esp32-mesh重构总结

`mesh`

本文档主要对 esp32-mesh 重构的总结，重构过程主要从如下几个方面进行：
美化代码：调整代码格式，清理无用的头文件，未使用的函数，无用的信息，拆分长函数；
精减函数：在不改变功能和可读性的情况下，重新整理函数逻辑，删除无用的变量。将相似功能的函数进行归并；重构框架：引入事件机制，将所有的事件处理进行了统一，将http的解析提到了用户应用层，将不同功能的模式隔离，以达到高内聚低耦合；现将重构时的格式调整，编程思想进行整理，保证后序项目开发的一致性。

1 格式

1.1 版权信息

给每个文件，都注上版权信息。

```
/*
 * ESPRESSIF MIT License
 *
 * Copyright (c) 2017 <ESPRESSIF SYSTEMS (SHANGHAI) PTE LTD>
 *
 * Permission is hereby granted for use on ESPRESSIF SYSTEMS ESP8266 only, in which case,
 * it is free of charge, to any person obtaining a copy of this software and associated
 * documentation files (the "Software"), to deal in the Software without restriction, including
```

```
 * without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense,
 * and/or sell copies of the Software, and to permit persons to whom the Software is furnished
 * to do so, subject to the following conditions:
 *
 * The above copyright notice and this permission notice shall be included in all copies or
 * substantial portions of the Software.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
 * IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS
 * FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR
 * COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER
 * IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN
 * CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
 *
 */
```

1.2 C/C++混合编程

在函数的头文件中加入如下的代码：

```
#ifdef __cplusplus
extern "C" {
#endif

......

#ifdef __cplusplus
}
#endif
```

extern "C"的真实目的是实现类C和C++的混合编程。extern “C”是由 C++ 提供的一个连接交换指定符号，用于告诉 C++ 这段代码是 C 函数。extern “C”后面的函数不使用的C++的名字修饰,而是用C。这是因为C++编译后库中函数名会变得很长，与C生成的不一致，

造成 C++ 不能直接调用C函数。

1.3 头文件中的#ifndef

把头文件的内容都放在#ifndef和#endif中，避免被多个文件引用时冲突，标识的命名规则一般是头文件名全大写，前后加下划线，并把文件名中的"."也变成下划线。

```
#ifndef __MESH_MEM_H__
#define __MESH_MEM_H__

......

#endif /*!< __MESH_MESH_H__ */
```

1.4 程序注释

1. 函数注释

```
/**
 * @brief brief description
 * @param[in|out] <parameter-name> <parameter description>
 * @param[in|out] <parameter2-name> <parameter2 description>
 * @return <description of the return value>
 * @note
 * detailed description
*/
```

2. 单行注释

```
/*!< brief description */
```

1.5 长语句

较长的语句（ >80字符 ）要分成多行书写，长表达式要在低优先级操作符处划分新行，操作符放在新行之首，划分出的新行要进行适当的缩进，使排版整齐，语句可读。

```
report_or_not_flag = ((taskno < MAX_ACT_TASK_NUMBER)
```

```
                        && (n7stat_stat_item_valid (stat_item))
                        && (act_task_table[taskno].result_data != 0));
```

1.6 函数长度

一个函数中的代码最好不要超过600行左右，越少越好，最好的函数一般在100行以内，300行左右的函数就差不多

1.7 格式化工具

提交代码之前使用格式化工具对代码进行格式化，之保证代码风格的统一

1. 将window格式转化成linux格式

```
find . -type f | xargs dos2unix
```

2. 将所有代码统一的风格格式化
   - 使用`astyle`格式代码

```
astyle -A3s4SNwm2M40fpHUjk3n
```

   - 使用 esp-idf 中的默认格式化脚本

```
./esp-idf/tools/format.sh
```

2 命名

2.1 文件及函数名

文件及函数名之前需加入“模块”标识，防止编译、链接时产生冲突。

2.2 变量名

全局变量需加上 g，静态变量加 s

2.3 宏名

宏定义的名称必须全为大写

2.4 常量

避免使用不易理解的数字，用有意义的标识来替代。涉及物理状态或者含有物理意义的常量，不应直接使用数字，必须用有意义的枚举或宏来代替。

```
#define TRUNK_IDLE 0
#define TRUNK_BUSY 1

if (Trunk[index].trunk_state == TRUNK_IDLE)
{
    Trunk[index].trunk_state = TRUNK_BUSY;
    ...  // program code
}
```

