

**SDK-APP**

设计说明

最新版本：1.0

2017-06-11

# **目 录**

[1 SDK概述 1](#_Toc24057)

[1.1 SDK组成 1](#_Toc11385)

[1.2 Bootloader 1](#_Toc16003)

[1.3 SDK算法系统 1](#_Toc22968)

[2 APP系统概述 3](#_Toc30287)

[2.1 APP系统硬件驱动层 6](#_Toc28500)

[2.1.1 AFIO 6](#_Toc2971)

[2.1.1.1与AFIO有关的代码 6](#_Toc1464)

[2.1.1.2 AFIO功能的使用 6](#_Toc15289)

[2.1.1.3 AFIO功能的不使用 6](#_Toc14621)

[2.1.2 GPIO 7](#_Toc32108)

[2.2.2.1 GPIO有关的文件 7](#_Toc27389)

[2.2.2.2 GPIO的使用 7](#_Toc25343)

[2.1.3 SPI 8](#_Toc10314)

[2.1.3.1与spi相关代码 8](#_Toc16935)

[2.1.3.2 spi配置 8](#_Toc8774)

[2.1.4 Uart 9](#_Toc30711)

[2.1.4.1与uart相关代码 9](#_Toc22315)

[2.1.4.2 uart配置 9](#_Toc4458)

[2.1.4.3 uart接收 10](#_Toc12550)

[2.1.4.4 uart发送 10](#_Toc9557)

[2.1.5 EXTI 11](#_Toc21616)

[2.1.5.1与EXTI有关的代码 11](#_Toc21175)

[2.1.5.2 EXTI配置 11](#_Toc26122)

[2.1.6 Timer 12](#_Toc11973)

[2.1.6.1定时器相关代码 12](#_Toc1260)

[2.1.6.2 TIMER类型 12](#_Toc2086)

[2.1.6.3 GPTM配置 13](#_Toc26973)

[2.1.6.4 MCTM配置 14](#_Toc20791)

[2.1.7 ADC 15](#_Toc21144)

[2.1.7.1与adc有关的代码 15](#_Toc20822)

[2.1.7.2 adc配置 15](#_Toc873)

[2.2 UI和流程控制 16](#_Toc32122)

[2.3 config区 17](#_Toc19675)

[3 SDK和APP交互 18](#_Toc6825)

[3.1 系统调用API 20](#_Toc24414)

[3.2 SDK注册API到APP 20](#_Toc8461)

[3.3 SDK与APP之间传递信息 21](#_Toc7688)

[3.4 APP注册API到SDK 21](#_Toc19633)

[4 设备驱动标准接口的实现 24](#_Toc19672)

[4.1设备驱动框架 24](#_Toc28978)

[4.1.1驱动架构的实现 24](#_Toc31273)

[4.1.2上层访问 25](#_Toc4069)

[4.2 硬件设备 25](#_Toc17596)

# 

# 版本历史

|  |  |  |  |
| --- | --- | --- | --- |
| **日期** | **版本号** | **注释** | **作者** |
| 2017-06-11 | 1.0V | 建立 | 李永勇 |
|  |  |  |  |
|  |  |  |  |

# 1 SDK概述

## 1.1 SDK组成

从结构上看，SDK 由三大部分组成

APP系统

SDK算法系统

BootLoader

从软件架构上，有bootloader ， sdk算法系统， APP系统组成。

FLASH 空间看SDK

分为四部分：

Config

app系统

sdk算法系统

BootLoader

**BootLoader**  ： 0 ~ 0x2000 ROM : 8K

**Sdk 算法系统** ：0x2000 ~ 0x36c00 ROM : 211K RAM: 42K

**App系统**： 0x33400 ~ 0x3fbff ROM : 50K RAM: 6K

**Config 区**： 0x3f800 ~ 0x3fc00 ROM: 1K

Flash空间通icf文件分配，具体可查看



文件中如下描述是对APP系统ROM和RAM的分配

define symbol \_\_ICFEDIT\_region\_ROM\_start\_\_ = 0x00033400;

define symbol \_\_ICFEDIT\_region\_ROM\_end\_\_ = 0x0003FBFF;

define symbol \_\_ICFEDIT\_region\_RAM\_start\_\_ = 0x2000a800;

define symbol \_\_ICFEDIT\_region\_RAM\_end\_\_ = 0x2000BFFF;

