int bFIFO_Read(int no, uint8_t *pbuf, uint16_t size);*

4.1.11. AT

AT 功能模块是为了方便发送 AT 指令。首先指定发送接口

int bAT_Regist(pAT_TX ptx)

发送 AT 指令

int bAT_Write(int no, bAT_ExpectedResp_t *pe_resp, const char *pcmd, ...)

参数分别是

实例 ID

指针，准备接收回复数据（可以为空表示不关心回复数据）

发送类容

可变参数。

4.1.12. 阳历阴历

提供 API 给用户调用。

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

4.1.13. KV 键值对存储

基于 SPIFlash 的键值对存储，首先划分一块空间给功能模块使用。

*int bKV_Init(int dev_no, uint32_t s_addr, uint32_t size, uint32_t e_size);*参数分别为存储设备 ID，起始地址，大小，最小擦除单位。分配的大小应该大于等于 4 倍的最小擦除单位。

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

4.1.14. Xmodem128 和 Ymodem

两种协议完成了接收机部分，使用方式一样。

首先初始化给定发送单个字节的函数以及接收完一帧数据的回调函数。*bXmodem128Init(pcb_t fcb, psend fs)*。

调用 *bXmodem128Start()*开始文件的传输，用户将当接收到的数据给到*bXmodem128Parse(uint8_t *pbuf, uint8_t len)*进行解析，解析一帧后调用回调。

Ymodem 的使用方式和 Xmodem128 一样。

4.1.15. Shell

b_shell 基于第三方代码 nr_micro_shell

启动 shell

```
int bShellStart()
```

接收到的数据提交给 shell 进行解析处理

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

Shell 指令和指令的处理在 b_shell.c 中增加。

```
const static_cmd_st static_cmd[] =
{
    {"ls", _bShellLSCmd},
    {"test", _bShellTestCmd},
    {"\0", NULL}
};
```

4.1.16. FlexibleButton

b_button 是基于 FlexibleButton。

在 b_button.h 中填写 button id。

在 b_button.c 中填写每个 button 被按下时的裸机电平。

下面是两个 weak 函数：

```
void  bButtonCallback (void *p)
uint8_t  bButtonRead (void *p)
```

将回调以及读取按钮逻辑电平的函数重新实现即可。

4.2. 驱动

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。