

DA14580 学习汇总

简介

芯片名称:DA14580

内核:Cortex-M0 32-bit

系统时钟:16MHZ 睡眠时钟:32K (所以要外挂两个晶振)

协议栈:不开源,采用 Riviera Waves 授权协议栈 IP

Ram:42 kB System SRAM (存放运行数据) 8 kB RetentionSRAM (低漏电存储器,暂存休眠状态下的运行数据)

存储方式:32k 的 OTP(一次性烧录),要实现反复烧录则需要外挂一个 flash 或者 EEPROM 84 kBROM (存放协议栈)

最小系统只需 7 个元件

支持仿真

烧录方式:串口烧录(JTAG 也可以烧录,烧录到外挂的芯片中)

封装: 34 pins,40pins, 48 pins

功耗: 首款突破 4mA 无线收发电流极限的蓝牙智能解决方案(小米手环可满足 30 天续航)

术语

Profile: 配置文件(在 GATT 的基础上进行数据的本地处理)

GATT: Generic Attribute Profile 通用的配置文件(负责基础的数据通信)

DISS:设备信息服务(显示设备的制造商信息)

UUID:全球唯一识别码,如 0x2A45 位设备序列号的 UUID(任意蓝牙都可以通过他获取到设备序列号)。

AES:Advanced Encryption Standard 是 DA14580 中内置的 128 位加密处理器

development_guide: 开发手册

GAP:Generic Access Profile 通用接口配置。跟蓝牙的 advertising 相关

GTL: Generic Transport Layer 通用传输层。当工作于外部主控模式时,用来传输主控到 DA14580 的数据

NVDS: Non-Volatile Data Storage 非易失性数据存储器

OTP: One Time Programmable (memory) 单次可编程存储器

PHY:physical layer 物理层

LL: Link Layer 链路层

外挂 EEPROM

DA14580 的芯片是没有 flash 空间的(其实有个 32kb 的 OTP,但只能烧写一次),也可以使用烧录到内存,但是掉电过后就没有程序了。所以开发过程中一般使用以下几个模式:

- 1.Debug 模式，即通过 jlink，又或者其它工具使用 swd 接口（vcc，gnd，swclk，swdio），通过 KeilMDK 将程序写进 RAM 中，直接调试。（缺点就是断电后数据就丢失）。
- 2.外挂 E2Prom，此种方法相当于将 E2 当成 DA14580 的 Flash 空间，源码写在 E2 里面，DA14580 芯片上电之后将 E2 的数据复制到 RAM 中，运行。（缺点就是烧写比较麻烦~但是当用到睡眠模式时，必须使用它来进行软件功能验证，因为有可能会出问题的！所以必须验证）。
- 3.OTP 模式~此种模式，只能烧一次~最终产品才烧~通过 SmartSnippets 工具下载代码到 OTP 一般前期阶段使用的是 Debug 模式，后期需要用到睡眠等等其他 Debug 不能调试的情况下采用外挂 E2Prom 模式。

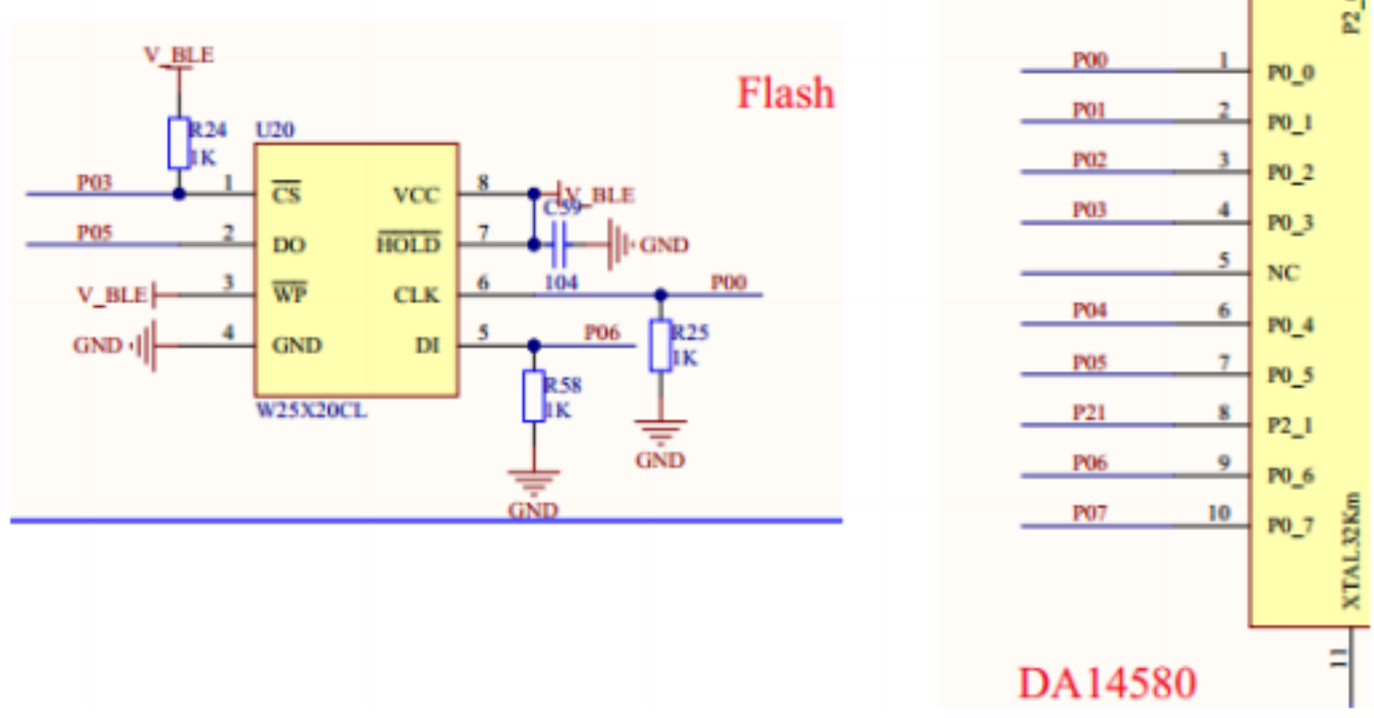
启动顺序

DA14580 P0 口与其他普通 I/O 口不同之处在于 P0 口除了能当普通 I/O 用之外还具备上电 booting 的功能。即当 DA14580 上电后，芯片内固化的程序会在 P0 口上按 Table 1 的时序进行扫描，当扫描到相应的设备且发现该设备具备启动程序的话便会从该设备启动，不再往下扫描，否则会继续循环扫描，多次扫描不到时才停止扫描。

Table 1: Pin assignment and booting sequence from external devices

Pin	Step	SPI Master	Step	SPI Master	Step	UART	Step	SPI Slave	Step	I2C
P0_0	1	SCK	2	SCK	3	TX	7	SCK	8	SCL
P0_1				CS		RX				SDA
P0_2				MISO	4	TX			9	SCL
P0_3		CS		MOSI		RX		CS		SDA
P0_4					5	TX			10	SCL
P0_5		MOSI				RX		MISO		SDA
P0_6		MISO			6	TX		MOSI	11	SCL
P0_7						RX				SDA

上电第一步就看可以从SPI中启动不，如果外挂了Flash，则为SPI通信，接在对应的IO口上，上电过后则从SPI启动



DA14580 的开发者关注的问题

对于蓝牙单芯片应用开发来说，我们要关注的问题是：蓝牙协议栈方面如何新增一个 GATT profile（服务和特征值定义及操作）、SOC 内核方面如何驱动外围设备、系统应用框架上如何使用定时器和任务间消息通信等等。DA14580 单芯片发布时并不是一颗裸片，而是带有开发平台和 SDK 包，还有常用的应用例程（如防丢 proximity），我们要做的就是通过 SDK 和相关的文档去理解它整个系统架构和应用框架，在这个基础上才能去完成以上三个方面的开发。

SDK 目录结构

DA14580 的 SDK 开发平台使用 keil，我们先来看看开发例程的目录结构，再来看 SDK 目录结构。前者简单一些，后者因为涉及到第三方 IP、ROM 等原因，目录实在是太多太细了，初接手真的会歇菜。

防丢（proximity，英文是接近的意思）的开发目录结构如下：



这里需要注意的是，ROM 里面的固化代码，包括协议栈和单任务操作系统的相关管理代码也是整个工程应用的一部分，只不过没有列到开发目录里面。

SDK 目录架构如下：



Profile（BLE 的 GATT 服务）

Profile 部分或许是为了更加便利蓝牙应用推出的。

一、蓝牙Profile有哪些

典型的蓝牙Profile有：

- A2DP (Advanced Audio Distribution Profile)，高级音频传输
- AVRCP (Audio/Video Remote Control Profile)，音视频远程控制
- BPP (Basic Printing Profile)，基本打印
- BIP (Basic Imaging Profile)，基本图像
- DUN (Dial-up Networking)，拨号网络
- FTP (File Transfer Profile)，文件传输
- HIDH (Human Interface Device Host)，人机界面
- HFP (Hands-free Profile)，免提
- MAPS (Message Access Profile (Server)，信息存取
- OPP (Object Push Profile)，对象交换
- PAN (Personal Area Network)，个人局域网
- PBAP (Phone Book Access Profile)，电话本存取
- PRXM (Proximity Monitor)，
- PRXR (Proximity Reporter)，
- SIMAP (SIM Access Profile)，SIM卡存取
- SPP (Serial Port Profile)，串口

注意：在 DA14580 中，只有上述的一部分，比如银屏传输，文件传输是不可以使用的。

从工程的代码目录结构来看，每个 profile 都有一个以 profile（如 proxr）命名的.c 文件，也有一个以 profile_task（如 proxr_task）命名的.c 文件；相应地，每个应用子任务也有一个 app_profile（如 app_proxr）的.c 文件，和 app_profile_task（如 app_proxr_task）的.c 文件。一般地：

在操作系统 ke 内核看来，Profile 和 profile_task 共同完成一个 task 任务，其中 app_proxr_task 的 task ID 标识是 TASK_PROXR。但 app_profile 和 app_profile_task 并不是一个具体的 task 任务，在代码目录的 app 目录，所有的 task，包括 app_proxr_task 和 app_batt_task(电池)、app_sec_task(安全)共同组成一个 task，在 app.c 中完成任务创建，task 的 ID 标识是 TASK_APP。各个 app_profile_task 只不过完成应用的一个子场景功能，如防丢、电池告警等。

app 是主动发送消息给 profile，以执行相应的蓝牙 GATT 服务和操作，并接受回调。即 app 是 profile 的上层。

Profile 任务执行 GATT 服务/属性的具体创建 create、开启服务 enable 和属性特征的读写等操作，其调用 ATT 和 GAP 等底层接口来实现具体功能。Profile 作为接口供给 app 层调用，app 是通过消息通信来完成接口调用的。

app_profile 的代码一般包括主动调用的接口实现，而 app_profile_task 则是接受消息回调的接口实现。两者的分工是非常清晰的。

应用开发框架

DA14580 的应用开发框架的核心是基于状态机和消息回调。以下分析以防丢 proxr 为例。

1. 状态机

每个任务都必须明确自己的状态表，例如 proxr 的状态表是：

```
/// Specifies the message handler structure for every input state.
const struct ke_state_handler proxr_state_handler[PROXR_STATE_MAX] =
{
    [PROXR_DISABLED] = KE_STATE_HANDLER(proxr_disabled),
    [PROXR_IDLE] = KE_STATE_HANDLER(proxr_idle),
    [PROXR_CONNECTED] = KE_STATE_HANDLER(proxr_connected),
};
```

状态表

状态响应接口集

状态的初始化和转换是由用户主动切换的。在某个确定的状态时，内核会在对应的状态响应接口集中遍历所有发给该任务的消息。

每个任务都会在初始化时被创建，例如 proxr 任务的创建是：

```
void proxr_init(void)
{
    // Reset Environment
    memset(&proxr_env, 0, sizeof(proxr_env));

    // Create PROXR task
    ke_task_create(TASK_PROXR, &TASK_DESC_PROXR);

    ke_state_set(TASK_PROXR, PROXR_DISABLED);
}
```

这时，假设有个其他的任务发一个消息给 TASK_PROXR，则会在 proxr_disabled 中查找相应的消息回调接口，并执行回调。

2. 消息回调

接下来看看各个状态的状态响应接口集，例如 PROXR_CONNECTED 连接状态时的状态响应接口集如下。可见，其会对两个消息进行回调，一个是底层 ATT 收到对特征值的写操作时执行回调，另一个应用层主动改写另一个特征值。在笔者的防丢和计步应用中，前者是实现防丢告警功能，后者是上报计步数据。

```
/// Connected State handler definition.
const struct ke_msg_handler proxr_connected[] =
{
    {GATTC_WRITE_CMD_IND, (ke_msg_func_t) gattc_write_cmd_ind_handler},
    {PROXR_JIBU_UPDATE_REQ, (ke_msg_func_t) proxr_jibu_update_req_handler},
};
```