3 空间

3.1 结构体内存对齐

各个硬件平台对存储空间的处理上有很大的不同。一些平台对某些特定类型的数据只能从某些特定地址开始存取。其他平 台可能没有这种情况， 但是最常见的是如果不按照适合其平台要求对数据存放进行对齐，会在存取效率上带来损失。比如有些平台每次读都是从偶地址开始，如果一个int型（假设为 32 位系统）如果存放在偶地址开始的地方，那么一个读周期就可以读出，而如果存放在奇地址开始的地方，就可能会需要2个读周期，并对两次读出的结果的高低字节 进行拼凑才能得到该int数据。显然在读取效率上下降很多。

```
struct A {                          struct B {
#progma pack (1)
    int a;                            char b;                          struct
C {
    char b;                           short c;                          char
b;
    short c;                          int  a;                           int a
```

```
;
};                                  };                                  short
c;}
```

在A结构，3个不同的4个字节被分配到三种数据类型，而在B结构的前4个自己char和short可以被采用，int可以采纳在第二个的4个字节边界（一共8个字节），在C结构
结构体A中包含了4字节长度的int一个，1字节长度的char一个和2字节长度的short型数据一个。所以A用到的空间应该是7字节。但是因为编译器要对数据成员在空间上进行对齐。
所以使用sizeof(strcut A)值为8。sizeof(struct B)的值是12。sizeof(struct C)的值是7。

`#progma pack (n)` 指定结构体对齐方式。#pragma pack(n)来设定变量以n字节对齐方式。
n 字节对齐就是说变量存放的起始地址的偏移量有两种情况：
第一、如果n大于等于该变量所占用的字节数，那么偏移量必须满足默认的对齐方式，
第二、如果n小于该变量的类型所占用的字节数，那么偏移量为n的倍数，不用满足默认的对齐方式。
结构的总大小也有个约束条件，分下面两种情况：如果n大于所有成员变量类型所占用的字节数，那么结构的总大小必须为占用空间最大的变量占用的空间数的倍数； 否则必须为n的倍数。

```
#pragma pack(push) //保存对齐状态
#pragma pack(4)//设定为4字节对齐
struct test
{
char m1;
double m4;
int m3;
};
#pragma pack(pop)//恢复对齐状态
```

3.2 内存分配

动态分配内存时，以2的n次方分配，这样可以减少产生内部碎片

3.3 作用域

限定变量的作用域，尽可以避免使用全局变量

```
do {
    char verson[64] = {0};
    sprintf(verson, "v1.3.0 %s %s", __DATE__, __TIME__);
    mesh_json_pack(mesh_info, "ver", verson);
} while (0);
```

4 效率

4.1 分支语句

1. 移除分支预测，我们可以使用一些方法，移除条件判断，避免由于分支预测错误，破坏流水线工作，进而来提高性能，例如我可以将下面的分支语句：

```
if (data[c] >= 128)          |          int t = (data[c] - 128) >> 31;
    sum += data[c];          |          sum += ~t & data[c];
```

2. 使用条件传送指令，对于简的条件语句可以使用分支语句可以使用，一条重要任务传送指令来替换。

3. switch
   在程序中经常会使用switch-case语句，每一个由机器语言实现的测试和跳转仅仅是为了决定下一步要做什么，就浪费了处理器时间。为了提高速度，可以把具体的情况按照它们发生的相对频率排序。即把最可能发生的情况放在第一，最不可能发生的情况放在最后，这样会减少平均的代码执行时间。当switch语句中的case标号很多时，为了减少比较的次数，明智的做法是把大switch语句转为嵌套switch语句。把发生频率高的case标号放在一个switch语句中，并且是嵌套switch语句的最外层，发生相对频率相对低的case标号放在另一个switch语句中。比如，下面的程序段把相对发生频率低的情况放在缺省的case标号内。

4.2 位操作

1. 使用位操作代替*、/、%运算来实现
   左移相对于乘法，右移相当于除法，与相对于模运算

```
num *=15;
```

```
    num = num >> 8 +mun >> 4 + num;
    num = num >> 16 - num;

    num %= 4;
    num &= 3;
```

2. 常用位操作小技巧
    - 检查整数是奇数还是偶数 `if ((x & 1) == 0)`
    - 交换两个数 `a^=b^=a^=b`
    - 变换符号 `~a + 1`
    - 求绝对值 `i = (a >> 31) & 1; ((a ^ i) - i);`

