工程使用手册

目录

[1 文件结构 2](#_Toc3179)

[1.1 树状图 2](#_Toc12149)

[1.2 宏定义选择 3](#_Toc25817)

[1.3 程序入口 3](#_Toc18962)

[2 固件结构&下载 4](#_Toc10086)

[2.1 eeprom.dat结构 4](#_Toc20700)

[2.2固件下载 4](#_Toc5845)

[2.3 ram空间 4](#_Toc10701)

[3 eeprom 读写接口 7](#_Toc32368)

[4 BLE 相关 7](#_Toc25427)

[4.1 adv/scan 7](#_Toc10744)

[4.2 BLE GATT 9](#_Toc23825)

[5 2.4G收发说明 11](#_Toc14810)

[6 低功耗接口说明 13](#_Toc24497)

[6.1 lpm模式相关配置 13](#_Toc7751)

[6.2 lpm模式自唤醒 13](#_Toc25636)

[7 PWM模块 14](#_Toc6118)

[8 认证 15](#_Toc3585)

[9 动态加载 15](#_Toc21303)

[9.1动态加载简介 15](#_Toc14834)

[9.2 动态加载代码空间说明 16](#_Toc7132)

[10 ICE使用说明 17](#_Toc18494)

[11 ADC使用说明 17](#_Toc20326)

[11.1 ADC功能PIN脚 17](#_Toc19703)

[11.2 读取设置 18](#_Toc13146)

## 1 文件结构

### 1.1 树状图

以下树状图，对工程中关键的文件夹/文件进行说明。

├─doc

├─format：ROM中固化的常量、变量，不可更改；

├─format\_patch：定义RAM中常量、变量，用户根据工程需要自行添加和更改；

│ patch\_format.format：用户的变量申请可以放在此.format文件中

├─output：编译生成的文件；

│ eeprom.dat：最终下载到eeprom/flash中的16进制文件；

│ memmap.format：编译后，变量与地址的映射关系；

│ program.lis：编译后程序与其指针的映射关系，可以在调试的时候结合e命令查看程序的运行轨迹；

├─program：ROM中固化的代码，不可更改；

├─program\_patch：存放所有patch代码的文件夹，用户可自行在其中添加\*.prog文件；

│ patch\_init.prog：芯片初始化文件，其中的**p\_app\_init为初始化函数，芯片上电/唤醒只执行一次**

│ main\_patch.prog：patch代码开始的prog文件；

│ patch\_keyscan.prog：矩阵键盘的prog文件；

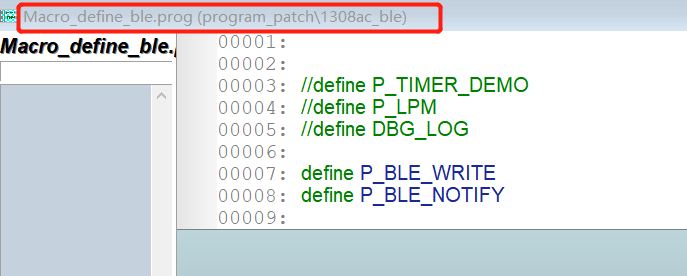
├─sched：存放.dat文件（可用于变量初始化）；

└─util：存放编译的脚本文件；

### 1.2 宏定义选择

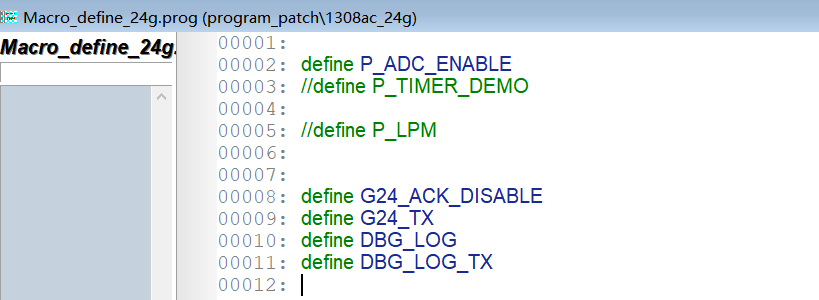
**ble工程：**

在macro\_define\_ble.prog中选择需要使用的宏定义，宏定义也可以自行添加。



**2.4G工程**

在macro\_define\_24g.prog中选择需要使用的宏定义，宏定义也可以自行添加。



### 1.3 程序入口

**p\_app\_init:**

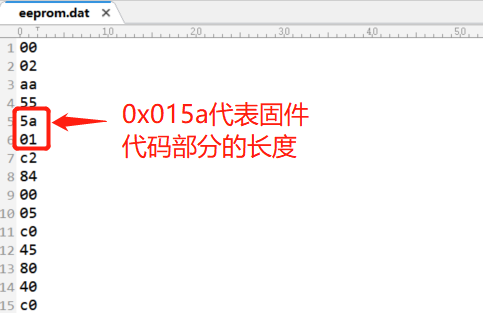
位于patch\_init.prog中，是ram中应用代码的初始化函数，也是应用程序执行的第一行，每次上电/低功耗唤醒执行一次。