3. 任务间通信

消息发出之后，系统即会执行 proxr_jibu_update_req_handler 回调。

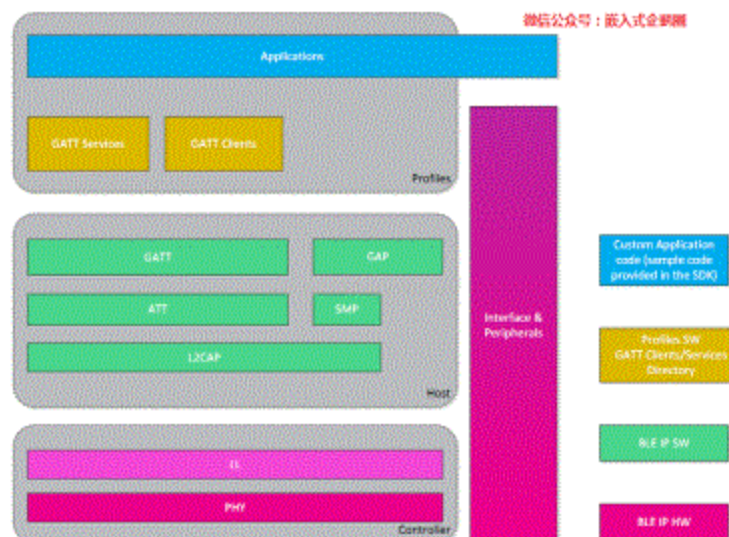
```
struct proxr_jibu_update_req * req = KE_MSG_ALLOC(PROXR_JIBU_UPDATE_REQ, TASK_PROXR, TASK_APP, proxr_jibu_update_req);

req->conhdl = app_env.conhdl;
req->jibu_count = value;

ke_msg_send(req);
```

另外，笔者会根据文章的阅读量考虑进一步对 DA14580 的 SDK 进行分析，如系统启动过程、服务建立过程以及上面说的，如何进行应用开发，即蓝牙协议栈方面如何新增一个 GATT profile（服务和特征值定义及操作）、SOC 内核方面如何驱动外围设备、系统应用框架上如何使用定时器和任务间消息通信等等。

软件层次架构



BLE 协议栈可以大致分为应用层、profile 服务层、BLE Host 层（软件实现）、BLE controller 层（硬件实现，属于基带部分）

BLE host 实现数据适配 L2CAP、链路管理 GAP、基础属性协议 ATT，GATT 是基于 ATT 进行封装并向上层提供接口服务，以让用户更方便地使用 ATT 来进行数据交互。他们都分别对应 RW 内核的一个或者多个 task，例如 GAP 包括 GAP 管理和 GAP 控制两个 task。

Profile 层是基于 GATT 来向应用层提供数据通信服务的，每个 profile 都负责自己的专有的服务功能。例如，有电池服务、设备信息服务和自定义的一些 profile 等等。

GATT 是负责基础的数据通信，而 profile 则是在 GATT 的基础上进行数据的本地处理。例如，GATT 收到对方的写请求时会通知 profile，由 profile 来决定怎么处理接收到的数据，可以写到属性字段中，也可以不写，profile 也可以进一步给 task_app 发送通知。

各个 profile 是独立并且平等的，因为各个 GATT 都是平等服务的。每个 profile 都对应 RW 内核的一个 task。

应用层是基于多个 profile 来实现自己的需求。由于 RW 内核实质是一个单任务内核，所以应用层是一个特别的 task。其和所有的 profile 打交道，其控制所有 profile，并处理各个 profile 的消息回调。

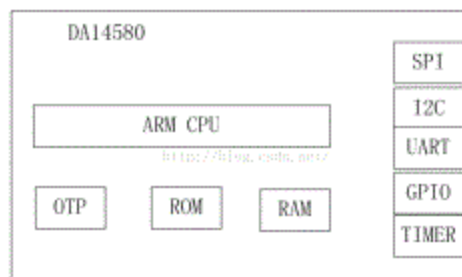
1.2 task 层次

根据上面的分析，我们可以得到以下 task 层次图：



蓝牙 BLE

DA14580 集成的是第三方公司 RW 的蓝牙协议栈 IP，范围包括 GAT 和 GAP 层及以下。因此我们可以在代码框架目录上看到 RW 开头命名的目录和头文件，官方文档涉及到蓝牙协议栈方面大部分都是 RW 公司出品。



蓝牙通信分为了两个部分：底层协议栈和上层 APP（我们只能见到上层 app），协议给上层发消息，会有对应的消息处理函数。上层向底层发送消息，底层协议栈执行后，会有信息反馈回来。

低功耗

四种工作模式：active、idle、extended sleep、deep sleep；

active 与 idle 模式，不做详细表述。

extended sleep 浅度休眠，休眠之后，除 ROM 外，仅仅有 RAM 处于供电状态以保存数据，其他模块部分会掉电。

deep sleep 仅保留 ROM 供电。

DA14580 是低功耗蓝牙模块，因此每隔一段时间就会进入休眠。然后开定时器之后会再次唤醒。由于进入休眠再唤醒

之后，一些模块的状态会重新恢复为初始化状态，所以需要特别留意。

编程-GPIO

DA14580 I/O 口使用比较灵活，除了几个特殊口之外大部分皆可实现以下功能：

Table 226: P00_MODE_REG (0x50003006)

Bit	Mode	Symbol	Description	Reset
4:0	R/W	PID	Function of port 0 = Port function, PUPD as set above 1 = UART1_RX 2 = UART1_TX 3 = UART2_RX 4 = UART2_TX 5 = SPI_DI 6 = SPI_DO 7 = SPI_CLK 8 = SPI_EN 9 = I2C_SCL 10 = I2C_SDA 11 = UART1_IRDA_RX 12 = UART1_IRDA_TX 13 = UART2_IRDA_RX 14 = UART2_IRDA_TX 15 = ADC (only for P0[3:0]) 16 = PWM0 17 = PWM1 18 = BLE_DIAG (only for P0[7:0]) 19 = UART1_CTSN 20 = UART1_RTSN 21 = UART2_CTSN 22 = UART2_RTSN 23 = PWM2 24 = PWM3 25 = PWM4 Note: when a certain input function (like SPI_DI) is selected on more than 1 port pin, the port with the lowest index has the highest priority and P0 has higher priority than P1.	0x0

Port0 有 8 个引脚，Port1 有 6 个引脚(其中包括 DEBUG 引脚 SW_CLK 与 SWDIO)，Port2 有 10 个引脚，Port3 有 8 个引脚；

每个引脚都可以选择上拉或者下拉 25KOhm 的电阻；

每个引脚上拉电压在 VBAT3V(降压模式)与 VBAT1V(升压模式)两者可选；

4 路模数转换的引脚固定分配为 Port0 中的 0:3 引脚；

当系统进入睡眠模式时，引脚保持最后的状态。

开发工具

手机上安装一个 Light blue (苹果，8.0 以上系统支持) 的 APP，电脑安装 KEIL 的安装, SmartSnippets 的安装。

引脚图

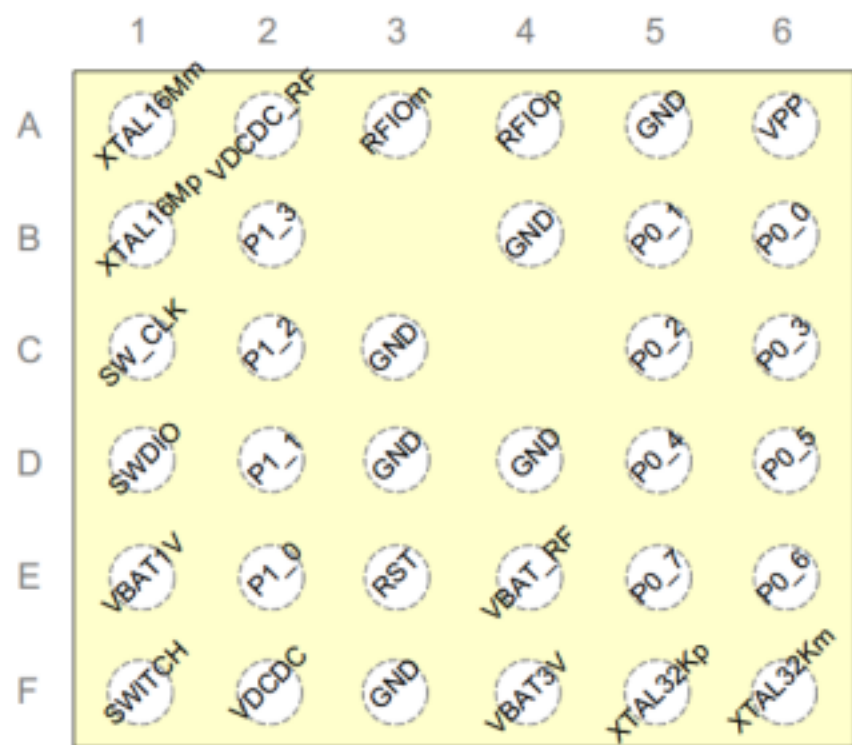
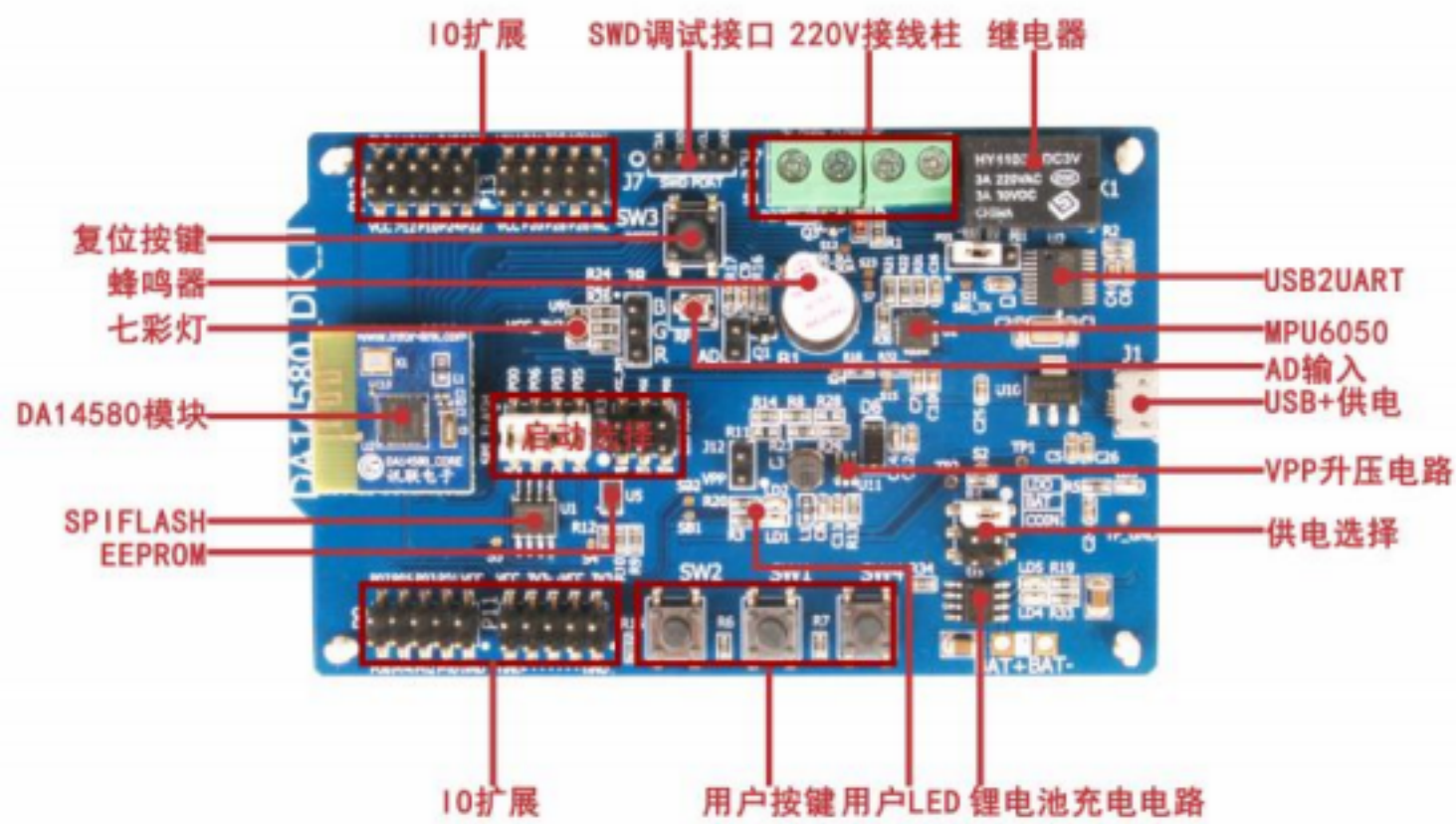