4.3 内联函数

消除函数调用时的时间开销。它通常用于频繁执行的函数，对于小内存空间的函数非常受益。
内联相对于宏定义：
1. 宏的定义很容易产生二意性。
2. 宏定义参数检查不方便

```
inline uint16_t swap_byte_16(uint16_t x)
{
    return (((x & 0x00ffU) << 8) |
            ((x & 0xff00U) >> 8));
}
```

> 注：一个可执行文件的源代码中，同一个函数只能被定义一次；所以，你如果把函数定义放在.h头文件中，然后有两个.c源文件同时包含该头文件，就会出现函数重复定义的错误，但是，如果将该函数声明为inline函数，则不会报错。

4.4 无符号整数

有些处理器处理无符号的整数，比有符号整数的运算速度要快

4.5 指针代替数组

在许多种情况下，可以用指针运算代替数组索引，这样做常常能产生又快又短的代码。与数组索引相比，指针一般能使代码速度更快，占用空间更少。使用多维数组时差异更明显。下面的代码作用是相同的，但是效率不一样。

```
for(;;) a=array[t++];                for(p=array;;) a=*(p++);
```

4.6 增量和减量操作符

在使用到加一和减一操作时尽量使用增量和减量操作符，因为增量符语句比赋值语句更快，原因在于对大多数CPU来说，对内存字的增、减量操作不必明显地使用取内存和写内存的指令。

5 习惯

5.1 使用typedef替代macro

在可能的情况下使用typedef替代macro。当然有时候你无法避免macro，但是typedef更好。

```
typedef int* INT_PTR;
INT_PTR a, b;
# define INT_PTR int*;
INT_PTR a, b;
```

在这个宏定义中，a是一个指向整数的指针，而b是只有一个整数声明。使用typedef a和b都是整数的指针。

5.2 内部函数加 static

确保声明和定义是静态的，除非您希望从不同的文件中调用该函数。
在同一文件函数对其他函数可见，才称之为静态函数。它限制其他访问内部函数，如果我们希望从外界隐藏该函数。现在我们并不需要为内部函数创建头文件，其他看不到该函数。
静态声明一个函数的优点包括：
A）两个或两个以上具有相同名称的静态函数，可用于在不同的文件。

B ）编译消耗减少，因为没有外部符号处理。

5.3 if语句对出错的处理

先对错误情况进行处理，尽可能的不使用 else，这种做有如下两个好处

```
if ( ch >= '0' && ch <= '9' ) {
    /* 正常处理代码 */
} else {
    /* 输出错误信息 */
    printf("error ......\n");
    return ( FALSE );
}
```

这种结构很不好，特别是如果"正常处理代码"很长时，对于这种情况，最好不要用else。先判断错误，如：

```
if ( ch < '0' || ch > '9' ){
   /* 输出错误信息 */
   printf("error ......\n");
   return ( FALSE );
}

/* 正常处理代码 */
```

这样的结构，突出了错误的条件，让别人在使用你的函数的时候，第一眼就能看到不合法的条件，于是就会更下意识的避免。

5.4 出错信息的处理

出错信息或是提示信息，应该统一处理，如果要管理错误信息，那就要有以下的处理：

```
/* 声明出错代码 */
#define     ERR_NO_ERROR    0  /* No error                */
#define     ERR_OPEN_FILE   1  /* Open file error          */
#define     ERR_SEND_MESG   2  /* sending a message error  */

/* 声明出错信息 */
char* errmsg[] = {
```

```
    /* 0 */       "No error",
    /* 1 */       "Open file error",
    /* 2 */       "Failed in sending/receiving a message",
};

/* 声明错误代码全局变量 */
long errno = 0;

/* 打印出错信息函数 */
void perror( char* info)
{
    if ( info ) {
        printf("%s: %s\n", info, errmsg[errno] );
        return;
    }

    printf("Error: %s\n", errmsg[errno] );
}
```

一个即有共性，也有个性的错误信息处理，这样做有利同种错误出一样的信息，统一用户界面，而不会因为文件打开失败，A程序员出一个信息，B程序员又出一个信息。而且这样做，非常容易维护。

5.6 用for做循环

基本上来说，for可以完成while的功能，我是建议尽量使用for语句，而不要使用while语句，特别是当循环体很大时，for的优点一下就体现出来了。因为在for中，循环的初始、结束条件、循环的推进，都在一起，一眼看上去就知道这是一个什么样的循环。

```
p = pHead;
while ( p ){
    ...
    p = p->next;
}
```