**p\_idle\_process:**

位于main\_patch.prog中，main函数，类似于while(1);不开蓝牙的情况下，百微秒左右会访问一次，开蓝牙情况下，就在不开蓝牙的耗时再加上蓝牙窗口的时间。如果蓝牙rx窗口为1ms，则访问间隔增加1ms。

## 2 固件结构&下载

### 2.1 eeprom.dat结构



### 2.2固件下载

如果辅芯片为eeprom的芯片，在下载前需要先关写保护。下载eeprom时，运行**eep.bat**即可，eep.dat包含了关写保护和烧录固件。

编译下载时，检查代码是否下载成功的技巧是：**在下载代码前读一次otp，看是否为空，下载后再读一次otp，对比eeprom.dat的长度字节，即可判断是否烧录成功。**

### 2.3 ram空间

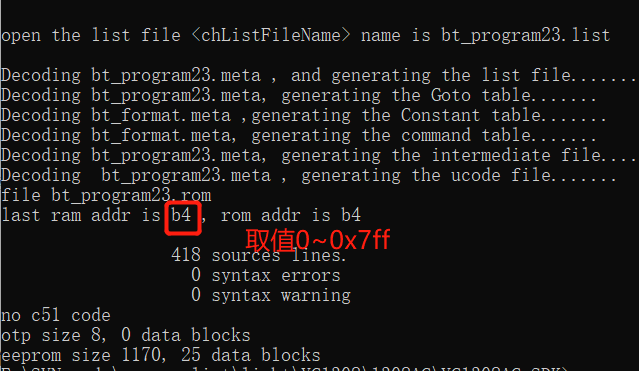
#### 2.3.1 code ram

1306的RAM为2K byte，共1024 \* 2 = 2048 byte。

**1 合封otp的芯片**

一行汇编代码4字节，所以2K byte ram最大可以写0x200行代码，取值0~0x1ff.

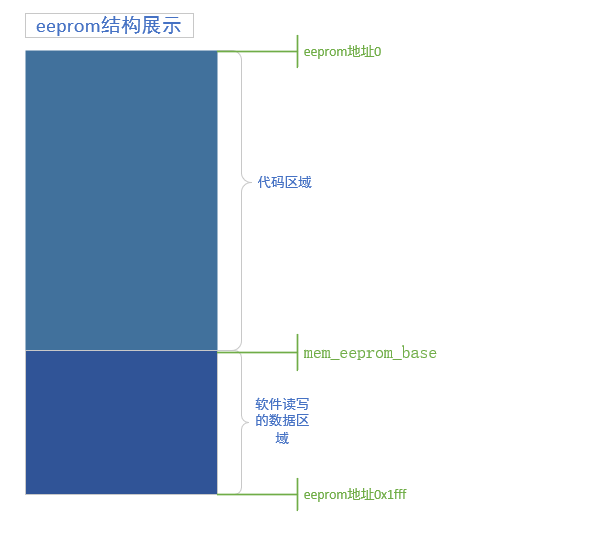
剩余代码量可在编译界面（do eep）中查看。如下图：



**2合封eeprom的芯片**

代码**和**需要掉电保存的数据都是存储在eeprom中，所以实际最大代码行数受限于eeprom的大小。

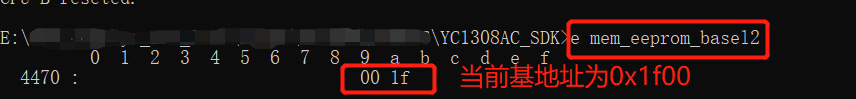
软件读写eeprom的掉电保存数据的方式是，选定一个读写数据的地址，作为基地址，然后把需要存储的数据，依次在该基地址上往后偏移。基地址保存在mem\_eeprom\_base中。



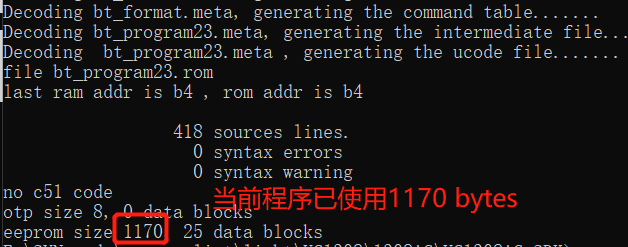
下图为eeprom结构展示图：

以合封64k bits（8k bytes）的eeprom为例，计算剩余的代码空间。

1. 查看自己代码中mem\_eeprom\_base的值。因为程序代码，不能超过该变量所指向的eeprom空间。（该变量的值根据自己软件需要自行更改）



2、查看编译结果（do eep）界面eeprom size的值。



**结合以上两点，当前程序可使用的最大字节数为：0x1f00 (7936) bytes, 当前已使用1170 bytes，还剩余7936 - 1170 = 6766 bytes。一行汇编代码占4字节，所以还剩6766 / 4 = 1691行代码。**