ROM=0x0003FBFF - 0x00033400 = 50k

RAM=0x2000BFFF - 0x2000a800 =6k

两个地址相减即为空间大小，后续版本可能会随着需求不同APP系统ROM和RAM的分配进行调整。

## 1.2 Bootloader

在sdk中的作用：用来升级sdk算法系统和APP系统软件,config区。（非open）

## 1.3 SDK算法系统

在sdk中的作用：完成slam算法的主体功能。（非open）

Kernel

OS Lib层

IC硬件驱动（以及其框架）

sdk算法系统

外围硬件驱动（以及其框架）如陀螺仪

其他基础算法,比如陀螺仪校准，slip算法，地毯检测，码盘定位计算等等

处理各种系统外围事件，如cliff，bump

Rebound子系统

沿边子系统

地图标识

建图和地图搜索子系统

路径搜索

路径规划

路径规划和导航子系统

Path follow

行为调度

速度控制

姿态控制

系统事件控制

流程控制

# 2 APP系统概述

在sdk中的作用： 适应不同的方案，进行个性化设计， open 给客户进行二次开发

硬件驱动层

Charge充电模块

异常自检模块

APP系统

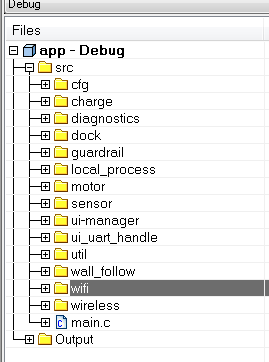
Dock 模块

Sensor处理模块

远程交互协议层

UI和流程控制

目前APP工程目录：



APP系统文件的内容介绍

|  |  |  |
| --- | --- | --- |
| **文件夹** | **源文件** | **文件主要代码内容** |
| cfg | afio\_cfg.c | 主要是配置引脚的复用功能（硬件驱动配置） |
| exti\_cfg.c | 配置睡眠时使用外部中断方式唤醒 |
| gpio\_cfg.c | gpio的配置 |
| spi\_cfg.c | 配置spi，用于nor flash和陀螺仪 |
| tm\_cfg.c | 配置定时器，用于输出pwm（pwm用于吸尘器、主、边刷和充电） |
| uart\_cfg.c | 配置串口通信 |
| charge | charge.c | 跟充电有关的函数和充电流程 |
| diagnostics | hardward\_check.c | 硬件检测函数 |
| dock | dock\_ir\_signal.c | 发送红外信号 |
| docking-new.c | 实现回座充电（自座）的各个行为 |
| docking-sensors.c | 对自座时用的红外检测函数进行封装 |
| guardrail | dock-avoid.c | 避座行为代码 |
| virtual-wall.c | 虚拟墙有关代码 |
| local\_process | act.c | 对按键行为的处理（物理和遥控按键） |
| display.c | 显示代码，这里用led灯来进行显示 |
| led\_drv.c | 数码管显示的驱动代码 |
| lib.c | 跟命令有关的函数（发送命令，向前、向后等命令） |
| local\_key\_check.c | 物理按键相关代码 |
| local\_process.c | 主线程和初始化函数 |
| remote.c | 遥控按键相关代码 |
| motor | motor.c | 电机的pwm动态更新（包括真空吸尘和主边刷） |
| robot\_brush.c | 主边刷pwm |
| robot\_suction.c | 真空吸尘pwm和对真空的电流状态的一些检测 |
| sensor | lt\_bump.c | 计算是否产生lt\_bump |
| sensor.c | 关于lt和cliff的采样和计算处理 |
| sensor\_comm.c | 申请lt、cliff采样中断，采样处理 |
| ui-manager | exception.c | 机器故障的检测和处理 |
| ir\_decode.c | 采集红外接收头，并对采集到的信号进行处理 |
| remote\_handle.c | 对红外遥控信号的解码 |
| robot-battery-monitor.c | 检测充电是否异常移除、电压是否过低、是否低电回座 |
| robot\_battery.c | 跟电源有关，如在dock还是jack上充电，电池的更新（电池监测） |
| song-player.c | 跟语音播报有关 |
| test\_cmd\_handle.c | 测试程序，测试专用 |
| ui-config.c | 设置UI结构体的默认值 |
| ui-manager.c | 管理UI结构体函数指针 |
| ui-msg-handler.c | SDK给APP发送消息 |
| ui\_uart\_handle | command\_handle.c | Uart通信命令包（发送应答包、发送系统状态） |
| uart\_protocol.c | Uart通信协议 |
| uart\_queue.c | Uart发送、接收数据缓冲区 |
| ui\_uart\_handle.c | Uart接收校验 |
| util | current.c | 对adc的数据进行过滤、判断电压、温度等 |
| wall\_follow | wall\_follow\_common.c | 沿边清扫算法 |
| wifi | SimSweep.c | 预约时间，保存、上传地图数据 |
| SimWifi.c | Wifi发送、接收数据处理 |
| uart\_wifi\_chek.c | Uart接收Wifi发送的命令 |
| wireless | ARF2496K.C | 该无线芯片的驱动代码 |
| Output | app.map | 编译的输出文件 |

UI系统用IAR IDE进行开发，工程里的使用改进过的启动文件（包括全局变量的初始化等等）， 不使用iar 工程默认的， 尽量不使用iar工程里的库文件，而使用系统调用。这样可以节省ui工程的代码空间。比如很多C函数，例如printf， memset， memcpy等等

提供给UI系统开发的IAR 工程，规定好代码空间以及ram空间，不得随便修改，否则可能出现数据覆盖错误等等情况。

创建线程

算法系统

Bl ui系统

Main函数

APP系统

系统初始化过程，bl main

说明：

APP系统初始化过程也是非常的重要，这个不用iar模式的系统文件，而是必须要经过改写后使用， 最后跳转到main函数， 之后main需要创建ui系统自身使用的线程，这个线程的优先级是系统分配的一个固定范围。

初始化完后，要返回到算法系统。 这个时候ui就可以工作了。线程在算法系统开始调度自动运行。



startup.a就是改写过iar的启动库。



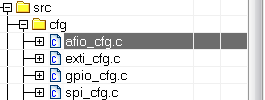
link文件也是需要写好，二次开发者不能随意改动里面的地址分配之类

## 2.1 APP系统硬件驱动层

### 2.1.1 AFIO

#### 2.1.1.1与AFIO有关的代码

在介绍外设配置之前，先介绍管脚复用，管脚复用有关的是afio\_cfg.c，



#### 2.1.1.2 AFIO功能的使用

里面把所有开放的外设功能都列了出来，需要使用哪个外设，在相应的外设里面填好管脚，如spi0对应的四个管脚

|  |
| --- |
| const afio\_init\_t spi0\_afio[] =  {  {GPIO\_C,AFIO\_PIN\_10},  {GPIO\_F,AFIO\_PIN\_13|AFIO\_PIN\_14|AFIO\_PIN\_15},  {IO\_NULL,IO\_NULL},  }; |

spi1对应的四个管脚

|  |
| --- |
| const afio\_init\_t spi1\_afio[] =  {  {GPIO\_F,AFIO\_PIN\_4 | AFIO\_PIN\_5 | AFIO\_PIN\_6 |AFIO\_PIN\_7 },//gyro  {IO\_NULL,IO\_NULL},  }; |

注意gpio名称不同时的填写格式。不然会出错。

**注意**：一个管脚多个功能只能填写到一个复用功能表里，如果同时在多个复用功能表里填写了同一个管脚，系统会出错。并且一旦在表中填上了该管脚复用的功能，该管脚就不能当做普通的io申请使用了。

#### 2.1.1**.3 AFIO功能的不使用**

如果该外设功能没有使用，又需要对应的管脚做普通IO口，只需将外设功能对应的复用功能表留空即可，如uart3

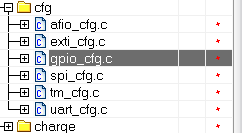
|  |
| --- |
| const afio\_init\_t uart3\_afio[] =  {  {IO\_NULL,IO\_NULL},  }; |