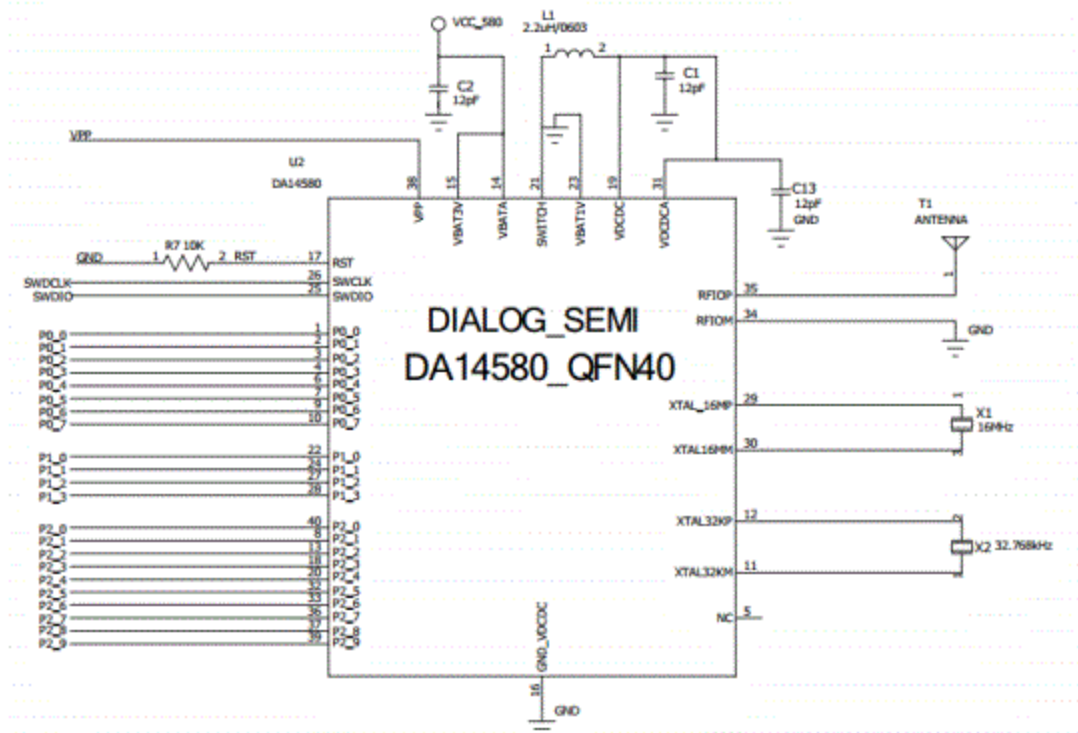
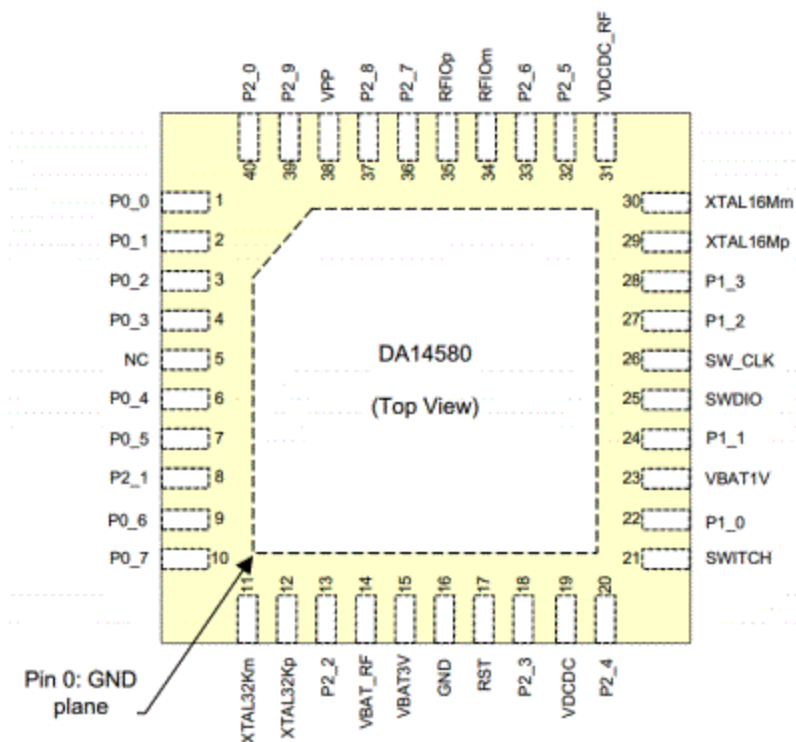
Figure 2: WLCSP34 ball assignment

迅捷 DA14580 学习

开发板接口图



引脚图



下载到内存

方法一：SmartSnippets 下载到 ram

- ① 打开 SmartSnippets
- ② 新建工程 new
- ③ 选择串口下载模式，芯片选择 DA14580-01
- ④ OPEN
- ⑤ 设置里面选择 04/05 引脚作为串口
- ⑥ 点击 booter 选项
- ⑦ Browse 浏览到 hex 文件
- ⑧ Download, 复位。
- ⑨ 下载到内存，断电后丢失数据。

方法二：Keil 下载到 Ram 中

- ① 需要 JTAG 接 SW 引脚
- ② 点击 Debug 则会下载到 RAM 中

下载到 Flash（参考实战教程从外部 SPI Flash 启动）

- ① 连接 JLINK
- ② 连接 SPI 的 4 个跳线帽，去掉 EEPROM 的跳线帽、



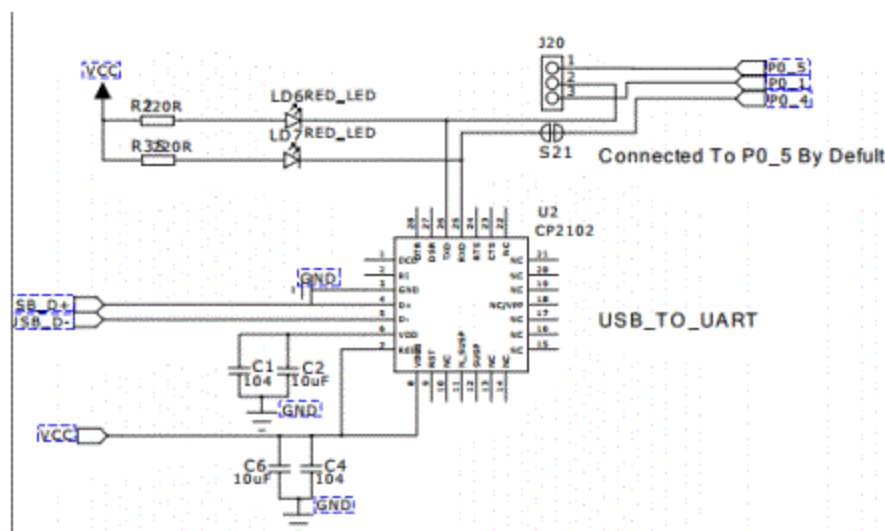
- ③ 连接如下：
- ④ 打开 Smartsnippet 软件
- ⑤ 选择 flash 下载
- ⑥ 点击 connect, 此时会把 bootloader 下载到 RAM 中，会显示烧录成功
- ⑦ 点击 Erase, 成功会把所有地址里面的内容全部写为 1
- ⑧ Browse 下载 hex 文件
- ⑨ Burn 进行烧录
- ⑩ 点击 YES
- 11 然后就可以从 FLASH 中读取内容回来，看是否和烧录的内容一致

USB 转串口

USB 转串口采用的芯片 CP2102，需要驱动安装，

USB_D+-----接单片机的 P0_3

USB_D - ----默认接单片机的 P0_5,也可以接到 P0_1



SDK 介绍（参考官网提供的软件架构）

dk_apps: Development Kit application directory 开发包应用文档（包含官网提供的 DEMO 程序）。

Prod_test: production test 的用来进行产品测试的固件源代码。

proximity: monitor_fe、monitor_fe_usb、reporter_fe、reporter_fe_spi、repoeter_fe_usb 这几个工程都是主控模式的防丢器 DEMO；report_fh 是非主控的防丢器 DEMO。**report_fh 是常用的**，其它几个可以不用理会。讯联电子提供的防丢器测试程序就是基于这个修改而来的。这里提下 **_fe** 是 Fully_Embedded 的缩写，带这个字眼的工程 DA14580 是在**主控模式**下运行；**_fh** 是 Fully_Hosted 的缩写，带这个字眼的工程 DA14580 运行在**非主控模式**下。

template: 这个是官方提供的一个工程模版。以后我们的应用都可以在这个模版的基础上修改。

throughput_eval: 这个是一个评估数据吞吐率的源代码。（用于**与手机 app 通信**）

patch_code: 里面放的是分散加载文件

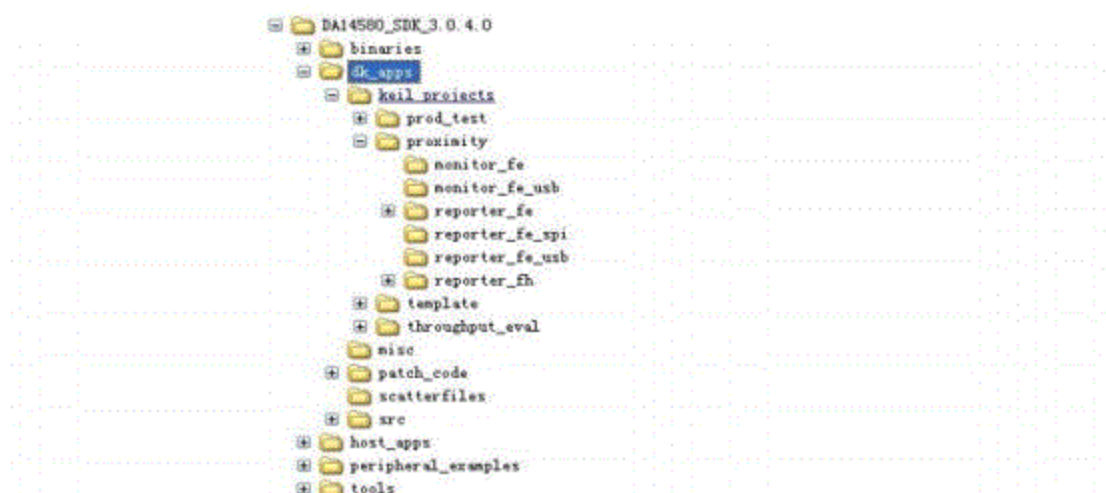


Figure 5 dk_apps 文件夹

dk_apps 文件夹:

dk_apps 文件夹下的内容才是我们的重点。官方提供的 DEMO 程序的项目工程都放在 keil_projects 文件夹里面。

prod_test: 就是我们前面说的用来进行产品测试的固件源代码。

proximity: monitor_fe、monitor_fe_usb、reporter_fe、reporter_fe_spi、reporter_fe_usb 这几个工程都是主控模式的防丢器 DEMO; report_fh 是非主控的防丢器 DEMO。report_fh 是常用的，其它几个可以不用理会。讯联电子提供的防丢器测试程序就是基于这个修改而来的。这里提下 _fh 是 Fully_Embedded 的缩写，带这个字眼的工程 DA14580 是在主控模式下运行; _fh 是 Fully_Hosted 的缩写，带这个字眼的工程 DA14580 运行在非主控模式下。

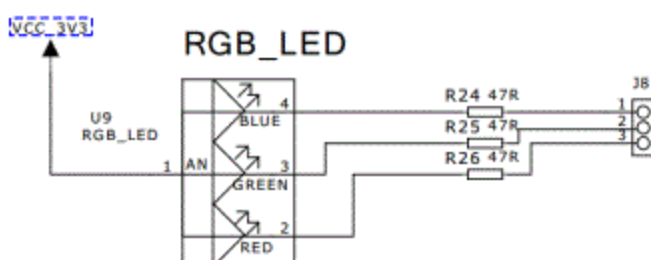
template: 这个是官方提供的一个工程模版。以后我们的应用都可以在这个模版的基础上修改。

throughput_eval: 这个是一个评估数据吞吐率的源代码。

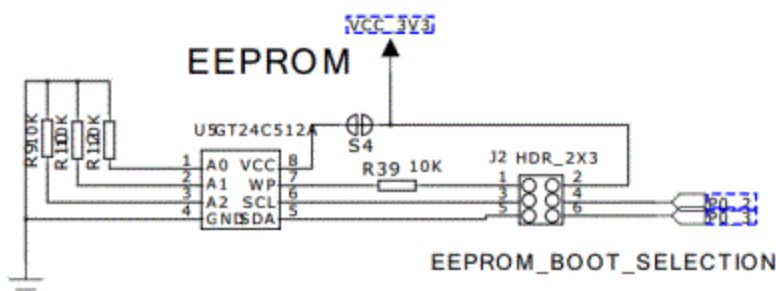
patch_code: 里面放的是分散加载文件

接口说明

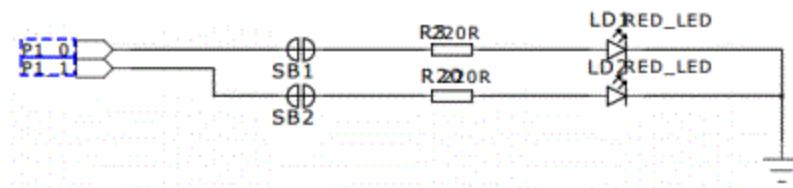
RGB_LED: 接在 J8 上，需要跳线与单片机连接？



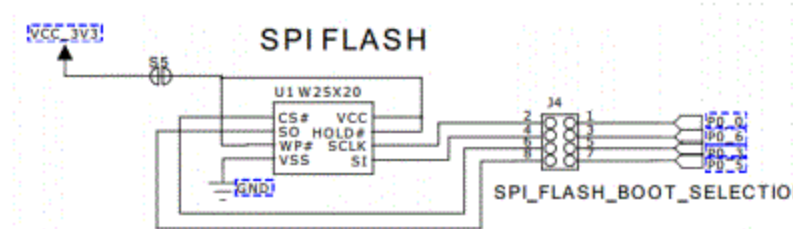
EEPROM: 需要下载到 EEPROM 中时，J2 需要接跳线帽



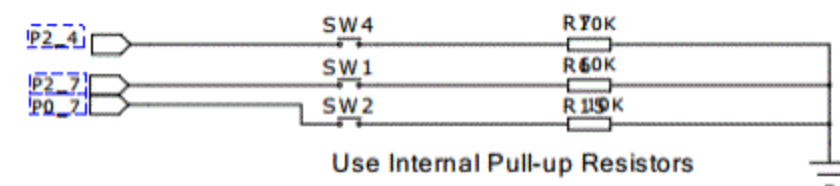
用户 LED:接 P1_0 和 P1_1



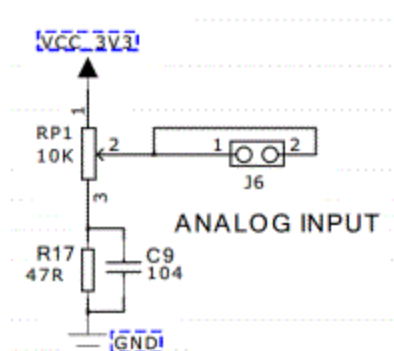
SPI_Flash: 把拨码开关全部拨到上面, 则从 Flash 启动



用户按键: P2_4, P2_7, P0_7



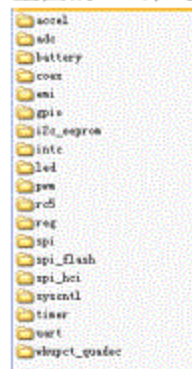
电量模拟采集:



蜂鸣器: P2-0; 继电器: P2-3

官网驱动示例

位置: \DA1458x_SDK_3.0.6\dk_apps\src\plf\refip\src\driver, 比如写 MPU6050 就可以参考这
里面的 i2c 来写。



编程学习（WT 科技）

裸板 LED 点亮

- ① 包含 gpio.c, 因为引脚置高置低调用的是 gpio.c 中的函数
- ② 初始化 gpio, 在 peripherals.c 的 set_pad_functions 中。GPIO_ConfigurePin(LED1_PORT, LED1_PIN, OUTPUT, PID_GPIO, false);
- ③ 端口定义在 peripherals.h 中 #define LED1_PORT GPIO_PORT_1 #define LED1_PIN GPIO_PIN_0
- ④ 在主函数中调用 GPIO_SetActive(LED1_PORT, LED1_PIN); GPIO_SetInactive。
- ⑤ 主函数中调用 periph_init(); //外围设备初始化

裸板定时器 timer

- ① 主函数中调用 timer_init();对时钟分频、计数大小等等初始化。
- ② 定时器的中断处理函数在 pwm.c 中的 SWTIM_Handler 中。不需要重新初始化定时器

ADC 实验

只有 P0.0-P0.3 这四个引脚支持 ADC, 从 ADC.h 中也可以看出

```
/* ADC channels */
#define ADC_CHANNEL_P00 0
#define ADC_CHANNEL_P01 1
#define ADC_CHANNEL_P02 2
#define ADC_CHANNEL_P03 3
#define ADC_CHANNEL_AVS 4
#define ADC_CHANNEL_VDD_REF 5
```

中断实验

从 CM0.S 文件里可以看到只有四个中段处理函数, 如果是

```
DCD GPIO0_Handler
DCD GPIO1_Handler
DCD GPIO2_Handler
DCD GPIO3_Handler
DCD GPIO4_Handler
```

UART 实验

①Uart 带 booting 功能:

P0.0&P0.1、P0.2&P0.3、P0.4&P0.5、P0.6&P0.7 皆可用作带 Boot 功能的 UART，作为调试、升级口，一般默认采用 P0.4(TX)&P0.5(RX)，在其他 I/O 口资源紧张的情况下也可当作普通 I/O 口使用，前提是该复用功能不会造成冲突

②Uart 不带 booting 功能:

如 UART 不需要上电时启动，也可采用除 P0[0:7]以外的其他 I/O 口模拟，比如仅需与 MCU 通讯的 UART 功能。

RW 系统内部定时器 LED 闪烁

- ① 在 `periph_setup.c` (外围设备初始化) 的 `GPIO_reservations` 中申请使用的 GPIO 口
- ② 在 `set_pad_functions` 函数中初始化 GPIO 口 `GPIO_ConfigurePin`
- ③ 在 `app_api` 的 `APP_MSG` 中添加时间定时器任务标识 `LED_FLASH_TIMER`
- ④ 在 `app_test_proj.h` 中添加消息处理函数 `led_flash_timer_handle(****)`
- ⑤ 在 `app_test_proj.c` 中添加头文件 “`GPIO.h`” 和 `led_flash_timer_handle` 函数实体
- ⑥ 在 `app_task_handlers.h` 中把任务标识符 `LED_FLASH_TIMER` 与处理函数 `led_flash_timer_handle` 关联起来
- ⑦ 在 `app_test_proj.c` 的 `app_adv_func` 函数中添加定时器初始化 `app_timer_set`，以后的定时器重新填充时间则有消息处理函数中实现。其中 `app_timer_set` 是使用的 RW 系统内部时钟。想要关闭 LED 闪烁，调用 `app_timer_set(LED_FLASH_TIMER,TASK_APP,10)`(10 大概就是 100ms); `ke_timer_clear(LED_FLASH_TIMER,TASK_APP);`

RW 硬件定时器实现 LED 闪烁

- ① 在 `periph_setup.c` 添加 `timer_init()` 函数，和裸板的初始化函数一样
- ② 在 `periph_setup.c` 中添加 `timer_callback()` 函数;
- ③ 在 `periph_setup.c` 中的 `periph_init` 函数中添加 `timer_init()` ; `timer0_register_callback(timer_callback);`对 timer 的一个初始化
- ④ 在 `app_test_proj.c` 中添加 `timer0_start()`;此函数也是在 `PWM.c` 中定义的。

RW 系统实现 ADC 转换

- ① 在 `periph_setup.c` 中对 ADC 引脚进行初始化 `GPIO_reservations`、`GPIO_ConfigurePin` 函数中
- ② `periph_init` 中进行 ADC 初始化，使能 ADC 引脚等功能。
- ③ 开启一个 ADC 定时器任务，在任务中进行 ADC 采集和执行相应的动作

RW 系统实现中断

- ① 配置中断引脚
- ② 在 `periph_setup.c` 中初始化中断函数 `gpio01_init()`。
- ③ 在 `periph_init()` 中调用 `gpio01_init()` 函数
- ④ 添加中断处理函数 Handler
- ⑤ 添加中断函数，要在 `\dk_apps\misc\rom_symdef.txt` 中的对应的 `GPIOx_Handler` 注释掉。

RW 系统实现串口

- ① 添加 driver 下 `uart1` 文件下的 `uart.c`
- ② 在 `uart.c` 中添加头文件 `"app.h""ke_timer.h""app_api.h"` 及 `uart` 中断处理函数(则可以对接收回来的数据进行处理)
- ③ 在 `periph_setup` 中添加接口配置语句
- ④ 在 `periph_init` 中调用 `uart_init`; (里面会进行波特率之内的设置等等)
- ⑤ 如果要定时发送，则添加一个定时器，并在定时器处理函数中发送

BLE 基础学习

单模设备：只支持 BR(BasicRate)或者 BLE(Bluetooth Low Energy)

双模设备：同时支持 BR 和 BLE(为蓝牙 4.0 的主题)

Advertising: 广播

Standby: 准备

Scanning: 监听/扫描

Connected: 已连接

Initiator: 发起人

发起连接请求的叫主 master，接收连接请求的叫从 slaver

服务与特征值

Profile 通常用一个或多个**服务**组成，每个服务对应特定功能，比如心率、体温分别对应一个服务。一个服务由包含一个或多个**特征值** (characteristic value)，比如心率服务会有一个心率特征值。每个特征值必须占用一个**特征申明结构**，它是服务端和客服端共享的读写空间。这个特征值就是用于设备之间的**数据交流**。

UUID 代表服务与特征值的唯一通用标识。有 128 和 32 位两种形式，有一些服务已经分配了固定的 UUID，所以用户自定义的服务与特征值不能使用这些 UUID。

特征值的权限

Permissions（权限）定义了特征值的访问权限，比如规定某个特征值只能被读，或者可读可写等。

通信方式

已连接设备之间的通信方式：

Read：使用指定的 handle 读特征值；

Write：使用指定的 handle 写特征值；

Indication：某个特征值发送到客户端，需要确认；

Notification：某个特征值发送到客户端，不需要确认，设备向手机 APP 发送数据的方法。

广播

广播包的发送是单向的，不需要任何连接。设备发送广播包进入广播状态。

广播包可以包含特定的数据定义，最大 31 个字节。

广播包可以直接指定特定的设备，也可以不指定。

广播包中可以声明是可被连接的设备，或者是不可连接的设备

广播间隔是指两次广播时间之间的最小间隔（0.625ms 的倍数+随机延时），其中随机延时为 0~10ms，为了避免多个设备之间的数据碰撞。

DA14580 的 BLE 协议栈只在 37 通道中广播。

BLE 实战

最简单的蓝牙程序

① 在 da14580_config.h 文件中，更改应用程序标识：#define CFG_APP_TEMPLATE 改成 #define CFG_APP_TEST 。

② 在 app_api.h 中添加如下代码#include "app_test_proj.h"

③ 在 app_test_proj.c 中的 app_db_init_func 函数的 switch 中添加 case (APP_DIS_TASK):

```
{  
    app_dis_create_db_send();  
}break;
```

④ 在 profile 中添加 diss.c,diss_task.c,prf_utils.c 三个文件，并在 da14580_config.h 中使能 DISS（Device Information Service Server 设备信息服务），使能过后会在 rwble_hl_config.h 中会把 BLE_DIS_SERVER 置 1。

⑤ 在 app 中，添加 app_dis.c, app_dis_task.c 两个文件。并把这两个文件的头文件包含在 app_test_proj.h 中

Profile 介绍

Profile 是 BLE 的配置文件

一个 profile 中有 4 个文件: xx.h, xx.c, xx_task.h, xx_task.c (diss.c, diss.h, diss_task.h, diss_task.c)

Xx.h 中: 服务、特征值、UUID、环境变量等

Xx.c 中: 服务、特征值、配置初始化

Xx_task.h: 消息、消息结构体变量

Xx_task.c: 消息对应的处理函数

定义好服务过后，需要定义对应的应用层文件

应用层文件也有 4 个: app_xx.h, app_xx.c, app_xx_task.c, app_xx_task.h

与手机通信

- ① 打开官网 SDK 中的 throughput_eval_peripheral 工程。
- ② UUID 可以在 streamdatad.h 中修改
- ③ 在 streamdatad_task.c 中修改一次性发送的字节数
- ④ Gattc_write_cmd_ind_handler 函数中处理接收到的语句
- ⑤ 修改 MAC 地址和名字在 nvds.c 中

I2C 模块介绍

从设备默认地址 0x055

中断或者轮询操作模式

I2C_CON_REG: 通信速度、主从选择、决定地址为 7 位还是 10 位开始

I2C_TAR_REG: 目标地址设置

I2C_DATA_CMD_REG: 接收发送数据缓存与命令寄存器

I2C_CLR_TX_ABRT_REG: 清除 TX_ABRT 中断

I2C_ENABLE_REG: 使能寄存器

I2C_STATUS_REG: 状态寄存器

I2C_RXFLR_REG: 接收 FIFO 数目寄存器

I2C_TX_ABRT_SOURCE_REG: I2C 发送异常终止源寄存器

MPU6050 驱动

寄存器: 采样速率分频寄存器

配置寄存器

角速度配置寄存器

加速度配置寄存器
加速度测量值寄存器
角速度测量值寄存器
电源管理寄存器
设备身份验证寄存器

OTA 升级

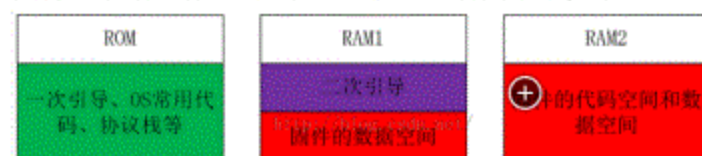
SUOTA 空中软件升级 (software update over the air), 一般的 OTA 要求的固件位 bin 格式, 需要 hex 转化成 bin (所以升级需要软件 hex2bin)。为了让引导程序快速判断是否是固件, 宜在固件头部进入固件标识字段, 如 0x7050 是 DA14580 的第一次引导辨别的固件标识, 而 0x7051 是二次引导辨别的固件标识。

外存固件分布图:



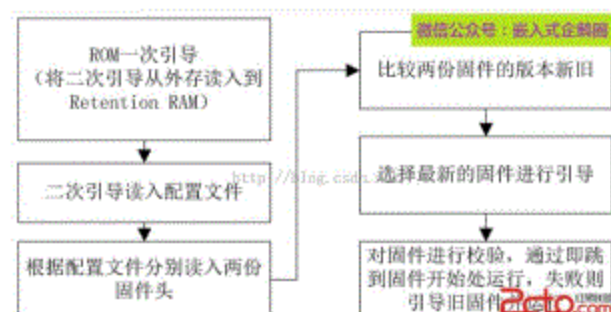
二次引导程序运行过程中也要将最新固件引导到 RAM。因此, 二次引导程序和固件所运行的 RAM 中是不可以重复的, 必须分布独立的内存空间。