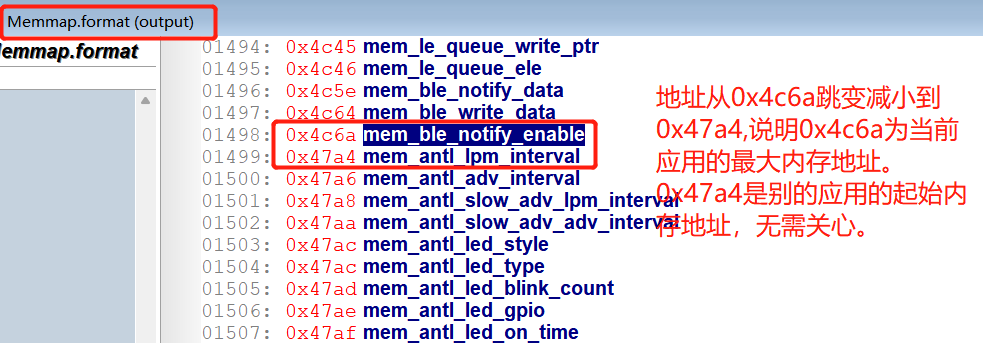
#### 2.3.2 data ram

1 、memalloc()关键字申请的变量取值范围：0~0xfff，4K Bytes。这部分内存，进休眠后值**不会**保持。每次唤醒后都是随机值。

2 、xmemalloc()关键字申请变量取值范围：0x4000~0x4fff，这部分内存，进休眠后值**会**保持，即唤醒后，保持睡眠前的值。

查看变量地址的方式：

output文件夹下有个memmap.format文件，里面有已经分配的内存分布。内存的分布是递增的，当发现两个相邻地址，后一个地址小于前一个地址，则表明前一个地址为当前应用分配到的最大内存地址。如下图：



## 3 eeprom 读写接口

**写接口：**

iicd\_write\_protect\_eep\_data

**读接口：**

iicd\_read\_eep\_data

**参数：**

temp:length

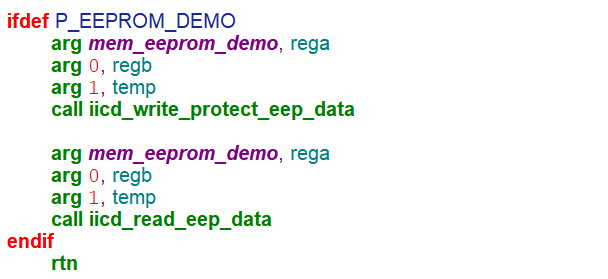
rega:mem ptr

regb:eeprom addr

备注 ：eeprom 读写地址为mem\_eeprom\_base + regb。

其中mem\_eeprom\_base是变量，为当前软件，读写eeprom的基地址，它的赋值可以在软件初始化中完成。64k bit的eeprom， mem\_eeprom\_base的取值范围为0~0x2000。确定好mem\_eeprom\_base后，regb的值为以mem\_eeprom\_base为基础往后的偏移量。如，mem\_eeprom\_base为0x1000, regb为1，则最终操**作**的eeprom的地址为0x1001。

下图为参考代码：



## 4 BLE 相关

### 4.1 adv/scan

**1 BLE adv**

接口名——“p\_le\_send\_adv\_ind”

参数：

mem\_le\_adv\_enable：

使能位，置1，enable adv；置0，disable adv

mem\_le\_adv\_interval:

广播间隔，单位：0.625ms

需要广播的数据写入到mem\_le\_adv\_data中

BLE 的广播包包结构如下所示：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Preamble | Access Address | PDU | | | CRC |
| 1 Byte | 4 Bytes | PDU Header | Adv Address | Adv Data | 3 Bytes |
| 2 Bytes | 6 Bytes | 0 - 31 Bytes |

Preamble ——0xAA。

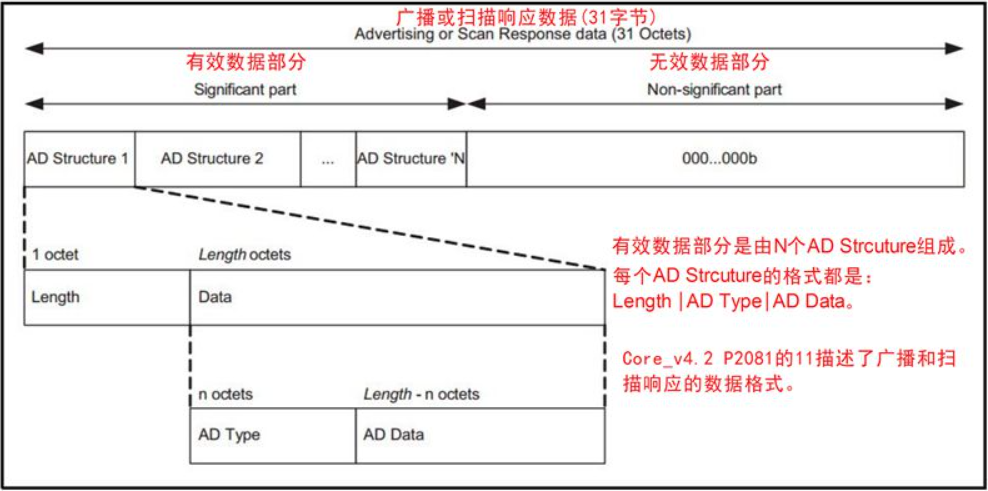
Access Address ——使用 BLE 通用广播的配置{0xD6, 0xBe, 0x89, 0x8E}，CRC 的作用范围为 PDU（Header, Adv Address, Adv Data）。