注意复用功能表填写和留空的格式。不然会出错。

**注意**:由于在复用功能表里填入管脚后，系统会自动设置相应管脚的状态。所以gpio初始化状态表里相应管脚的状态初始化，就会失效。换句话说就是gpio初始化状态表只对管脚做普通的IO时有效。

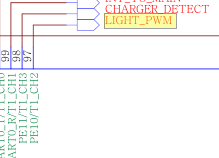
### 2.1.2 GPIO

#### 2.2.2**.1 GPIO有关的文件**



#### 2.2.2**.**2 **GPIO的使用**

GPIO是单片机最基本，也是最常用的功能。以我们light touch红外发射管使能管脚为例，通过使能管脚置1或0，就可以打开或关闭light touch红外发射管。我们可以通过两种方法配置GPIO，第一种是通过初始化管脚使用，例如light touch红外发射管使能管脚，红外发射使能管脚在原理图上用了PE10，如下图



在gpio\_init.h的gpio初始化状态表中对于PE10我们是这么初始化的，

|  |
| --- |
| m( GPIO\_DIR\_OUT , 4 , 10 , 1 ,GPIO\_PR\_UP )/\*light pwm \*/ |

这个宏有5个参数，第一个参数是管脚输入输出方向。第二个参数是管脚名，4代表E。第三个参数是管脚号。第四个参数是管脚宽度，表示连续的多少个管脚。第五个参数是上拉还是下拉或者禁用上下拉。

这样就把管脚初始化成上拉输出了。使用irq\_syscall.h中的gpio\_set\_value()函数就可以使管脚输出0或1了,如下代码

|  |
| --- |
| gpio\_set\_value(AM\_IO\_LIGHT\_TOUCH\_LED,SENSOR\_LED\_ON);  gpio\_set\_value(AM\_IO\_LIGHT\_TOUCH\_LED,SENSOR\_LED\_OFF); |

AM\_IOLINGHT\_TOUCH\_LED就是PE15宏定义。

第二种方法是通过irq\_syscall.h里的函数，向系统申请IO，如启动\暂停按键，见local\_key\_check.c

|  |
| --- |
| ret = gpio\_request\_one(key\_press\_state[i].am\_io, GPIO\_F\_DIR\_IN); |

这样就把按键所对应的管脚设为了输入状态，需要注意的是当申请时没有加入上下拉状态时，系统就会默认保持gpio初始化状态表中配置的上下拉状态。在后面的按键扫描中就可以使用gpio\_get\_value()函数读取按键的值,如下代码

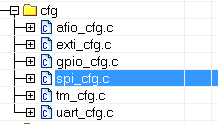
|  |
| --- |
| gpio\_state = gpio\_get\_value(key\_press\_state[i].am\_io); |

gpio\_init.h中的irq\_syscall.h中其他函数的用途可具体查看注释说明在此不再赘述。

如果要使用一个管脚，为了方便统一管理，建议用宏定义修改名称，把宏定义放入hal\_amicro\_gpio.h。

### 2.1.3 SPI

#### 2.1.3**.1与spi相关代码**



#### 2.1.3**.2 spi配置**

下面来讲讲如何配置并使用spi，首先在管脚复用功能表里填上对应管脚，以spi1为例

|  |
| --- |
| const afio\_init\_t spi1\_afio[] =  {  {GPIO\_F,AFIO\_PIN\_4 | AFIO\_PIN\_5 | AFIO\_PIN\_6 |AFIO\_PIN\_7 },//gyro  {IO\_NULL,IO\_NULL},  }; |

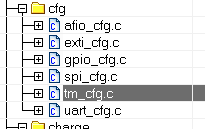
这样我们就可以对spi1进行配置了，在spi\_cfg.c中就是对spi0和spi1配置的代码，如下

|  |
| --- |
| SPI\_GYRO\_ID = open(DEV\_SPI,SPI\_CH1); //spi1，目前陀螺仪使用这一路  SPI\_InitStructure.SpiInit.SPI\_Mode = SPI\_MASTER;  SPI\_InitStructure.SpiInit.SPI\_FIFO = SPI\_FIFO\_DISABLE;//SPI\_FIFO\_ENABLE;//  SPI\_InitStructure.SpiInit.SPI\_DataLength = SPI\_DATALENGTH\_8; SPI\_InitStructure.SpiInit.SPI\_SELMode = SPI\_SEL\_SOFTWARE;//SPI\_SEL\_HARDWARE; SPI\_InitStructure.SpiInit.SPI\_SELPolarity = SPI\_SELPOLARITY\_LOW;  SPI\_InitStructure.SpiInit.SPI\_CPOL = SPI\_CPOL\_LOW;//SPI\_CPOL\_HIGH;  SPI\_InitStructure.SpiInit.SPI\_CPHA = SPI\_CPHA\_FIRST;//SPI\_CPHA\_SECOND;  SPI\_InitStructure.SpiInit.SPI\_FirstBit = SPI\_FIRSTBIT\_MSB;  SPI\_InitStructure.SpiInit.SPI\_RxFIFOTriggerLevel = 1;  SPI\_InitStructure.SpiInit.SPI\_TxFIFOTriggerLevel = 4;  SPI\_InitStructure.SpiInit.SPI\_ClockPrescaler = SPI\_NOR\_SPEED; //36 1M  SPI\_InitStructure.irqMode = 0;  SPI\_InitStructure.DmaMode = 0;  ioctl(SPI\_GYRO\_ID,SPI\_INIT,&SPI\_InitStructure);  ioctl(SPI\_GYRO\_ID,SPI\_SET\_GYRO\_ID,&SPI\_GYRO\_ID);//配置陀螺仪ID号 |

相应的spi配置参数可到AM380x\_spi.h查看。但是目前spi的自由度不大，仅开放了NOR flash有关的三个函数，分别是读、写和擦除NOR flash。具体使用和相关的注意事项请看syscall\_spi\_nor\_flash\_api.h中的注释说明.