二次引导程序在引导出固件后, 它的使命就完成了, 因此它所占有的内存空间应该被定义为固件的数据运行空间 (如.bss 段), 否则就浪费了。



当然没有外存是不可以 OTA 的, 如果烧在 OTP 中, 则不可以更改程序了。

启动过程:



OTA 升级步骤:

- ① 烧写产品头文件(判断是否为一个完整的程序, 是否为最新固件)
- ② 烧写镜像文件 (mkimage 制作 img 文件, mkimage 可以在 SDK 中找到)
- ③ 烧写二次引导程序 (secondary bootloader)

二次引导烧录在 OTP 中, 则直接引导去加载相应的程序到 SRAM 中。

Bin 文件生成方法: 直接把 xxx.hex 文件拖到 hex2bin.exe 中, OK, 则生成了 bin 文件。

操作步骤: 参考 (基于 SDK5.03 的 SUOTA 使用方法)

- ① 以 SDK5.0.4 为例, 打开 5.0.4\projects\target_apps\ble_examples\prox_reporter\Keil_5
- ② 在"user_config.h"中修改设备名称"SUOTA01", 生成 hex 命名为 fw01.hex。用 hex2bin

转化成 fw01.bin 文件

③ 同理生成 fw02.hex。

④ 在 \5.0.4\sdk\platform\include 中复制 ble_580_sw_version.h 文件。重命名为 fw01_version.h 何 fw02_version.h (第二版本时间比第一版本低, 且版本要高)

⑤ 打开 5.0.4\utilities\secondary_bootloader 工程生成 secondary_bootloader.hex

⑥ 把 fw01.hex 通过 hex2bin 转化成 bin 文件 (直接拖在里面就可以了)

⑦ 开始运行 --CMD-- 指定到目录 --mkimage.exe single fw01.bin fw01_version.h fw01.img。生成升级包 img。同理 fw02.img

⑧ mkimage.exe multi spi secondary_bootloader.bin fw01.img 0x8000 fw02.img 0x13000 0x1f000 multi_part.bin 生成最终 multi_part.bin 文件。

⑨ 下载--选择 burn&verify 进行下载 (工程到此为止)

⑩ 升级步骤

11 安装 suota 软件, 通过电脑把升级 img 文件拷贝到 suota 中

12 按照默认的设置, image bank 选择为 0, 按 send to device 完成。

移植空中升级任务

① 在 profile 和 app 中添加 spotar 服务和任务

② Da14580_config 中添加, 注意选择的是 I2C_DISABLE, 所以采用 SPI 升级

```
//-----空中升级任务
#define CFG_PRF_SPOTAR
#ifdef CFG_PRF_SPOTAR
    #define CFG_SPOTAR_I2C_DISABLE
    #define CFG_SPOTAR_UPDATE_DISABLE
    #define CFG_APP_SPOTAR 1
    #define SPOTAR_PATCH_AREA 1
#endif
```

③ 添加..\..\..\src\modules\app\src\app_project\spotar 编译路径

Mpu6050 计步

地址默认 0X68, 高七位由寄存器决定, 低一位决定数值

DMP 就是指 MPU6050 内部集成的处理单元

DMP_PEDOMETER: 计步特性, 一直使能, 在 MPU 上电后即在 DMP 中运行。

包括注入计步器等针对特定应用的特性

电池电量

可以开通一个 Battery service, 里面可以直接获取电量的多少, 参考官网提供的防丢实验 zai。

① 在 app 中添加 \src\modules\app\src\app_profiles\bass (app_batt 和 app_batt_task)

② 在 profile 中添加 src\ip\ble\hl\src\profiles\bass\bass 中的基础服务

③ 在 driver 中添加 battery 驱动 src\plf\refip\src\driver\battery

④ 在 da14580_config.h 中添加 #define CFG_PRF_BASS

⑤ 在 app_task_handler.h 中添加 "#include "bass_task.h" " #include "app_batt_task.h" "

(为什么别的都没有添加, 这里添加了)

⑥ 在工程文件 `app_db_init_func()` 函数中添加

```
app_stream_create_db();
    break;
#endif
//-----添加
#if (BLE_BATT_SERVER)
case (APP_BASS_TASK):
{
    app_batt_create_db();
} break;
#endif //BLE_BATT_SERVER
default:
```

⑦ 在 `app_disconnect_func` 中添加

```
uint8_t state = ke_state_get(task_id);
//-----电池服务
#if BLE_BATT_SERVER
app_batt_poll_stop();
#endif // BLE_BATT_SERVER
```

⑧ 在 `app_init_func` 函数中添加

```
//-----添加
#if (BLE_BATT_SERVER)
    app_batt_init();
#endif
```

⑨ 在 `app_connection_func` 函数中添加

```
//-----添加
#if BLE_BATT_SERVER
//cur_batt_level = 0;
app_batt_enable(cur_batt_level, USE_BAT_LEVEL_ALERT, GPIO_BAT_LED_PORT, GPIO_BAT_LED_PIN);

app_batt_poll_start(6000); //10 mins
#endif // BLE_BATT_SERVER
```

⑩ 在工程头文件中添加头文件

```
//-----添加电池服务
#if (BLE_BATT_SERVER)
#include "app_batt.h"
#include "app_batt_task.h"
#include "periph_setup.h"
#endif
```

11 在 `periph_setup.h` 中添加

```
#define USE_BAT_LEVEL_ALERT 1
#define GPIO_BAT_LED_PORT GPIO_PORT_1
#define GPIO_BAT_LED_PIN GPIO_PIN_0
```

添加 HID 服务

① 在 profile 中添加 `src\ip\ble\h\src\profiles\hogp\hogpd` 里的服务文件

② 在 `da14580` 中添加 `#define CFG_PRF_HOGPD`

③

HID 调节音量(改蓝牙键盘程序)

利用 DA14580 官网提供的透传实验, 不能通过系统连接, 只能连接 app Profile 和应用总是基于 GAP 和 GATT 上。

HID 设备的范畴, 但其数据通信是走的蓝牙协议

HIDS | HID 服务 | 显示 HID 报告和其它 HID 数据, 旨在用于 HID 主机和 HID 设备。

官方设计的 keyboard 支持 8*18 按键, 矩阵扫描形式

```

DA14580_config.h 中#define CFG_MULTI_BOND → // #define CFG_MULTI_BOND
app_kbd_config.h#define MITM_ON → // #define MITM_ON
app_kbd_config.h#define EEPROM_ON → // #define EEPROM_ON
app_kbd_config.h #define MATRIX_SETUP (8) // changed from (10)
app_kbd_config.h 中打开 NORMALLY_CONNECTABLE_ON
app_kbd_proj.c // app_set_deep_sleep();
                // app_set_extended_sleep();
                app_disable_sleep();

```

Consumer HID: 音量加的键值: 0xF406, 音量减的键值: 0xF407, home 键 0xF415; 播放/暂停键值 0xF404; 上一曲 F401, 下一曲 F400; 锁屏界面下 0x66 可以唤醒锁屏 (app_kbd_matrix_setup_1.h) matrix--矩阵的意思。

在 HID-Usage-table 中的键值: 0x48 键值 paush 暂停; 0x4A 键值 home (测试不行啊); 0x80 键值 音量加 (测试不行); 0x81 音量减 (测试不行); 文档说明: 可以使用 HID 表中的 0x65 以下,

3, HID报表, 通常定义为:

- a Keyboard (包括多键同时压下与抬起)
- b, Mouse (能实现加速移动游标, 短距离移动游标)
- c, 电源管理 (电脑关机, 待机, 唤醒), 开机要主板配合
- d, 消费类, 在影音设备常常用到, 如(像手机耳机):
Play, Rec, Pause, Stop, Next, skip, Mute,

增加锁屏按键到 Consumer HID

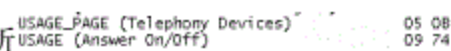
① 打开  软件, 双击 USAGE_PAGE, 再双击 Consumer device---双击 usage page---选择 power, 可以看出, power 的键值要赋值为, 0x09

0x30。 

② 在 app_kbd_proj_task.c 中, 把音量减改成 09 30; 然后音量减则变成了锁屏

增加接听电话

在 keyboard device (键盘设备) 和 consumer device (消费类设备) 中, 没有包含电话接听。

只有在 telephone device 中才包含有电话接听 

Keyboard 代码解析

App_kbd_matrix.h 这个函数中定义了键盘的矩阵形式
App_kbd_config.h 中对矩阵的形 #define MATRIX_SETUP
App_kbd_key_matrix_setup_10.h 对键盘对应引脚初始化

(8)

有按键按下（且蓝牙连接）-->app_kbd_prepare_keyreports-->if (app_kbd_buffer_has_data())-->prepare_kbd_keyreport()-->kbd_process_keycode-->
 kbd_process_keycode(&kbd_keycode_buffer[kbd_keycode_buffer_head]);-- 短接 P13 和 P11 时值为 0，短接 P13 和 P06 时，值为 1
 发送一个键值如下：

```

kbd_keycode_buf.flags=0x10;
kbd_keycode_buf.output=0x00;
kbd_keycode_buf.input=0x01;
kbd_process_keycode(&kbd_keycode_buf);
kbd_keycode_buf.flags=0x00;
kbd_keycode_buf.output=0x00;
kbd_keycode_buf.input=0x01;
kbd_process_keycode(&kbd_keycode_buf);

```

app_state_update---用来更新状态机，在 IDLE 空闲状态的时候，进入 DEEP DEEP，如何唤醒？不广播了，是在状态机中，广播状态下，ADV_TIMER_EXP_EVT 广播定时器超时了。

取消停止广播的动作：app_hid_adv_timer_handler--屏蔽下面这句话

//app_state_update(ADV_TIMER_EXP_EVT);执行这个会进入 app_adv_stop(); EXP--expires 到期

adv_timer_remaining---广播的时间，app_start_adv_msg_handler 函数中开始

#define NORMALLY_CONNECTABLE_ON --- 不设置，则为空闲状态

void app_mitm_passcode_report(uint32_t code)//发送键值

int app_kbd_prepare_keyreports(void)

app_kbd_buffer_has_data

prepare_kbd_keyreport

send_hid_report

hogpd_ntf_send---里面有 stream 种的透传 prf_server_send_event；有可能是 HID 传输

hogpd_boot_report_upd_req_handler---这应该是发送 report

hogpd_report_upd_req_handler

hogpd_boot_report_upd_req_handler//以上这三个函数都用到 hogpd_ntf_send

/// Request sending of a report to the host - notification

HOGPD_REPORT_UPD_REQ,

请求发送一个报告是一个函

数名称

app_kbd_scan_matrix---扫描矩阵按键并处理结果

kbd_key_report

report_list

Keyboard 里面有电池电量的指示，可以参考

/// APP Task messages

① 在 app_api 中添加 enum APP_MSG 任务标识

② app_task_handlers.h 中添加消息对应的处理函数

{APP_BATT_TIMER, (ke_msg_func_t)app_batt_timer_handler};

③ 在 app_adv_func 中添加轮询电量的函数

```
app_batt_poll_start(BATTERY_LEVEL_POLLING_PERIOD/10); // Start polling
```

④ App_batt_poll_start 函数，开始了定时器--APP_BATT_TIMER 函数

```
void app_batt_poll_start(uint16_t poll_timeout)
{
    bat_poll_timeout = poll_timeout;

    app_timer_set(APP_BATT_TIMER, TASK_APP, 10); //first poll in 100 ms
}
```

⑤ 定时器任务的处理函数

```
int app_batt_timer_handler(ke_msg_id_t const msgid,
                           void const *param,
                           ke_task_id_t const dest_id,
                           ke_task_id_t const src_id)
{
    app_batt_lvl();

    app_timer_set(APP_BATT_TIMER, dest_id, bat_poll_timeout);

    return (KE_MSG_CONSUMED);
}
```

⑥ 停止电量采集函数

```
void app_batt_poll_stop(void)
{
    ke_timer_clear(APP_BATT_TIMER, TASK_APP);
}
```

⑦ 电量采集 LED 报警，再建一个定时器任务 APP_BATT_ALERT_TIMER--

--app_batt_alert_timer_handler; 任务专门负责 LED 闪烁，电量采集里面判断是否使能 LED 闪烁。

```
int app_batt_alert_timer_handler(ke_msg_id_t const msgid,
                                void const *param,
                                ke_task_id_t const dest_id,
                                ke_task_id_t const src_id)
{
    //read LED GPIO state
    if(bat_lvl_alert_used)
    {
        if (bat_led_state)
        {
            GPIO_SetInactive( bat_led_port, bat_led_pin);
            bat_led_state = 0;
            app_timer_set(APP_BATT_ALERT_TIMER, dest_id, 20);
        }
        else
        {
            GPIO_SetActive( bat_led_port, bat_led_pin);
            bat_led_state = 1;
            app_timer_set(APP_BATT_ALERT_TIMER, dest_id, 5);
        }
    }

    return (KE_MSG_CONSUMED);
}
```