当while的语句块变大后，你的程序将很难读，用for就好得多：

```
for ( p=pHead;  p; p=p->next ){
    ...
}
```

for(;;)与while(1) for循环明显在语义上更适合。更容易理解括号里面是循环的条件。括号循环条件根本不写，就是〖无条件循环〗。即便没有认识这个结构的也很容易猜到，没有循环条件就是无条件循环。但是while(1) 从代码风格的角度来说并不好，因为它额外的引入了一个常量，这在没有充分优化的编译器上一定比无条件for循环效率低。

5.7 使用sizeof类型

```
1.    pScore = (int *) malloc( sizof(int) * SUBJECT_CNT );
```

1. 易读：很明确的看出分配的长度
2. 易维护：sizeof 内是指针而不数组时出错

```
1.        memset( pScore, 0, sizeof(pScore) );
```

sizeof(pScore)返回的就是4（指针的长度），不会是整个数组，于是，memset就不能对这块内存进行初始化。

6 字符处理函数

6.1 strcpy和strncpy的缺陷分析

- strcpy：
  潜在的内存越界问题，当dest所指对象的数组长度 < src的数组长度时，由于无法根据指针判定其所指数组长度，故数组内存边界是不可知。因此会导致内存越界，尤其是当数组是分配在栈空间的，其越界会进入你的程序代码区，将使你的程序出现非常隐晦的异常。
- strncpy：
  1. 字符串结束标志符'\0'丢失
     当dest所指对象的数组长度 == count时，调用strncpy使得dest字符串结束标志符'\0'丢失。
  2. 效率较低
     当 count > src所指对象的数组长度时，会继续填充'\0'直到count长度为止。

6.2 sscanf

sscanf与scanf类似，都是用于输入的，只是后者以屏幕(stdin)为输入源，前者以固定字符串为输入源。sscanf支持通过正则表达式对字符进行解析，极大的方便对复杂字符串的处理。如给定一个字符串iios/12DDWDFF@122，获取 / 和 @ 之间的字符串，先将 "iios/"过滤掉，再将非'@'的一串内容送到buf中

```
sscanf("iios/12DDWDFF@122", "%*[^/]/%[^@]", buf);
printf("%s\n", buf);
```

结果为：12DDWDFF

6.3 strtok

切割字符串，将str切分成一个个子串

```
char buf[]="hello@boy@this@is@heima";
char*temp = strtok(buf,"@");
for(;temp;){
    printf("%s ",temp);
    temp = strtok(NULL,"@");
}
```

6.4 strtol

会将参数nptr字符串根据参数base来转换成长整型数。参数base范围从2至36，或0。参数base代表采用的进制方式，如base值为10则采用10进制(字符串以10进制表示)，若base值为16则采用16进制(字符串以16进制表示)。当base值为0时则是采用10进制做转换，但遇到如''0x''前置字符则会使用16进制做转换。

```
char a[] = "100";
char b[] = "100";
char c[] = "ffff";
printf("a = %d\n", strtol(a, NULL, 10)); //100
printf("b = %d\n", strtol(b, NULL, 2));  //4
printf("c = %d\n", strtol(c, NULL, 16)); //65535
```

7 do{}while(0)

1. 辅助定义复杂的宏，避免引用的时候出错
2. 避免使用goto对程序流进行统一的控制：
   这里将函数主体使用do()while(0)包含起来，使用break来代替goto，后续的处理工作在while之后，就能够达到同样的效果。
3. 避免空宏引起的warning
   4.定义一个单独的函数块来实现复杂的操作，当你的功能很复杂，变量很多你又不愿意增加一个函数的时候，使用do{}while(0);，将你的代码写在里面，里面可以定义变量而不用考虑变量名会同函数之前或者之后的重复。

8 日志工具

1. SecureCRT
   - 常用设置，使之能够显示 linux 下的颜色方便使用
     http://jingyan.baidu.com/article/cdddd41c7f7c5053cb00e1ea.html
   - 日志自动保存并加入时间戳
     http://jingyan.baidu.com/article/335530da88aa0b19cb41c3b9.html
2. minicom
   - 日志保存

     ```
     1.  minicom -c on -D /dev/ttyUSB$i -C "XXX.log"
     ```

   - 加入时间

     ```
     1.  ctrl+a n
     ```

3. monitor
   CSC 对idf monitor 进行了修改，使用如下指令就可以加入时间戳

```
1.  python idf_monitor.py -b 115200 -p /dev/ttyUSB$1 -sf $log_file_name esp
    32-mesh-demo.elf
```