### 2.1.4 Uart

#### 2.1.**4.1与uart相关代码**



#### 2.1.**4.**2 **uart配置**

以uart0为例先在复用功能表里填入管脚，如下

|  |
| --- |
| const afio\_init\_t uart0\_afio[] =  {  {GPIO\_D,AFIO\_PIN\_12|AFIO\_PIN\_13}, //wifi  {IO\_NULL,IO\_NULL},  }; |

然后就可以进行uart0的配置了，与uart有关的配置在uart\_cfg.c中，其中uart0配置代码如下

|  |
| --- |
| uart\_wifi = open(DEV\_UART,USART\_ID0);  UserUartInit.UartInit.USART\_BaudRate = 38400;  UserUartInit.UartInit.USART\_WordLength = USART\_WORDLENGTH\_8B;  UserUartInit.UartInit.USART\_StopBits = USART\_STOPBITS\_1;  UserUartInit.UartInit.USART\_Parity = USART\_PARITY\_NO;//USART\_PARITY\_ODD;  UserUartInit.UartInit.USART\_Mode = USART\_MODE\_NORMAL;  UserUartInit.UartInit.USART\_LSB\_MSB = USART\_LSB\_FIRST;  UserUartInit.UartMode.RxTxMode = IRQ\_MODE;//CPU\_MODE;//02:irq 0x1;//dma  UserUartInit.UartMode.irqMode = USART\_IER\_RDAIE;//0;//USART\_IER\_RDAIE;//no  UserUartInit.UartMode.DmaMode = 0;//DMA\_RX\_MODE;//1:rx 0:tx,rx 2:tx UserUartInit.UartMode.RxTxEna = UART\_RXTX;//tx,rx  ioctl(uart\_wifi,UART\_INIT, &UserUartInit);  request\_irq(uart\_wifi, (long)wifi\_uart\_handler,0); |

在当前代码中，uart0是配置成与wifi模块通讯。

|  |  |  |  |
| --- | --- | --- | --- |
| 功能名称 | 申请时的ID | 复用功能表中名称 | MCU中的名称 |
| 通用同步和异步收发器 | USART\_ID0 | uart0 | UART0 |
| 通用同步和异步收发器 | USART\_ID1 | uart1 | UART1 |
| 通用异步收发器 | UART\_ID0 | uart2 | UART2 |
| 通用异步收发器 | UART\_ID1 | uart3 | UART3 |

与uart配置有关的参数在am\_uart\_api.h中。

代码里的request\_irq(uart\_wifi, (long)wifi\_uart\_handler,0)是注册uart中断。wifi\_uart\_handler是中断处理函数中的回调函数的函数指针，其内容为

|  |
| --- |
| int wifi\_uart\_handler(int sr, int data)// sr为判断发送还是接收的标志  {  int ret = -1;  if(sr == USART\_IID\_RDA) //接收  {  IRQ\_Usart1RxData\_Process((u8)(data&0xff));  }  if(sr == USART\_IID\_THRE) //发送  {  if(uart\_txrx\_q\_empty(WIFI\_TX\_INDEX))  {  ret = -1;  }  else  {  ret = uart\_txrx\_q\_get(WIFI\_TX\_INDEX);//uart\_out\_3\_getc( ) ; //  }  return ret;  }  if(sr == USART\_IID\_CTI)  {  }  if(sr == USART\_IID\_RLS)  {  }  return 0;  //return ret;  } |

这个回调函数是处理接收和发送数据的。

#### 2.1.**4.3 uart接收**

当接收到数据时触发中断进入回调函数。函数入口的data是接收数据。

|  |
| --- |
| IRQ\_Usart1RxData\_Process((u8)(data&0xff)); |

这一句代码是对接收数据进行处理。

#### 2.1.**4.4 uart发送**

对于发送处理，在SimWifi.c中如下代码是发送函数

|  |
| --- |
| void Usart1\_SendHexString(uint8\_t \*pHexBuf,uint32\_t Long)  {  uint32\_t i;  for(i=0;i<Long;i++)  {  uart\_txrx\_q\_put(WIFI\_TX\_INDEX,pHexBuf[i]);  }  set\_wifi\_uart\_tx\_fifo\_empty\_irq();  //sys\_uart\_trig\_tx\_evnet(WIFI\_UART);  } |

其中set\_wifi\_uart\_tx\_fifo\_empty\_irq()是触发发送中断，set\_wifi\_uart\_tx\_fifo\_empty\_irq()的内容如下

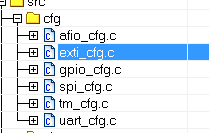
|  |
| --- |
| void set\_wifi\_uart\_tx\_fifo\_empty\_irq(void)  {  u8 trig = 1;  ioctl(uart\_wifi,UART\_TRIG\_TX,(void\*)&trig);//触发一个发送中断  } |

发送函数中前面先通过uart\_txrx\_q\_put函数把数据放入缓冲区，然后通过set\_wifi\_uart\_tx\_fifo\_empty\_irq()触发发送中断，这时候就会进入回调函数，同时sr判断为USART\_IID\_THRE，再判断缓冲区是不是为空，不为空就将缓冲区的数据读出来返回给中断去发送。

如果需要配置一个uart来使用，只需仿照上面代码配置即可。主要是中断处理函数里的回调函数要参考已写好的代码进行编写。

### 2.1.5 EXTI

#### 2.1**.5.1与EXTI有关的代码**



#### 2.1**.5.2** EXTI**配置**

目前外部中断仅开放中断唤醒功能，当机器处于睡眠状态时外部中断功能有效，这时候触发外部中断即可唤醒机器，但是当机器处于正常运行状态时外部中断功能就失效了。

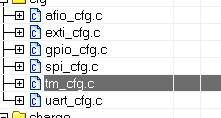
以暂停/启动键配置为例

|  |
| --- |
| exti\_user\_init.IntEn = ENABLE;  exti\_user\_init.Int\_type = EXTI\_POSITIVE\_EDGE;  exti\_user\_init.WakeupirqEn = ENABLE;  exti\_user\_init.Wakeup\_type = EXTI\_WAKEUP\_LOW\_LEVEL;  exti\_user\_init.Wakeup\_type = EXTI\_WAKEUP\_HIGH\_LEVEL;  id = open(DEV\_EXTI,AM\_IO\_MC\_KEY\_WAKE\_ON);  ioctl(id,EXTI\_INIT,&exti\_user\_init);  request\_irq(id,(long)wake\_handler,0);  wakeup\_id[i++] = id; |

**注意**：exti\_user\_init.Wakeup\_type表示唤醒信号，而wake\_handler表示中断处理函数，但目前外部中断是属于保留功能，所以这一处并没有什么实际作用。这样配置之后暂停/启动键就有了两个功能，当机器处于睡眠状态时，该按键是唤醒源，当机器正常运行时，外部中断功能就失效了，该按键就只是普通的按键。