PDU Header ——高位表示广播模式，配置为 0x42 （单向广播、随机地址）。低位为 PDU

Length表示 Adv Address 和 Adv Data 两部分的总长度，取值范围为 [6, 37]。

Adv Address ——设备地址

Adv Data ——由多个 AD Structure 组成，每个 AD Structure 需要满足特定格式：



每 个 包 都 是 31 字 节 ， 数 据 包 中 分 为 有 效 数 据 （ significant ） 和 无 效 数 据 （non-significant）两部分。

有效数据部分 ：包含若干个广播数据单元，称为 AD Structure 。如图中所示，AD

Structure 的组成是：第一个字节是长度值 Len ，表示接下来的 Len 个字节是数据部分。

数据部分的第一个字节表示数据的类型 AD Type ，剩下的 Len - 1 个字节是真正的数据 AD

data 。其中 AD type 非常关键，决定了 AD Data 的数据代表的是什么和怎么解析。

无效数据部分 ：因为广播包的长度必须是 31 个 byte，如果有效数据部分不到 31 自

己，剩下的就用 0 补全。这部分的数据是无效的，解释的时候，忽略即可.

CRC——报文的最后是3个字节的CRC。CRC对报头、长度域以及净荷域进行计算。

**2 BLE scan**

接口名——“p\_le\_scan\_patch”

参数：

mem\_le\_scan\_enable:

置1，enable scan；置0，disable scan;

mem\_le\_scan\_interval:

扫描间隔，单位：0.625ms

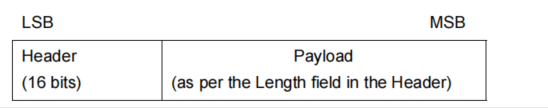
mem\_le\_scan\_window:

扫描窗口，单位：0.625ms

mem\_le\_rxbuf：

为收包数据buffer，可以通过e命令（e mem\_le\_rxbufm27）动态的查看它的变化。

**mem\_le\_rxbuf**的第0字节，对应协议中Header的第0字节。



*adv 帧结构*

备注：

实际的间隔为：mem\_le\_scan\_interval - mem\_le\_scan\_window， 所以mem\_le\_scan\_interval要大于mem\_le\_scan\_window。 adv和scan的间隔会在.dat中默认初始化。

### 4.2 BLE GATT

**1 UUID自定义**

service配置格式：

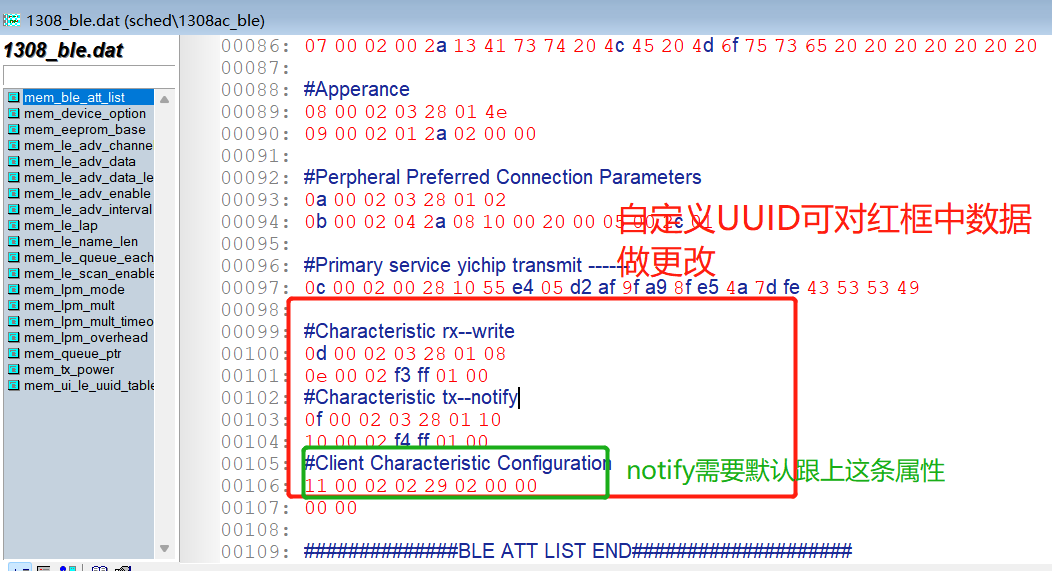
|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Attribute Handle | | Attribute Type Len | Attribute Type | | Attribute Value Len | Attribute Value |
| XX XX | | 02 | 00 28  UUID for «Primary Service» OR 0x2801 for «Secondary Service» | | 02 Or 10 | 16-bit Bluetooth UUID or  128-bit UUID for Service |
| byte0 | byte1 | byte2 | byte3 | byte4 | byte5 | byte6~n |