计步实验中添加电池服务

① 在 da14580_config 中添加#define CFG_PRF_BASS---程序会在 rwble_hl_config.h 中把 BLE_BATT_SERVER 置 1

② 在 app_api.h 中会添加两个服务 APP_BATT_TIMER(用于定时采集电量)、APP_BATT_ALERT_TIMER(用于电池电量过低报警)

③ 在 app_task_handlers.h 中，会对这两个任务标识添加对应的处理函数

④ 采集任务的开始函数在 app_batt.c 中定义，在工程文件的蓝牙连接函数中开始,停止在断开函数中

```
void app_batt_poll_start(uint16_t poll_timeout)
{
    bat_poll_timeout = poll_timeout;

    app_timer_set(APP_BATT_TIMER, TASK_APP, 10);
}
```

- ⑤ battery level---电池电量

低功耗

extended sleep: RAM 数据保存

Deep sleep: RAM 数据不保存

唤醒: 定时器(唤醒定时器和 BLE 定时器), 或者外部唤醒中断(外部输入事件触发唤醒定时器), 可配置任意引脚为唤醒引脚。CPU 执行 WFI 指令时, 进入睡眠模式。内核定时器 ke_timer。

① 在 da14580_config.h 中开启 CFG_EXT_SLEEP, 主循环过后, 就会进入低功耗模式, 如果开启有定时器服务, 那么定时器会不断的唤醒, 所以此时如果是定时电量 LED, 则依然可以闪烁。

② wkupct_register_callback, 唤醒中断过后别调用, 可以进行一些初始化(内核, 晶振可以让它工作起来)

③ wkupct_enable_irq(配置唤醒中断的引脚, 电平, 中断延时等) wkupct_disable_irq(为注销唤醒中断)

④ app_set_extended_sleep---进入浅度睡眠

⑤ app_ble_ext_wakeup_on(); 使能外部唤醒, 紧接着要执行 2.3 步

BLE 透传

采用 Dialog 官方提供的 DSPS.APK(主要安卓 4.0 以上, 三星除外的支持)。

USBdongle

BLE 的协议分析固件, 用来抓取空气中的 BLE 数据包

BLE 协议中, 有两个角色, 周边(Periphery)和中央(Central); 周边是数据提供者, 中央是数据使用/处理器。手机作为中央, BLE 设备提供数据。

添加 streamdatad 服务

① Profile 中添加 streamdatad.c, streamdatad_task.c

② App 中添加"app_streamdatad.c" "app_streamdatad_task.c"和 app_stream_queue。
(\dk_apps\src\modules\app\src\app_utils\app_stream_queue)

③ 在 da14580_config.h 中添加

```
#define CFG_INTEGRATED_HOST_GTL
#define CFG_PRF_STREAMDATAD
#define CFG_STREAMDATA_QUEUE
#define METRICS
#define CFG_INTEGRATED_HOST_GTL
#define CFG_APP_THROUGHPUT_PERIPHERAL
```

④ 在 app_task_handle.h 中添加头文件

```
#include "streamdatad.h"
#include "app_streamdatad.h"
#include "streamdatad_task.h"
#include "app_streamdatad_task.h"
```

⑤ 在工程中需要添加

```
#if (STREAMDATA_QUEUE)
#include "app_stream_queue.h"
#endif
```

```
uint8_t test_state __attribute__((section("retention_mem_area0"), zero_init));

#define MAX_TX_BUFS (32)
#define STREAMDATAD_PACKET_SIZE (20)
uint8 test_pkt [MAX_TX_BUFS][STREAMDATAD_PACKET_SIZE];

#ifdef METRICS
extern struct metrics global_metrics_tosend;
#endif
```

app_connection_func 函数中

```
#if BLE_INTERPRETED_HOST_GTS
struct app_ext_connect_ind *cmd = RX_MSG_ALLOC(RFX_EXT_CONNECT_CMD_IND,
        TASK_GTL, TASK_APP, app_ext_connect_ind);

cmd->state = true;
cmd->conn_handle = param->connhdl;

cmd->addr_type = param->peer_addr_type;
memcpy(cmd->addr, param->peer_addr.addr, BD_ADDR_LEN);

// Send the message
rx_msg_send(cmd);
#endif

#if (STREAMDATA_QUEUE)
stream_setup_isr_tomax(param->con_interval);
#endif

#if BLE_STREAMDATA_DEVICE
app_streamdatad_enable();
#endif
```

app_send_pairing_rsp_func 函数内容注释掉

app_send_tx_exch_func 的内容注释掉

app_ext_transmit_cmd_handler 函数中

```
/// Start transmission
#if (STREAMDATA_QUEUE)
if (param->action == APP_STREAM_START)
{
    if (test_state != APP_STREAM_START)
    {
        set_pxact_gpio();
        test_pkt_init();
    }
    test_state = APP_STREAM_START;
}
else
    test_state = APP_STREAM_STOP;
#endif
```

添加函数

⑥ 在工程头文件中添加

```
#if BLE_STREAMDATA_DEVICE
#include "app_streamdatad.h"
#include "app_streamdatad_task.h"
#endif
```

```

struct app_uart_test_req
{
    uint8_t value;
};
struct app_uart_test_rsp
{
    uint8_t value;
};
struct app_ready_ind
{
    int x;
};
struct app_scan_req
{
    uint8_t x;
};
struct app_ext_scan_rsp_ind
{
    uint8_t x;
};

```

编译路径

```

..\\..\\..\\src\\ip\\ble\\hl\\src\\profiles\\streamdata\\streamdatad
..\\..\\..\\src\\ip\\ble\\hl\\src\\profiles\\streamdata\\streamdatah
..\\..\\..\\src\\modules\\app\\src\\app_profiles\\streamdatad
..\\..\\..\\src\\modules\\app\\src\\app_utils\\app_stream_queue

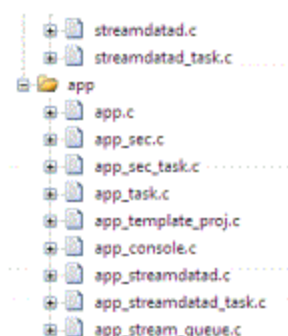
```

添加最简单的 streamdatad

- ① 在 profiles 和 app 中添加服务及应用，添加路径
- ② 在 da14580 中定义使能 stream
- ③ 在 app_task_handle 中添加头文件
- ④ 在 app_stream_queue 中注销//#ifdef METRICS
- ⑤ 在工程.h 中添加头文件
- ⑥ 在工程文件 app_db_init_func() 函数中添加
- ⑦ 在 app_disconnect_func 中添加
- ⑧ 在 app_connection_func 函数中添加、

添加最简单的 streamdatad2（添加的最简单的，但是不能通信）

- ① 在 profiles 和 app 中添加服务及应用，添加路径



```

..\\..\\..\\src\\ip\\ble\\hl\\src\\profiles\\streamdata\\streamdatad
..\\..\\..\\src\\ip\\ble\\hl\\src\\profiles\\streamdata\\streamdatah
..\\..\\..\\src\\modules\\app\\src\\app_profiles\\streamdatad
..\\..\\..\\src\\modules\\app\\src\\app_utils\\app_stream_queue

```

② 在 da14580 中定义使能 stream

```
//=====
#define CFG_PRF_STREAMDATAD
#define CFG_STREAMDATA_QUEUE
//=====

int stream_rcv_data_packet_ind_handler(ke
                                     str
                                     ke
                                     ke
)
{
    // add_packets_rx(param->size);
}
```

PROGRESS

③ 在 app_template_proj.h 中添加

```
//=====
#if BLE_STREAMDATA_DEVICE
#include "app_streamdatad.h"
#include "app_streamdatad_task.h"
#endif
```

④ 在工程文件 app_db_init_func()函数中添加

```
//=====
    #if (BLE_STREAMDATA_DEVICE)
    case (APP_STREAM_TASK):
    {
        // Add proxr Service in the DB
        app_stream_create_db();
    } break;
    #endif
//=====
```

⑤ 其他：在 app_streamdatah_task.c 屏蔽 `//#include "app_stream_queue.h"`，da14580_config.h 中屏蔽 `//#define CFG_STREAMDATA_QUEUE`，删了 app_stream_queue.c 都可以有服务出来，只是不能通信而已

读写 Flash-SPI

目前程序占的空间，难道外部寻址空间是从 0x20000000 开始的？不可能，16 位的单片机， $2^{16}=65535=0xffff$

0x20000000	00 98 00 20 55 04 00 20	リ U
0x20000008	5D 04 00 20 75 04 00 20] u
0x20000010	00 00 00 00 00 00 00 00	

0x200078E8	01 00 00 00 08 00 00 00	
0x200078F0	01 00 00 00 07 00 06 00	
0x200078F8	06 0E 00 00 -- -- -- --	

可用空间， $0x7ff8=32.760k$

0x07FD8	FF FF FF FF FF FF FF FF
0x07FE0	FF FF FF FF FF FF FF FF
0x07FE8	FF FF FF FF FF FF FF FF
0x07FF0	FF FF FF FF FF FF FF FF
0x07FF8	FF FF FF FF FF FF FF FF

四线：CLK=P0.0 CS=P0.3 MOSI=P0.5 MISO=P0.6

SPI 通信, CLK 上升沿时发送 MOSI 中的电平, 下降沿时接收
开发板用的是 W25X20, 2M-bit=256k 字节 地址大小 2*1024*1024;---

STM32 有关 SPI FLASH 实验

W25Q128, 128M-BIT/8=16M 位, 大小就叫 16M。分为 256 块---每块 64k---每块 16 个扇区---
每个扇区 4k (所以一次最小擦除 4k)

写状态寄存器 01h。

- ① 设置 SPI 时钟
- ② 设置 SPI 模式, 主机模式, 设置数据格式 8 位, CLK 时钟极性, 时钟频率, 数据格式 (MSB 在前还是 LSB 在前)
- ③ 使能 SPI
- ④ 固件提供了 SPI 的发送和接收的函数, 查看 SPI 传输状态, 是否传输完成
- ⑤ 根据 SPI 函数, 写 FLASH 的读写函数, 写包括数组指针、长度、地址

问题及解决方法

问题一: 外部按键中断, 单击一次会进入中断多次, why?

需要接上拉电阻, 触发的时候直接接地, 如果接地电阻过大, 是不行的。也可以软件设置 INPUT_PULLUP。注意最好是直接接地。

问题二: 每次运行的时候, 会进入中断一次, why?

未解决

问题三: 睡眠状态, BLE 是否会断开?

问题四: HID 可以挂接电话吗?

问题五: 为什么蓝牙连接系统不行, 连接 app 则可以, 目前只有 HID 的蓝牙键盘能连接系统。

问题六: 开启的服务可以停止吗, 比如 HID 不用的时候, 把服务关闭

问题七: SPI 下载不进入

注意, SPI 的引脚和串口引脚共用, 所以要把 P05 引脚和串口分开

问题八: SPI 下载无法连接

可以尝试在 Burn Header 中连接, 如果还不行, 就短接 P05 和串口, 然后复位 (这样程序就不会执行), 然后用 keil 进行调试, 调试成功过后再在 SmartSnippet 中连接。

问题十: 为什么蓝牙不可以直接连接苹果系统

问题十一: 金刚狼的 BLE 连接不上解决办法

先用 nRF 软件连接上, 使单片机有数据不断发送回来, 然后断开 nRF, 采用金刚狼软件连接

问题十二: nRF 其实是可以看到接收回来的值的

以计步为实验, 在透传服务的第二个特征值前面, 把连着三个向下的箭头取消, 则可以不断的接收发回来的值了

问题十三: 添加了 streamdatad 服务, 却搜索不到蓝牙了?

app_db_init_func 中对应的函数有问题, 一旦定义了, 这个函数必须有且正确

问题十四：在 app_task_handle.h 中不用添加头文件,把以下添加到工程头文件中即可

```
#if (BLE_PROX_REPORTER)
#include "app_proxr.h"
#include "app_proxr_task.h"
#endif
```

问题十五：锁屏、接电话、挂电话的 HID 值

问题十六：在蓝牙键盘中，裁减电池电量服务，不能正常工作，连接则马上断开

在 void prepare_bonding_info(struct bonding_info_ *inf)函数中，屏蔽掉下面的部分就可以了

```
// Read Notifications' status
// Battery Level
// handle = 0;
// uuid = ATT_CHAR_BATTERY_LEVEL;
// length = 0; // read
// if (process_value_of_ccc_uuid((uint16_t *)&handle, (uint8_t *)&uuid, &length,
//     inf->info != ((* (uint16_t *)value & 0x01) ? BASS_CCC_MASK : 0);
// // else default setting is off
```

问题十七：金刚狼老是要断开，是因为程序声明外部变量的时候，有 bug，解决了

问题十八：需要调用任务如：APP_LED_TASK，需要包含头文件#include "app_task.h"

问题十九：串口用了 P05，Flash 也用了 P05，所以初始化 SPI 的时候，确保引脚没有初始化成串口

spi_read_flash_jedec_id spi_flash_wait_till_ready spi_flash_read_status_reg

问题二十：初始化中断的时候，立刻要进入一次，解决的方法

先初始化中断，初始化引脚

```
//rom patch
patch_func();

#if KEY_INT_TASK
KEY_GPIO_init();
#endif //KEY_INT_TASK

//Init pads
set_pad_functions();
```

问题二十一：在 keyboard 中添加了 stream 总是会弹出调试出错，并且没有蓝牙信息，肯定是 app_stream_create_db();函数有问题，调试的时候，添加了这两个断点，就能有蓝牙发出，并可以调试

```
1136 switch (app_env.next_prf_init)
1137 {
1138
1139     //=====
1140     #if (BLE_STREAMDATA_DEVICE)
1141     case (APP_STREAM_TASK):
1142     {
1143         // Add proxr Service in the DB
1144         app_stream_create_db();
1145     } break;
1146     #endif
```

问题二十二：信号出来不稳定，连接不容易连接上，按键控制蓝牙广播的开关

串口服务

注意添加以下两个函数

```
#if BLE_BATT_SERVER
    APP_BATT_TIMER,
    APP_BATT_ALERT_TIMER,
#endif //BLE_BATT_SERVER

#if (BLE_SPS_SERVER)
    {SPS_SERVER_CREATE_DB_CFM, (ke_msg_func_t)app_sps_create_db_cfm_handler},
    {SPS_SERVER_ENABLE_CFM, (ke_msg_func_t)app_sps_server_enable_cfm_handler}
#endif //BLE_SPS_SERVER
```

GAP central and peripheral roles.