### 2.1.6 Timer

#### 2.1.6.1定时器相关代码



#### 2.1.6.**2** TIMER**类型**

定时器一共有三个类型基本定时器BFTM、通用定时器GPTM、多功能定时器MCTM，其中BFTM目前没有开放。与定时器配置相关的代码在tm\_cfg.c中

GPTM、MCTM之间功能的区别如下

**GPTM**

●2个16位通用定时器；

●4 个独立通道

-PWM输出

-输出比较

-输入捕获

●外部触发输入。

**MCTM**

●2个16位向上、向下、向上\向下自动重载计数器

●16位可编程预分频器，允许将计数器时钟频率分割成1至65536之间的任意数值

●4个独立通道

-输入捕获功能

-比较匹配输出功能

-PWM波形产生（边缘对齐或中心对齐模式）

-单脉冲模式输出

-死区时间可编程的互补输出

●带有两路正交解码控制器的编码器接口

●支持三相电机控制和霍尔传感器接口

●刹车输入信号可以将定时器输出信号置于复位状态或者一个已知状态

所有的定时器时钟主频都是72M。

#### 2.1.6.3 **GPTM配置**

GPTM以主刷和边刷的PWM通道为例，

首先在管脚复用功能表里填入相应的管脚如下图

|  |
| --- |
| const afio\_init\_t gptm1\_afio[] =  {  {GPIO\_D,AFIO\_PIN\_14 | AFIO\_PIN\_15},//main/side  {IO\_NULL,IO\_NULL},  }; |

以主边刷的定时器配置为例，在tm\_cfg.c找到如下代码

|  |
| --- |
| gtm1\_id = open(DEV\_GPTM,GP\_TM1\_ID);  gftmInit.CounterReload = 1000;  gftmInit.Prescaler = 5; //lqw  gftmInit.CounterMode = GPTM\_CNT\_MODE\_UP;  gftmInit.PSCReloadTime = GPTM\_PSC\_RLD\_IMMEDIATE;  ioctl(gtm1\_id, TM\_GPTM\_BASE\_INIT, &gftmInit);  UserOutInit.OutInit.OutputMode = GPTM\_OM\_PWM1;  UserOutInit.OutInit.Control = GPTM\_CHCTL\_ENABLE; //enable chx  UserOutInit.OutInit.Polarity = GPTM\_CHP\_NONINVERTED;//h is active  UserOutInit.OutInit.Compare = 0; //占空比  UserOutInit.OutInit.AsymmetricCompare = 0;  UserOutInit.OutMode.irq = 0;//  UserOutInit.OutMode.dmaReqMode = 0;//dma request  for(int i = 2; i<4; i++)  {  UserOutInit.OutInit.Channel = (GPTM\_CH\_Enum)i;  ioctl(gtm1\_id,TM\_GPTM\_OUT\_MODE\_INIT, &UserOutInit);  } |

GP\_TM0\_ID代表GPTM0、GP\_TM1\_ID代表GPTM1、GP\_TM2\_ID代表GPTM2，这个PWM输出频率为72M/((1000+1)+(5+1))=11.988KHz，这段代码中同时配置通道2和通道3。

|  |  |  |
| --- | --- | --- |
| 定时器 | 申请时ID | 在MCU名称 |
| GPTM0 | GP\_TM0\_ID | T0 |
| GPTM1 | GP\_TM1\_ID | T1 |
| GPTM2 | GP\_TM2\_ID | T2 |

以下是边刷PWM更新处理，更新GPTM1通道3的PWM占空比

|  |
| --- |
| case SIDE\_BRUSH\_PWM\_CHANNEL:  id = gtm1\_id;  ioctl(id,GPTM\_CHAN3\_SET\_DUTY,(void \*)&Dutyfactor);  break; |

占空比取值0~1001 。

#### 2.1.6.**4** MCTM**配置**

MCTM以充电通道为例，同样在管脚复用功能表里填入相应的管脚如下图在tm\_cfg.c

|  |
| --- |
| const afio\_init\_t mctm1\_afio[] =  {  {GPIO\_F,AFIO\_PIN\_8},//charge  {IO\_NULL,IO\_NULL},  }; |

|  |
| --- |
| charge\_pwm\_id = open(DEV\_MCTM,MC\_TM2\_ID);//tm\_request(MCTM,MC\_TM2\_ID);  mctmInit.CounterReload = 720;  mctmInit.Prescaler = 0;  mctmInit.CounterMode = MCTM\_CNT\_MODE\_UP;  mctmInit.PSCReloadTime = MCTM\_PSC\_RLD\_IMMEDIATE;  ioctl(charge\_pwm\_id, TM\_MCTM\_BASE\_INIT, &mctmInit);  MUserOutInit.OutInit.OutputMode = MCTM\_OM\_PWM2;  MUserOutInit.OutInit.Control = MCTM\_CHCTL\_ENABLE;  MUserOutInit.OutInit.ControlN = MCTM\_CHCTL\_ENABLE;  MUserOutInit.OutInit.Polarity = MCTM\_CHP\_NONINVERTED;  MUserOutInit.OutInit.PolarityN = MCTM\_CHP\_NONINVERTED;  MUserOutInit.OutInit.IdleState = MCTM\_OIS\_LOW;  MUserOutInit.OutInit.IdleStateN = MCTM\_OIS\_HIGH;  MUserOutInit.OutInit.Compare = 0;  MUserOutInit.OutMode.irq = 0;  MUserOutInit.OutMode.dmaReqMode = 0;  MUserOutInit.OutInit.Channel = (MCTM\_CH\_Enum)1;  ioctl(charge\_pwm\_id, TM\_MCTM\_OUT\_MODE\_INIT, &MUserOutInit);  request\_irq(charge\_pwm\_id,(long)charger\_pwm\_handler,0); |

找到如下代码

MCTM占空比输出率与GPTM计算方式一样

|  |  |  |
| --- | --- | --- |
| 定时器 | 申请时ID | 在MCU名称 |
| MCTM0 | MC\_TM1\_ID | MT0 |
| MCTM1 | MC\_TM2\_ID | MT1 |