*characteristic 配置格式：*

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Attribute Handle | | Attribute Type Len | Attribute Type | | Properties  Len | Properties |
| XX XX  Characteristic Attribute Handle | | 02 | 03 28  UUID for «Characteristic» | | 01 | 08  Write  Or  10  Notify |
| byte0 | byte1 | byte2 | byte3 | byte4 | byte5 | byte6 |
| Attribute Handle | | Attribute Value Len | Attribute Value | | Len | data |
| XX XX  Characteristic Value Attribute Handle | | 02 Or 10 | 16-bit Bluetooth UUID or 128-bit UUID for Char  acteristic Value | | 01  固定不变 | 00  固定不变 |
| byte0 | byte1 | byte2 | byte3~byten-2 | | byten-1 | byten |

UUID的定义在“1306\_ble.dat中”，格式如上表。

示例：



**2 write attribute**

接口——“p\_ble\_receive\_writeatt”

示例：

下例中，把收到的write数据，存储在*mem\_ble\_write\_data中。*

*/\* rega is pointer to data, regb is length \*/*

*p\_ble\_receive\_writeatt:*

*copy regb, loopcnt*

*copy rega, contr*

*arg mem\_ble\_write\_data, contw //demo 中mem\_ble\_write\_data只申请了6字节，可自行调整*

*branch memcpy*

**3 notify attribute**

接口——“p\_ble\_send\_notify\_data\_set”

示例：

下例中，把需要notify的数据放在mem\_ble\_notify\_data中，然后入队，程序notify的时候，会从队列中取数据，然后发送。

*p\_ble\_send\_notify\_data\_set:*

*fetch 1, mem\_ble\_notify\_enable*

*rtn blank*

*jam 0, mem\_ble\_notify\_enable*

*fetch 1, mem\_ble\_notify\_data*

*pincrease 1*

*store 1, mem\_ble\_notify\_data*

***arg mem\_ble\_notify\_data, rega***

***branch queue\_push***

## 5 2.4G收发说明

**1）、2.4G发射接口**

接口名——“g24\_sim\_tx”

参数：

mem\_24g\_datalen：

发送数据的长度；

mem\_24g\_txbuf：

Tx buffer。

Demo中，发送了2个字节的数据，数据每1s自增1.

**2）、2.4G接收ACK**

ACK 标志位：

mem\_le\_24g\_recv\_ack\_flag

收到ACK置1，需要手动清0，是软件在维护。

收到的ACK信息可通过串口打印出来。

**Log打印：**

接口名——“p\_24g\_recv\_ack\_log”

Uart tx:pin7(GPIO11)

波特率：921600

**3）、2.4G接收接口**

接口名——“g24\_sim\_rx”

接收数据的buffer——“mem\_24g\_rxbuf”，其中mem\_24g\_rxbuf偏移2个字节处，为“mem\_24g\_rxbuf”所发送的数据。

收到2.4G数据包标志位：

mem\_le\_24g\_rx\_recv\_flag

收到数据包置1，需要手动清0，是软件在维护。

收到的2.4G信息可通过串口打印出来。

**Log打印：**

接口名——“p\_24g\_rx\_log”

Uart tx:pin7(GPIO11)

波特率：921600

**4）、2.4G发射ACK组包接口**

接口名——“p\_cb\_24g\_ackpayload”