执行流程

app_set_dev_config_complete_func---> app_db_init() ---> app_db_init_func(); --->
app_sps_create_db(); ---> app_db_init_complete_func(); ---> app_adv_start();

KEYBOARD 简化

可以删掉所有 HAS_EEPROM HAS_MULTI_BOND HAS_VIRTUAL_WHITE_LIST
HAS_WHITE_LIST HAS_KEYBOARD_LEDS app_kbd_scan_mutirix()
HAS_MITM app_kbd.h 中所有 HAS 为 0 的都可以删了

删 is_bonded eeprom_is_read bond_info 等等变量相关的删了 dbg_puts dbg_printf(有些是多排)

删完了 white_list 内容，删头文件 #include "app_white_list.h" #include "app_multi_bond.h"
#include "app_sec.h" #include "app_console.h" 除 IIC 中的#include "i2c_eeprom.h"
全局编译，没有用到的函数全部删了

```
unsigned char plc_eu_backup[PLC_EU_BACKUP_BUF/8]
__attribute__((section("NO_INIT"), zero_init));
```

变量属性修饰符 `__attribute__((section("name"),zero_init))` 用于将变量强制定义到 `name` 属性数据节中，`zero_init` 表示将未初始化的变量放到 `ZI` 数据节中。因为“`NO_INIT`”这显性命名的自定义节，具有 `UNINIT` 属性。

正误对比

正确的，进入这个函数过后，直接就完成 `app_module_init_cmp_evt_handler`
错误的，一直完成不了 `app_db_init` 返回为 0
能同时工作更改了两个地方

app_db_init_func 中添加了

这两个,把 HID 放在前面,才能仿真,否则会出现 debug 错误

DB 初始化完成过后，进入 `app_db_init_complete_func()` --- `app_state_update(NO_EVENT)`;

初始化为空闲状态

广播的 FSM 状态机：原始的 HID 广播流程

IDLE_ST(NO_EVENT) --- 很长时间

ADVERTISE_ST (CONN_REQ_EVT)

CONNECTION_IN_PROGRESS_ST (CONN_CMP_EVT)

CONNECTED_ST (DISCONN_EVT)

。。。。从此以后，一直循环后面 3 个，LED 指示灯闪烁大约 8 次过后，广播信号停止，

`start_adv_undirected` 无定向广播函数中 START ADV MSG

`APP_START_ADV_MSG` 对应的消息处理函数中，开始 HID ADV TIMER

`APP_HID_ADV_TIMER` 对应的消息处理函数中会

`app_state_update(ADV_TIMER_EXP_EVT)`;

工程搭建

第一步：键盘中测试空中升级

在 `periph_setup` 中初始化 SPI 引脚，注意其他引脚不要冲突

```
RESERVE_GPIO( SPI_CLK, GPIO_PORT_0, GPIO_PIN_0, PID_SPI_CLK);  
RESERVE_GPIO( SPI_DO, GPIO_PORT_0, GPIO_PIN_6, PID_SPI_DO);  
RESERVE_GPIO( SPI_DI, GPIO_PORT_0, GPIO_PIN_5, PID_SPI_DI);  
RESERVE_GPIO( SPI_EN, GPIO_PORT_0, GPIO_PIN_3, PID_SPI_EN);
```

```
GPIO_ConfigurePin( GPIO_PORT_0, GPIO_PIN_3, OUTPUT, PID_SPI_EN, true );  
GPIO_ConfigurePin( GPIO_PORT_0, GPIO_PIN_0, OUTPUT, PID_SPI_CLK, false );  
GPIO_ConfigurePin( GPIO_PORT_0, GPIO_PIN_6, OUTPUT, PID_SPI_DO, false );  
GPIO_ConfigurePin( GPIO_PORT_0, GPIO_PIN_5, INPUT, PID_SPI_DI, false );
```

在 `driver` 中添加 `spi_flash.c` 和 `spi.c`

在 `da14580_config.h` 中屏蔽着两句话


```

#ifdef CFG_PRF_SPOTAR
// #define CFG_SPOTAR_SPI_DISABLE //SPI support is disabled
// #define CFG_SPOTAR_UPDATE_DISABLE //SUOTA is not included
#define CFG_APP_SPOTAR 1
#define SPOTAR_PATCH_AREA 1 //Placed in the RetRAM when SPOT
#endif

```

Periph_setup.h 中加入 SPI 相关定义

```

// SPI Flash Manufacturer and ID
#define W25X10CL_MANF_DEV_ID (0xEF10) // W25X10CL Manufacturer and ID
#define W25X20CL_MANF_DEV_ID (0xEF11) // W25X10CL Manufacturer and ID

// SPI Flash options
#define W25X10CL_SIZE 131072 // SPI Flash memory size in bytes
#define W25X20CL_SIZE 262144 // SPI Flash memory size in bytes
#define W25X10CL_PAGE 256 // SPI Flash memory page size in bytes
#define W25X20CL_PAGE 256 // SPI Flash memory page size in bytes
#define SPI_FLASH_DEFAULT_SIZE 131072 // SPI Flash memory size in bytes
#define SPI_FLASH_DEFAULT_PAGE 256
#define SPI_SECTOR_SIZE 4096

```

升级过后，信号发射不出来，需要断开 SPI 的引脚，再下载程序，才能工作。。。而 5.0.4 却直接就能行。

第二步：测试键盘程序的工作稳定性，对比两个版本

老版本升级都发送步出来

第三步：添加按键中断控制音量

#define MATRIX_SETUP (9)

更改键值

```

//----音量加 home 音量减
//-----上一曲 播放/暂停 下一曲
{ 0xf406, 0xf415, 0xf407, 0x0000, }, // 0 (5)
{ 0xf401, 0xf404, 0xf400, 0x0000, }, // 1 (6)
{ 0x0030, 0x0037, 0x0036, 0x0010 } // 2 (7)

```

添加中断函数

添加按键值发送函数，注意，要在 app_kbd.h 中添加

int kbd_process_keycode(struct keycode_buffer_tag *keycode_idx)

并且，在这个函数前面的 struct 删掉

```
static int kbd_process_keycode(struct keycode_buffer_tag *keycode_idx)
```

注意初始化中断要在初始化键盘之前

```

3 #if KEY_INT_TASK
KEY_GPIO_init();
#endif //KEY_INT_TASK

//Init pads
set_pad_functions();

```

否则就会进入下面这个函数

```

kbd_rep_info* prepare_extended_report(kbd_rep_info *last)
{
    kbd_rep_info *p_report;

    // add one <type> report
    p_report = kbd_pull_from_list(&kbd_free_list);
    ASSERT_WARNING(p_report);
}

```

注意，引脚和其他有没有冲突，一定要检查，IIC 使用了 07 扫描 pin 脚使用了 2.7
 注意，一定要连接上了之后才能按按键，否则会出错

第三步：键盘上移植 stream

按照最简单的方法移植过后，运行不了，可以在下面这个地方加一个断点就可以运行了

```

1185 // Check if another should be added in the database
1186 if (app_env.next_prf_init < APP_PRF_LIST_STOP)
1187 {
1188     switch (app_env.next_prf_init)
1189     {
1190         //-----
1191         #if (BLE_STREAMDATA_DEVICE)
1192         case (APP_STREAM_TASK):
1193         {

```

把 stream 服务函数移到最后，就能有信号，但是安卓不能控制音量（应该是相互之间影响了）

```

//=====用了以下几个服务
#if (BLE_DIS_SERVER)
    APP_DIS_TASK,
#endif // (BLE_DIS_SERVER)

#if (BLE_HID_DEVICE)
    APP_HOGPD_TASK,
#endif // (BLE_APP_KEYBOARD)

#if (BLE_SPOTA_RECEIVER)
    APP_SPOTAR_TASK,
#endif // (BLE_SPOTA_RECEIVER)

#if (BLE_BATT_SERVER)
    APP_BASS_TASK,
#endif // (BLE_BATT_SERVER)

#if (BLE_STREAMDATA_DEVICE)
    APP_STREAM_TASK,
#endif // (BLE_STREAMDATA_DEVICE)

```

调试时，CPU 老是停止运行：添加了 stream 服务才有的现象
 感觉在这里出现了错误，把 total_size 改小点，就可以运行了，哈哈

```

//-----
// Add Service in the Database
//-----
status = attmdb_add_service(&hogpd_env.shdl[i], dest_id, nb_att, 0, 0,
                           total_size);

```

根据 total_size 的来源，所以选择了在头文件中做如下修改，total_size 则减小了，在 hogdp.h 中

```

// Maximal number of HIDS that can be added in the US
#ifndef USE_ONE_HIDS_INSTANCE
#define HOGPD_NB_HIDS_INST_MAX (2)
#else
#define HOGPD_NB_HIDS_INST_MAX (1)
#endif

// Maximal number of Report Char. that can be added in
#define HOGPD_NB_REPORT_INST_MAX (3)

// Maximal length of Report Char. Value
#define HOGPD_REPORT_MAX_LEN (20)
// Maximal length of Report Map Char. Value
#define HOGPD_REPORT_MAP_MAX_LEN (100)

// Length of Boot Report Char. Value Maximal Length
#define HOGPD_BOOT_REPORT_MAX_LEN (8)

```

这个不能是 100, 200 才能行, 最终选择 120

```

// Maximal length of Report Map Char. Value
#define HOGPD_REPORT_MAP_MAX_LEN (200)

```

第四步：确定连接成功之后，才进行按键键值的上报

请求连接会进入这个函数，start_reporting

```

void app_param_update_func(void)
{
    app_kbd_start_reporting(); // start sending notifications
    // Clear all buffers if reporting "old" keys is not wanted
    if (!HAS_REPORT_HISTORY)

```

发送键值前，添加这句话，就不会死机

```

if (kbd_reports_en != REPORTS_PAUSED) //REPORTS_ENABLED
{
    kbd_keycode_buf2.flags=0x10;
    kbd_keycode_buf2.output=0x00;
    kbd_keycode_buf2.input=0x00;
}

```

第五步：简化

把 app_kbd.h 中没用的定义屏蔽，删除

删除 HAS_MULTI_BOND HAS_EEPROM

MBOND_LOAD_INFO_AT_INIT bond_info(貌似这个不能删，删了就进不了系统了)

删除 start_adv_directed

第四步：停止广播和开启广播的控制（看状态机）

```
GAPM_CMP_EVT --->
gapm_cmp_evt_handler --->
app_adv_undirect_complete(param->status)
```

还是尝试透传吧

①添加头文件和 profile app driver 中的文件

```
.. \.. \src\ip\ble\hl\src\profiles\sps\sps_server
.. \.. \src\ip\ble\hl\src\profiles\sps
.. \.. \src\modules\app\src\app_profiles\sps
.. \.. \src\modules\app\src\app_project\sps
.. \.. \src\modules\app\src\app_project\sps\uart
.. \.. \src\modules\app\src\app_utils\app_stream_queue
```

②#define CFG_PRF_SPS_SERVER

```
#if (BLE_SPS_SERVER)
    APP_SPS_TASK,
#endif //(BLE_SPS_SERVER)
```

```
app_sps_enable();
```

```
#include "app_sps_uart.h"
#include "app_sps_proj.h"
#include "sps_server_task.h"
#include "sps_server.h"
#include "app_sps_ble.h"
#include "app_sps_proj_task.h"
```

```
#if (BLE_SPS_SERVER)
    {SPS_SERVER_CREATE_DB_CFM,
    (ke_msg_func_t)app_sps_create_db_cfm_handler},
    {SPS_SERVER_ENABLE_CFM,
    (ke_msg_func_t)app_sps_server_enable_cfm_handler},
#endif //BLE_SPS_SERVER
```

```
屏蔽 app_connect{
```

```
屏蔽这个，不知道有什么用，但是屏蔽了不影响// override_ble_xoff();
```

把所有工程文件.c 换了。就 OK 了

```
static inline bool app_async_proc(void)
{
    bool ret = false;
    #if (STREAMDATA_QUEUE)
        ble_data_poll();
    #endif
}
```