并且PWM更新方式一样。如下图

|  |
| --- |
| case CHARGER\_PWM\_CHANNEL:  id = charge\_pwm\_id;  //tm\_mctm\_output\_duty\_set(id,chan,Dutyfactor);  ioctl(id,MCTM\_CHAN1\_SET\_DUTY,(void \*)&Dutyfactor);  break; |

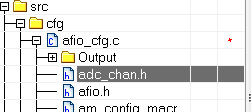
上面的代码配置中

|  |
| --- |
| request\_irq(charge\_pwm\_id,(long)charger\_pwm\_handler,0); |

是申请了一个中断，所有定时器都可以触发中断，但是这个中断的产生有可能会对系统产生一定的影响，所以这个功能需要斟酌使用。

### 2.1.7 ADC

#### **2.1.7.1与**adc**有关的代码**



#### 2.1.7.2 a**d**c配置

adc的配置相对没有那么复杂，首先在管脚复用功能表里填入相应的管脚如下图

|  |
| --- |
| const afio\_init\_t adc\_afio[] =  {  {GPIO\_A,AFIO\_PIN\_ALL},  {GPIO\_B,AFIO\_PIN\_0 | AFIO\_PIN\_1| AFIO\_PIN\_2 | AFIO\_PIN\_3 | AFIO\_PIN\_4},  {IO\_NULL,IO\_NULL},  }; |

上图代码把把当前机器需要用的adc通道复用了。然后通过函数获取对应通道的ADC值

|  |
| --- |
| EXPORT U16 get\_adc\_chan\_val(ADC\_ID\_E adc\_chan); |

其中adc通道列表在adc\_chan.h中，

|  |
| --- |
| #define ROBOT\_ADCCHANDEF(m) \  m(ADC\_CHAN\_CLIFF\_LEFT ,0 )\  m(ADC\_CHAN\_CLIFF\_FRONTLEFT ,1 )\  m(ADC\_CHAN\_CLIFF\_RIGHT ,2 )\  m(ADC\_CHAN\_REAR\_CLIFF\_LEFT ,3 )\  m(ADC\_CHAN\_REAR\_CLIFF\_RIGHT ,4 )\  m(ADC\_CHAN\_CLIFF\_FRONTRIGHT ,5 )\  m(ADC\_CHAN\_LT\_RIGHT ,6 )\  m(ADC\_CHAN\_LT\_CENTERLEFT ,7 )\  m(ADC\_CHAN\_LT\_CENTERRIGHT ,8 )\  m(ADC\_CHAN\_LT\_FRONTRIGHT ,9 )\  m(ADC\_CHAN\_LT\_FRONTLEFT ,10 )\  m(ADC\_CHAN\_LT\_LEFT ,11 )\  m(ADC\_CHAN\_VAC\_CURENT ,12 )\  m(ADC\_CHAN\_BATTERY\_THERMISTOR ,13 )\  m(ADC\_CHAN\_BATTERY\_CURRENT ,14 )\  m(ADC\_CHAN\_BATTERY\_VOLTAGE ,15 )\  m(ADC\_CHAN\_LEFT\_MOTOR\_CURRENT ,16 )\  m(ADC\_CHAN\_RIGHT\_MOTOR\_CURRENT ,17 )\  m(ADC\_CHAN\_MAIN\_BRUSH\_CURRENT ,18 )\  m(ADC\_CHAN\_SIDE\_BRUSH\_CURRENT ,19 )\  m(ADC\_CHAN\_BIN\_LEFT ,20 )\  m(ADC\_CHAN\_BIN\_RIGHT ,21 )\  m(HAL\_REV1 ,22 )\  m(HAL\_REV2 ,23 )\  m(HAL\_REV3 ,24 )\  m(HAL\_REV4 ,25 )\  #endif |

由于不同方案之间adc通道使用也不同，所以adc通道需要相应修改，如下代码是某款

案的adc通道使用。

|  |
| --- |
| #define ROBOT\_ADCCHANDEF(m) \  m(ADC\_CHAN\_CLIFF\_LEFT ,2 )\  m(ADC\_CHAN\_CLIFF\_FRONTLEFT ,1 )\  m(ADC\_CHAN\_CLIFF\_RIGHT ,3 )\  m(ADC\_CHAN\_REAR\_CLIFF\_LEFT ,0 )\  m(ADC\_CHAN\_REAR\_CLIFF\_RIGHT ,4 )\  m(ADC\_CHAN\_CLIFF\_FRONTRIGHT ,5 )\  m(ADC\_CHAN\_LT\_RIGHT ,7 )\  m(ADC\_CHAN\_LT\_CENTERLEFT ,11 )\  m(ADC\_CHAN\_LT\_CENTERRIGHT ,6 )\  m(ADC\_CHAN\_LT\_FRONTRIGHT ,9 )\  m(ADC\_CHAN\_LT\_FRONTLEFT ,8 )\  m(ADC\_CHAN\_LT\_LEFT ,10 )\  m(ADC\_CHAN\_VAC\_CURENT ,18 )\  m(ADC\_CHAN\_BATTERY\_THERMISTOR ,13 )\  m(ADC\_CHAN\_BATTERY\_CURRENT ,14 )\  m(ADC\_CHAN\_BATTERY\_VOLTAGE ,15 )\  m(ADC\_CHAN\_LEFT\_MOTOR\_CURRENT ,16 )\  m(ADC\_CHAN\_RIGHT\_MOTOR\_CURRENT ,17 )\  m(ADC\_CHAN\_MAIN\_BRUSH\_CURRENT ,12 )\  m(ADC\_CHAN\_SIDE\_BRUSH\_CURRENT ,19 )\  m(ADC\_CHAN\_BIN\_LEFT ,20 )\  m(ADC\_CHAN\_BIN\_RIGHT ,21 )\  m(HAL\_REV1 ,22 )\  m(HAL\_REV2 ,23 )\  m(HAL\_REV3 ,24 )\  m(HAL\_REV4 ,25 )\  #endif |

若是某个通道不用，又不想出现什么不必要的麻烦，在APIO里不要复用为ADC即可。

APP已经根据相应的功能命名好并且排好顺序，所以命名不能更改，顺序也不能更改，对应命名功能的通道不对，可以修改通道。

除了上面已经按照功能命名好的通道之外，HAL\_REV1~HAL\_REV4是可供用户扩展的用户可以修改成任何名字使用。

## 2.2 UI和流程控制

请参考程序。

Act\_ctrl

Key info

Test mode

……

Sdk info

Sensor info

Local info

Remote info

Main process

Ui display

Play song

monitor

……

外部事件或者info输入触发

app流程

## 2.3 config区

config区是 0x3f800 ~ 0x3fc00 的rom区间。

Config区，包括了4个 IO 配置表。 还有本地配置信息和sdk配置信息。下面就介绍一下本地配置和SDK配置信息.