ack的数据包规则和发2.4G数据包一样。

参数：

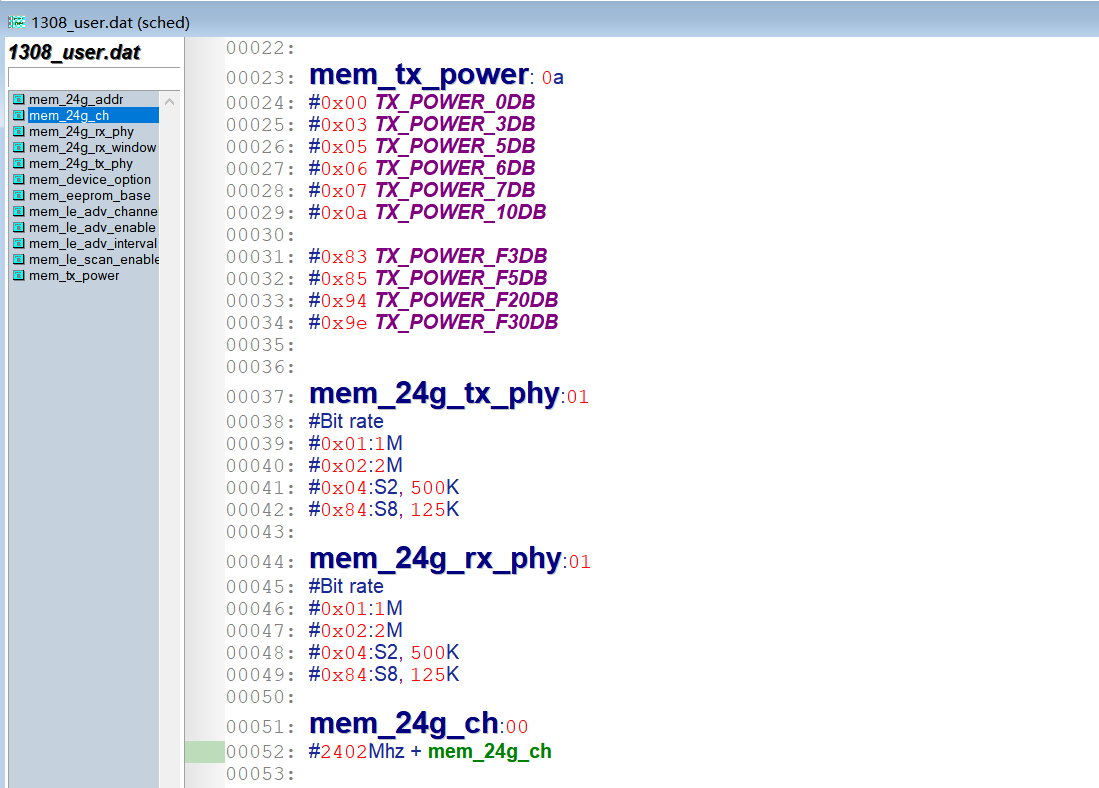
mem\_24g\_datalen：

发送数据的长度；

mem\_24g\_txbuf：

Tx buffer。

2.4G相关的变量配置如下图：



## 6 低功耗接口说明

### 6.1 lpm模式相关配置

芯片在进入睡眠（lpm模式）前，有两个地方需要配置。

1. 把未使用的IO配置为上拉或下拉输入，目的是给IO一个固定电平，防止IO漏电；
2. 配置唤醒源，指定特定的IO为低电平/高电平唤醒。

**1）配置gpio电平**

接口——“p\_lpm\_lpm\_gpio\_config\_flag\_init”

该接口中，已经配置了进休眠一定不能下拉的gpio。

应用中，有进休眠不能下拉的gpio给变量——“mem\_light\_lpm\_gpio\_config\_flag”的对应bit置1.

如：gpio5不需要下拉，就给mem\_light\_lpm\_gpio\_config\_flag bit5置1.

**注：Io11比较特殊，IO11无法作为唤醒io。**

1. **配置唤醒源**

接口——“p\_wakeup\_config”

需要给gpio配置为**高**电平唤醒时，将“mem\_gpio\_wakeup\_high”的对应bit值1；

需要给gpio配置为**低**电平唤醒时，将“mem\_gpio\_wakeup\_low”的对应bit值1。

1. **进入低功耗**

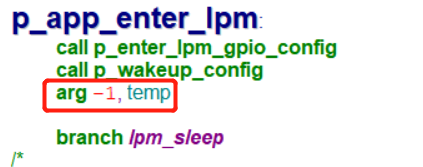
接口——“p\_app\_enter\_lpm”

gpio电平和唤醒源配置完成了，调用p\_app\_enter\_lpm芯片就会进入休眠模式

### 6.2 lpm模式自唤醒

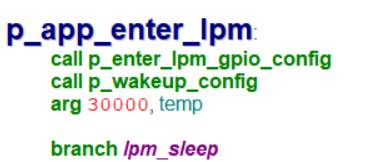
**睡眠时长参数**

在进入lpm模式之前需配置好lpm模式下的睡眠时长，单位：约为30kHz。

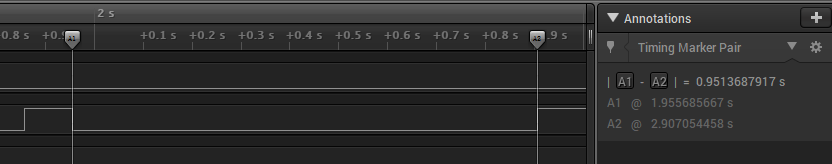


示例：

下例中，设置lpm模式下睡眠时长为1s。



通过在lpm模式下将io拉低，自唤醒后将io拉高可以观察到睡眠时长是否设置成功。搭配定时器使用，可以灵活配置醒来后再次进入低功耗模式时长。

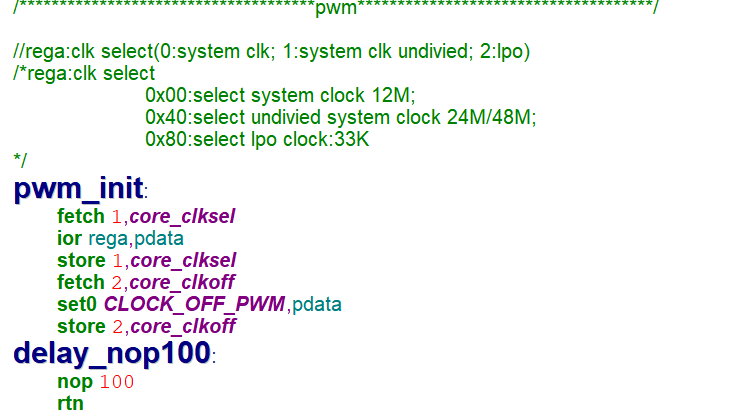


## 7 PWM模块

1. 接口名——”pwm\_init”