Da14580_config.h 中一定要定义

```
#define CFG_PRF_SPS_SERVER
```

```
#define CFG_STREAMDATA_QUEUE
```

gattc_write_cmd_ind_handler---- 接收到 sps 上传的数据到这里

```
UART_Handler    RECEIVED_AVAILABLE
```

```
uart_sps_timeout_data_avail_isr(void)
```

```
uart_push(rx_read_pointer, size, rx_state_ptr);    //====发送函数，在 uart_rx_callback(uint8_t res)中
```

SPS 接收过程

```
gattc_write_cmd_ind_handler    //进入 task，在 sps_sever_task 中，和 stream 相同
```

```
ble_push((uint8_t*)&(param->value[0]), param->length, 0);    //进入这个函数
```

```
uart_tx_callback(UART_STATUS_OK);//进入回调函数
```

```
uart_pull(tx_write_pointer, TX_CALLBACK_SIZE, &tx_state_ptr);//获取到响应的数值
```

```
uart_pull(tx_write_pointer, TX_CALLBACK_SIZE, &tx_state_ptr)    //---- 接收函数，在
uart_tx_callback(uint8_t res)中
```

```
callback(UART_STATUS_OK);
```

```
void uart_rx_callback(uint8_t res)
```

```
void uart_tx_callback(uint8_t res)
```

```
uart_sps_write(tx_write_pointer, size, &tx_state_ptr, &uart_tx_callback);
```


基于 SPS 搭建工程

简化键盘扫描

```
//删了以下函数，有关于扫描
fsm_scan_update//删了
app_kbd_scan_matrix
kbd_process_scandata
app_kbd_enable_scanning
app_kbd_enable_wakeup_irq
app_kbd_enable_delayed_scanning
app_kbd_reinit_matrix
kbd_membrane_output_sleep

app_kbd_update_status

app_kbd_start_scanning
kbd_start_sw_scanning
update_scan_times

app_kbd_disable_scanning
kbd_membrane_output_wakeup
```

简化串口发送

app_console.h;//常用函数，可以直接删了，包括其头文件

问题

删除 HID 的 batt 服务总是出错，一定要把上面那一句话也给屏蔽了

```
// value = process_attribute(inf, ATT_CHAR_BATTERY_LEVEL, 2, BASS_CCC_MASK, BASS_
// if (value)
//     bass_env.features[0] |= BASS_FLAG_NTF_CFG_BIT;
//
```

广播流程

```
start_adv_undirected
app_start_adv_msg_handler --APP_HID_ADV_TIMER
app_state_update(ADV_TIMER_EXP_EVT);
app_adv_stop
```

```
prepare_bonding_info
process_value_of_ccc_uuid
ASSERT_WARNING
```

删除电量服务还要删除这句话

```
// handle = 0;
// uuid = ATT_CHAR_BATTERY_LEVEL;
// length = 0; // read
// if (process_value_of_ccc_uuid((uint16_t *)&handle, (uint8_t *)&uuid, &length, {
//     inf->info |= ((*uint16_t *)value & 0x01) ? BASS_CCC_MASK : 0);
// // else default setting is off
// Boot reset
```

LED 指示优先级

充电状态

正在配对状态

消息状态

电量过低状态（配对成功，3s 快闪一次蓝色）

HID 报告描述

```
kbd_keymap
modify_kbd_keyreport pReportInfo->pBuf[i+2] = kbd_keymap[fn_mod][output][input]
kbd_process_keycode case 0xF4
```

屏蔽空中升级

工程头文件中加入如下：

```
#if BLE_SPS_SERVER
#include "app_sps_uart.h"
```

```
#include "sps_server_task.h"
#include "app_sps_proj_task.h"
#endif
```

暂时屏蔽了下面这几句话

```
    ; SPOTAR
    ;#if (SPOTAR_PATCH_AREA == 0)
    ;app_spotar.o {spotar_patch_area}          ; Placed in the RetRAM when SPOTAR_PATCH_SYSRAM is 0
    ;#endif
```

苹果 6 要开放这句话

```
...
//#ifdef MITM_ON
app_dis_enable_prf_sec(param->conhdl, PERM(SVC, AUTH));
//#else
//# ifdef EEPROM_ON
app_dis_enable_prf_sec(param->conhdl, PERM(SVC, UNAUTH));
//# else
app_dis_enable_prf_sec(param->conhdl, PERM(SVC, ENABLE));
//# endif
//endif
```

或者这句话

```
    // Fill in the parameter structure
    req->conhdl = app_env.conhdl;
//#ifdef MITM_ON
//    req->sec_lvl = PERM(SVC, AUTH);
//#else
//# ifdef EEPROM_ON
//    req->sec_lvl = PERM(SVC, UNAUTH);
//# else
//    req->sec_lvl = PERM(SVC, ENABLE);
//# endif
//endif
```

电池电压曲线

最高电压 4.2V

低于 3.6V 时提示低电量报警

由于引脚也会分电流走，所以电阻分压有误差，采用测量的方式，直接读取低电压和满电压。

再次简化

dbg_puts 删了

再次试验 SPS 接收

①在 `periph_init` 中一定要添加，添加过后可以接收两次，哪里还有问题

`uart_sps_init(UART_SPS_BAUDRATE, 3);` //exact baud rate defined in `uart.h`

替换了主函数过后，要更改 `DA14580_config.h` 中的配置，才能有蓝牙出来

相比之下，还有主函数不一样

把下面这句话屏蔽就好了，哈哈

```

//      //if there is data available, send data over uart
//      if(size > 0) {
//          uart_sps_write(tx_write_pointer, size, &tx_state_ptr, &uart_tx_callback);
//          return;
//      }
//      //if there is no data but only flow control just send flow control to UART
//      else if(tx_state_ptr == UART_XOFF || tx_state_ptr == UART_XON){
//          uart_sps_write(0, 0, &tx_state_ptr, &uart_tx_callback);
//          return;
//      }

```

低功耗

flash

Standby Current	ICC1	/CS = VCC, VIN = GND or VCC	25	50	μA
Power-down Current	ICC2	/CS = VCC, VIN = GND or VCC	<1	10	μA

```
button=0;//i2c_eeprom_read_byte(0xaa);
```

```

#define _LED_RED           //i2c_eeprom_write_byte(0x80,0x06)
#define _LED_YELLOW        //i2c_eeprom_write_byte(0x80,0x01);
#define _LED_GREEN         //i2c_eeprom_write_byte(0x80,0x05);
#define _LED_BLUE          //i2c_eeprom_write_byte(0x80,0x03);
#define _LED_PURPLE        //i2c_eeprom_write_byte(0x80,0x02);
#define _LED_WHITE         //i2c_eeprom_write_byte(0x80,0x00);
#define _LED_BLACK         //i2c_eeprom_write_byte(0x80,0x07);

```

```

// motion_init();
// #if KEY_INT_TASK
// Touch_init();
// #endif

```

```

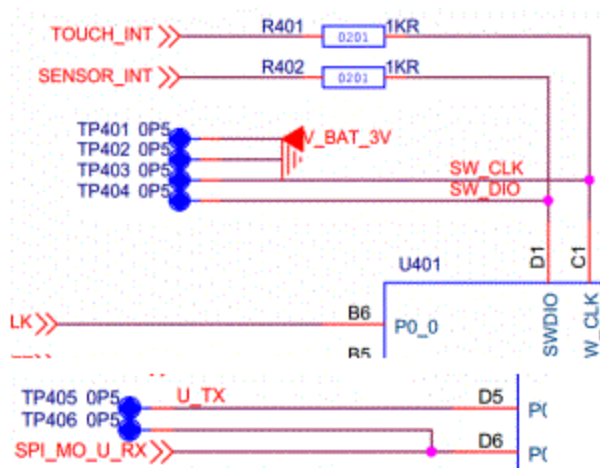
spi_flash_set_write_enable();
spi_flash_power_down();
i2c_eeprom_init(0x68, I2C_STANDARD, I2C_7BIT_ADDR, I2C_1BYTE_ADDR);
i2c_eeprom_write_byte(107,0x40);
i2c_eeprom_init(0x37, I2C_STANDARD, I2C_7BIT_ADDR, I2C_1BYTE_ADDR);
i2c_eeprom_write_byte(0x86,7);

```

关闭以上过后，功耗 66

中断的形式触摸

中断引脚 SW_CLK ---- p1.4



用串口调试 UART --- P0.4

有时候控制失灵，是因为从系统连接，app 也连接上了，如果只是单纯从 app 连接则可以有时候还是键值不能控制

计步低功耗

第一，在初始化之前，要重新定义地址

第二，把#define MPU6500

还未尝试

mpu_lp_accel_mode(rate);---放在初始化最后

```
..... //陀螺仪采样率, 1KHz
MPU6050_WriteReg(MPU6050_RA_CONFIG, 0x06);
```

mpu_set_sample_rate --- 初始化中

MPU6555 原始数据读取

加速度的满量程是 2g

注意以下地方


```

        // (optional)
        uart_push("<DOWN2>", 7, 1);
    else
        Key_Do(Scan_Pre);
}

Key_Do(Scan_Pre); |
break;
default:
break;

```

继续简化

HAS_PASSCODE_TIMEOUT --- 删了
HOGDP --- 删了没用的
USE_PREF_CONN_PARAMS_ON --- 删了

配置触摸，一定要对应对的校验值

JTAG 当成普通 IO 口使用

By the way, Can I use those Pins (SWCLK and SWDIO) as UART_TX and UART_Rx?

Device:
DA14580
[reply](#)

Mon, 2016-09-19 16:00
MT_dialog Hi jiangnanzhi,

Yes you can use them as GPIO's when the SYS_CTRL_REG[DEBUGGER_ENABLED] bit is not set. Just configure them as you would do with other GPIOs in the periph_init() and you will be able to use them.

Thanks MT_dialog

计步低功耗

The MPU-6555 includes support for Automatic Activity Recognition (AAR™) on a wrist-worn device. It works in conjunction with the AAR™ library to detect walk, run, bike, stationary, and sleep. The AAR™ library achieves high detection accuracy and low power by using the gyro sensor in a smart duty cycle fashion. It is capable of identifying a new activity within 10sec of its transition. The AAR™ library offers a high accuracy pedometer that benefits from the contextual awareness of knowing which activities will require steps and which will not.

苹果消息通知服务 ANCS

app_connect_confirm

邮件回复：

在连接的时候，我并不知道是 APP 还是系统请求的连接，

重新开始

暂时屏蔽如下：

```
//#define CFG_PRF_DISS          1

//#define CFG_PRF_SPOTAR
//#ifndef CFG_PRF_SPOTAR
//      #define SPOTAR_PATCH_AREA    1      // Placed in the RetRAM when
SPOTAR_PATCH_AREA is 0 and in SYSRAM when 1
//      #define CFG_APP_SPOTAR      1
//#endif

//      #if (BLE_APP_PRESENT)
//      app_dis_enable_prf(app_env.conhdl);
//      #endif
```

还能裁减多少

```
#define DEVELOPMENT_DEBUG      0  少了 1k 空间，刚好够用
```

低功耗

异步唤醒：外部输入事件来唤醒定时器和正交解码器来唤醒

app_timer_set(APP_ADV_TIMER, TASK_APP, 1800); — 在 app_adv_func 函数中

```
int app_adv_timer_handler(ke_msg_id_t const msgid,
                        void const *param,
                        ke_task_id_t const dest_id,
                        ke_task_id_t const src_id)
{
    app_adv_stop();
}
```

```

    ke_timer_clear(APP_ADV_TIMER, TASK_APP);

    app_set_deep_sleep();

    // disable wakeup for BLE and timer events. Only external (GPIO) wakeup events can wakeup
    processor.
    app_ble_ext_wakeup_on();

    app_button_enable();

    return (KE_MSG_CONSUMED);
}

```

最新程序功耗高了 0.4mA，不知道为什么

```

    //charge status--ÖÐ¶ïÐÊ%ÊµÖ
    RESERVE_GPIO( GPIO, GPIO_PORT_0, GPIO_PIN_2, PID_GPIO );
    GPIO_ConfigurePin(GPIO_PORT_0, GPIO_PIN_2, INPUT_PULLDOWN, PID_GPIO, false);
//屏蔽以上这个终端，高 0.4mA
//      //key find interrupt
//  RESERVE_GPIO( GPIO, KEY_INT_PORT, KEY_INT_PIN, PID_GPIO );
//  GPIO_ConfigurePin(KEY_INT_PORT, KEY_INT_PIN, INPUT_PULLUP, PID_GPIO, false);
屏蔽以上这个终端，少 0.4mA
--不能理解

```

广播结束过后，调用 CPU_sleep,只有 100uA 电流

```

void cpu_sleep(void)
{
    ke_timer_clear(APP_SIGN_TIMER, TASK_APP);
    ke_timer_clear(APP_PAIR_TIMER, TASK_APP);
    ke_timer_clear(APP_STA_TIMER, TASK_APP);
    ke_timer_clear(APP_HID_ADV_TIMER, TASK_APP);
    touch_enable(0);
    step_enable(0);
    app_set_extended_sleep(); //½øÊË~Ãß
    app_ble_ext_wakeup_on(); //¹ø±ÖÀ¶ÑÀ
}

```