Config区flash空间的组成：

localconfig

Sdkconfig

io\_init\_table

adc\_chan\_table

io\_table

afio\_table

本地配置：

typedef struct

{

cliff\_threshold\_config\_t cliff\_threshold;

lighttouch\_threshold\_config\_t lighttouch\_threshold;

remote\_ir\_code\_config\_t remote\_ir\_codes;

dock\_ir\_code\_config\_t dock\_ir\_codes;

**}ui\_local\_config\_t;**

**//cliff 阈值配置**

typedef struct

{

}cliff\_threshold\_config\_t;

**//light touch阈值配置**

typedef struct

{

}lighttouch\_threshold\_config\_t;

**/\* 遥控器的码值 \*/**

typedef struct

{

}remote\_ir\_code\_config\_t;

**/\* 自座器发出的码值 \*/**

typedef struct

{

}**dock\_ir\_code\_config\_t;**

**SDK系统配置：**

typedef struct

{

power\_config\_t power\_config;

hard\_structure\_config\_t hard\_structure;

speed\_config\_t speed\_config;

clean\_motor\_config\_t clean\_motor\_config;

charge\_info\_t charge\_info;

wf\_info\_t wf\_info;

play\_info\_t play\_info;

funtion\_info\_t funtion\_info;

}**ui\_config\_t;**

typedef struct

{

**}power\_config\_t;**

**/\* 物理结构上的一些信息 \*/**

typedef struct

{

**}hard\_structure\_config\_t;**

# 3 SDK和APP交互

固定地址调用main

固定地址跳转

BootLoader

SDK算法系统

APP系统

注册回调

系统调用

说明:

BootLoader ：是通过固定地址，直接跳转到算法系统，这个跳转是无返回的跳转。

SDK算法系统 ：首先通过固定地址，直接跳转到APP系统的main函数里，然后APP系统的main函数必须要返回的，一般APP系统创建线程后直接返回即可。

APP系统 ：main函数里要求创建线程。

Bootloader 和 算法系统之间一般来说无交互，因为到了算法系统后，BootLoader的使命已完成，其他ram控制等已释放。

SDK算法系统和APP系统的交互就比较多了，几乎是每时每刻的，首先ui系统通过系统调用方式或者函数指针方式调用算法系统的函数， 而算法系统也可以通过访问APP系统注册的指针函数来访问APP的函数。

Manager api

Access

App

SDK

Access

Offer

Offer

系统调用api

syscall\_fun

## 3.1 系统调用API

Sys Fun table

算法系统

cpu触发svc异常

app系统

SVC fun\_num

异常返回

Funx run

num

Num：funx

说明：

SDK算法系统提供一个syscall\_api.a 库文件以及.H 函数声明头文件给APP系统，APP系统即可使用SDK算法系统的函数。

所有的系统调用函数可以参考syscall\_api.h，中断函数里不能调用SDK算法系统的函数。

## 3.2 SDK注册API到APP

APP通过调用 init\_irq\_syscall()函数获取SDK中包含函数指针的结构体数据结构如下：

Irq\_syscall.h结构体中所有函数的注释，并且这些函数可以在APP代码中任意地方调用。

|  |
| --- |
| typedef struct  {  io\_request gpio\_request;  io\_free gpio\_free;  io\_request\_array gpio\_request\_array;  io\_free\_array gpio\_free\_array;  io\_set\_value gpio\_set\_value;  io\_get\_value gpio\_get\_value;  io\_to\_exti gpio\_to\_exti;  set\_afio gpio\_set\_afio;  io\_request\_one gpio\_request\_one;  remote\_val set\_remote\_val;  adc\_val get\_adc;  robot\_lt\_update\_t robot\_lt\_update;  printf\_t printk;  write\_t write;  read\_t read;  ioctl\_t ioctl;  runing\_mode\_t runing\_mode;  null\_fun\_t rev1;/\*保留扩展用\*/  null\_fun\_t rev2;/\*保留扩展用\*/  null\_fun\_t rev3;/\*保留扩展用\*/  null\_fun\_t rev4;/\*保留扩展用\*/  }syscall\_fun; |

## 3.3 SDK与APP之间传递信息

Sdk通过

const static ui\_manager\_t ui\_manager = {

.put\_massage = ui\_manager\_put\_massage,

}

这里的 ui\_manager\_put\_massage 函数发送系统的信息到APP上，这个信息可以参考ui-manager.h 的

typedef enum

{

} UiMsgType;

目前使用到的MSGTYPE， 可以参考ui-msg-handler.c。

APP通过函数void act\_command\_q(U32 ui\_cmd, U8 cmd\_state, void \*msg, U16 msg\_len)

发送命令给SDK，实现开始清扫、停止清扫、回座充电和局部清扫等功能。如开始清扫命令

act\_command\_q(CMD\_CLEAN, CMD\_RUN, NULL, 0);

相关命令控制字在ui\_commands.h中。

两者交互如下图所示

act\_command\_q函数发送命令

ui\_manager\_put\_massage函数发送信息

**SDK**

**APP**

## 3.4 APP注册API到SDK

前面两种方式都是APP 访问SDK的api，那么SDK 如何访问APP的api昵？

这里注册的api，一部分是系统需要应对某些场景是需要对应去修改机器的一些特性，比如在某些场合需要关闭LT\_bump、light touch和cliff等，在地毯的环境下需要调高cliff的阈值，主边刷、真空PWM等；一部分是系统负责控制什么时候去执行功能的开关，如充电的开始、结束等；其他一部分是提供给系统的判断条件，如碰撞检测、轮子抬起检测等；剩下一部分是SDK传递消息给APP有关的函数。