参数：

Rega：clk select



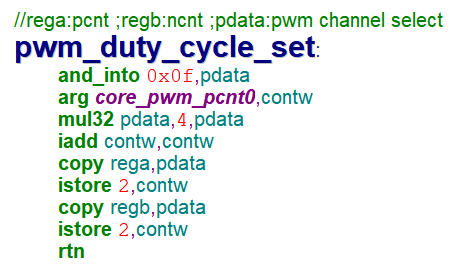
1. 接口名——”pwm\_duty\_cycle\_set”

参数：

Rega：pcnt

Regb：ncnt

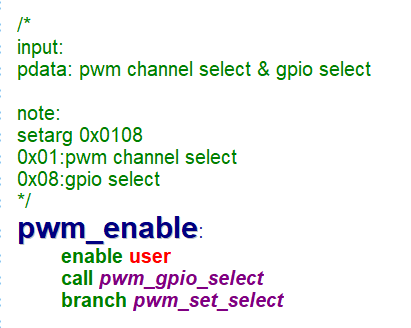
Pdata：pwm channel select



1. 接口名——”pwm\_enable”

参数：

pdata ：gpio select & pwm channel select



## 8 认证

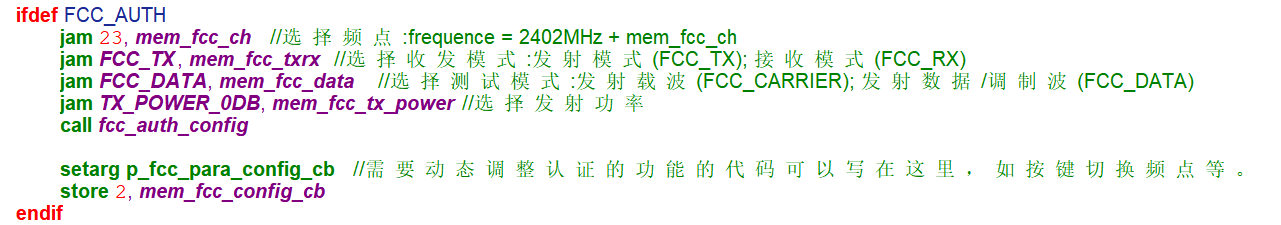
1、fcc\_auth\_config：

认证代码函数入口。

1. p\_fcc\_para\_config\_cb：

需要动态调整认证的功能的代码可以写在这里，如按键切换频点等。

3、频点切换，以及载波/数据切换的变量参数，如下图所示。



## 9 动态加载

### 9.1动态加载简介

仅有合封flash的芯片才会用到动态加载。

芯片存储资源：code ram：64K bits；flash：2M bits.

显然，flash中的代码无法一次性的加载到ram中运行。所以提出一个动态加载的概念：在ram留出一片空间（256bytes，后面称：动态加载空间），作为公共部分，当程序需要调用某一个接口时，如果这个接口没有写在ram里面，而是在flash中，这时芯片会将这个接口加载到动态加载空间中执行。当然，当需要执行下一次动态加载的接口时，之前的动态加载空间中的代码会被覆盖掉。也就是说，**动态加载空间的代码是动态的被覆盖的**。

动态加载的代码，写在“program\_zcode”文件夹中。

一次动态加载代码的行数的上限为64行。所以动态加载以64行为一块。不同块之间的分隔符为“org z”，为了与ram中的接口相区分，需要动态加载的的接口，可以都用字母“z”开头（不是必须的）。