APP通过调用 register\_ui\_config(get\_ui\_manager())， 把ui\_manager\_t注册到typedef struct ui\_manager\_t

|  |
| --- |
| typedef struct ui\_manager\_t  {  //初始化函数，系统得到此结构后需要先调用此函数，目前暂时没有  ui\_manager\_init\_t init;  //关机时调用此函数，目前暂时没有  ui\_manager\_deinit\_t deinit;  //系统有消息需要通知UI时调用此函数  ui\_manager\_put\_massage\_t put\_massage;  //取得整个APP对SDK的配置，随时可调用。  ui\_manager\_get\_config\_t get\_config;  //系统有故障发生时，调用此函数。  ui\_manager\_put\_error\_info\_t put\_error\_info;  //清除error info  ui\_manager\_clear\_error\_info\_t clear\_error\_info;  //  ui\_manager\_put\_sensor\_data\_t put\_sensor\_data;  //  ui\_manager\_put\_real\_map\_point\_t put\_real\_map\_point;  // 停止播放音频  ui\_manager\_quit\_song\_t quit\_song;  //dock 提供给系统调用的函数  dock\_new\_init\_t dock\_new\_init;  //  dock\_new\_start\_t dock\_new\_start;  //  dock\_new\_end\_t dock\_new\_end;  //  set\_dock\_new\_end\_t set\_ dock\_new\_end;  //关闭lt ，cliff的采集  lt\_on\_off\_set\_t set\_lt\_enable;  cliff\_on\_off\_set\_t set\_cliff\_enable;  robot\_is\_cliff\_t robot\_is\_cliff;  //cliff , light touch  //这个计算结果，需要提供给系统用  robot\_signal\_distance\_t robot\_signal\_distance;  robot\_rear\_lt\_distance\_t robot\_rear\_lt\_distance;  //这个函数提供给系统调用  robot\_is\_lighttouch\_t robot\_is\_lighttouch;  //charging 提供给SDK调用充电相关的  charging\_init\_t charging\_init;  process\_charging\_t process\_charging;  quit\_charging\_t quit\_charging;  charging\_state\_t charging\_state;  //driver  sys\_tm\_cfg\_t sys\_tm\_cfg;  sys\_tm\_close\_t sys\_tm\_close;  sys\_set\_pwm\_out\_duty\_t sys\_set\_pwm\_out\_duty;  //  sys\_uart\_cfg\_t sys\_uart\_cfg;  sys\_uart\_close\_t sys\_uart\_close;  //  exit\_wakeup\_cfg\_t exit\_wakeup\_cfg;  wakeup\_pin\_cfg\_t wakeup\_pin\_cfg;  //  sys\_spi\_cfg\_t sys\_spi\_cfg;  sys\_spi\_close\_t sys\_spi\_close;  //  robot\_suction\_set\_t robot\_suction\_set;  robot\_sidebrush\_set\_t robot\_sidebrush\_set;  robot\_midbrush\_set\_t robot\_midbrush\_set;  //  dock\_avoid\_active\_t avoid\_active;  dock\_avoid\_signal\_quality\_t avoid\_signal\_quality;  dock\_binocular\_avoid\_signal\_quality\_t binocular\_avoid\_signal\_quality;  battery\_is\_low\_t battery\_is\_low;  virtual\_wall\_active\_t virtual\_wall\_active;  get\_bump\_mask\_t get\_bump\_state;  get\_wheelDrop\_state\_t get\_wheelDrop\_state;  turn\_off\_touch\_bump\_t turn\_off\_touch\_bump;  turn\_on\_touch\_bump\_t turn\_on\_touch\_bump;  touch\_bump\_run\_t touch\_bump\_run;  fun\_t rev23;  fun\_t rev24;  fun\_t rev26;  fun\_t rev27;  fun\_t rev28;  fun\_t rev29;  fun\_t rev30;  }ui\_manager\_t; |

# 4 设备驱动标准接口的实现

为了统一管理驱动层的访问接口，以及节省系统调用的接口数， 设备驱动层做了如下的框架支持。

## 4.1设备驱动框架

APP

Open，close，ioct

Read，write, register irq标准接口

Register按规范写法，接口函数到注册到platform层

Plalform

虚拟接口层

其他…

Timer

Spi

Uart

### 

### 4.1.1驱动架构的实现

以Uart为例：

|  |
| --- |
| static platform\_device device\_info =  {  .name = DEV\_UART,  .read = sys\_uart\_read,  .write = sys\_uart\_write,  .request = sys\_uart\_request,  .release = sys\_uart\_release,  .ioct = sys\_uart\_ioct,  .request\_irq = sys\_uart\_register\_irq\_handler,  };  static void module\_init(void)  {  register\_driver(&device\_info);  }  static struct module this\_module @".module\_init" =  {  .init = module\_init,  }; |

这里可以看出，驱动架构上支持open，read，write，close，ioct，request几个函数的规范调用

### 4.1.2上层访问

实现了上面的架构后，上层访问设备驱动就可以按照标准的接口来访问了。

算法系统层，提供所有open的设备驱动的设备号，数据结构等等。

比如： Fd = Open(name,xx,xx);

write(fd, buf, len);

具体可以参考APP 代码。

## 4.2 硬件设备

系统硬件做好如下的命名，通过下面名字来操作设备，设备就可以当做是文件设备的概念

|  |
| --- |
| typedef enum  {  DEV\_UART = 0, //串口驱动  DEV\_BFTM , //base timer 目前暂时不能使用  DEV\_GPTM , //通用timer， 总共有三路，每路有4个通道  DEV\_IR , //红外协议模块，目前暂时用不上  DEV\_EXTI , //外部中断  DEV\_RTC , //rtc模块  DEV\_WDG , //看门狗  DEV\_BAKUP, //RTC域备份区， 系统复位不会清除改变数据。  DEV\_SPI , //SPI 模块，目前陀螺仪固定用了一路  DEV\_I2C , //i2c模块  DEV\_IPC , //暂时没有用  DEV\_MCTM , //多功能TIMER，总共有2路，每路有4个通道  }DEVICE\_E; |

如open(DEV\_MCTM,MC\_TM2\_ID);

open(DEV\_SPI,SPI\_CH0);

具体每个模块的控制命令字等等可以查看am\_device.h 里的注释。