### 9.2 动态加载代码空间说明

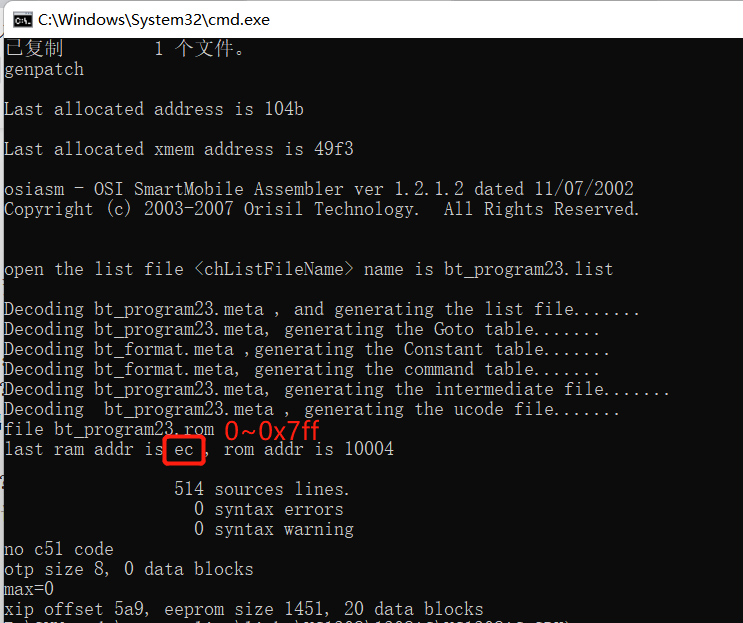
主要考虑两个问题：

一、ram是否溢出？

二、flash是否溢出？

一、ram是否溢出？

下图为一次编译截图：



因为RAM的空间为64K bits（8K bytes），一行代码4bytes，

固共可写汇编语句:8K / 4 = 0x800行，所以编译界面的”last ram addr”超过0x7ff即为ram溢出。

二、flash是否溢出？

尽管代码是动态加载的，一个动态加载块会产生2条附加的汇编代码，即一个动态加载块，至少耗8 bytes，粗略计算一下，假设所有ram都用来做动态加载使用，最多可以有1024（8192/8=1024）个动态加载块。每个动态加载块固定256 bytes（不到64行代码的部分，自动补0），则刚好是2M bits的空间。但是实际应用中，会有相当部分的ram不会用做动态，即实际应用时动态加载块远小于1024块，所以2M bits flash不会被写满。

所以结论就是：**在8k bytes ram的情况下，2M bits的flash不会被写满，不会溢出。**

## 10 ICE使用说明

ice脚存在一个默认上拉，且ice脚（gpio19）在做ice使用情况下，不具备常规gpio的功能，所以与ice合封了一个gpio。ice脚在ice模式下，上拉不能去掉，所以在对**合封gpio**操作时，注意一下几点：

1、可做上拉使用，如做按键扫描，初态上拉输入，按键低电平有效；

1. 做下拉输入时，因为上拉电阻一直存在，所以电平无法被完全拉低，一般不会这么用；
2. 可做输出高低使用，但是一旦配了输出使用，ice就无法通信。要想再次通信，只有下次上电时，让芯片在跑到配置输出的代码前暂停下来。

ice是无法直接操作gpio19的寄存器的，在复用为ice功能的时候，它gpio的功能是处于关闭状态。要ice输出低电平，有两种方式：

1、关闭ice功能，这样gpio19就是一个普通gpio，同时它的硬件上拉也被取消。但是ice无法通信；

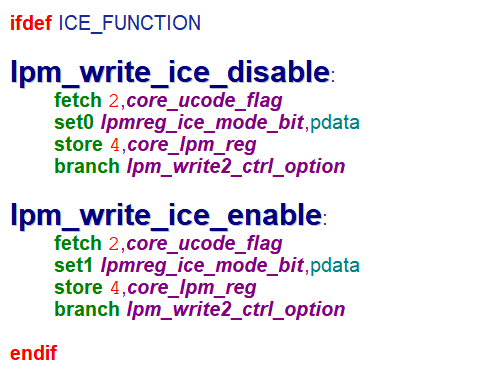
2、看芯片跟ice这个pin脚上，是否还拉了其他的gpio，如果要输出低，就把另一个gpio配置输出低模式，这个时候ice也会被拉低**。但是这种情况，由于ice的上拉电阻存在，与bonding在一起的另一个gpio的输出低模式，形成了通路，就会漏电。**

1、lpm\_write\_ice\_disable：

关闭ICE功能

2、lpm\_write\_ice\_enable：

使能ICE功能



**备注：ice的通信的条件：该pin脚，保持上拉输入特性。**

## 11 ADC使用说明

### 11.1 ADC功能PIN脚

YC1306 拥有8个外部通道输入的ADC和内部输入源HVIN。

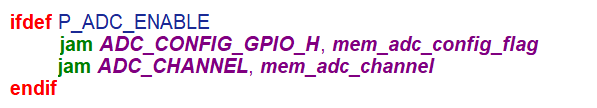
### 11.2 读取设置

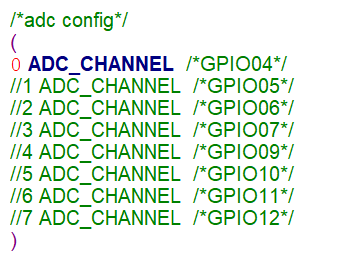
设置ADC读取模式（mem\_adc\_config\_flag）-( 1 hvin 3 gpio\_L 4 gpio\_H)、GPIO读取ADC通道(mem\_adc\_channel)--(未使用GPIO读取模式时不用填写)。

**hvin通道**



**gpio通道**





ADC结果存储在mem\_adc\_voltage\_result变量中

