**DJI入职前增肌计划RTOS组Week2**

**本周概览**

1、SPI采用DMA方式配合RTOS消息队列实现GUI优化（RTOS相关机制详解）；

2、SPI与WIFI的使用与配置，wireshark的使用；

3、SDIO配置与SD卡的读写；

4、FATFS或者其他FILELIB的使用；

5、JPEG硬件编解码；

6、通过FATFS实现bmp等格式图片读取&File Browser APP的开发；

7、图传与FFT格式字体加载；

**day7**

1、本日目标

（1）再次了解工程内存分配结构与MDK配置（包括编译优化等级），了解不同的分配和优化对程序的影响；

（2）在RTOS环境下使用DMA优化CPU使用率，学会使用线程标志位和事件标志位；

（3）完成WIFI模块的配置，并将其连接到路由器；

（4）提示：最近RTE框架更新频繁，请务必从Documents获取最新或者从当日Demo获取，旧工程不会同步更新RTE到最新版。

2、H7内存使用与分配

再次回到H7的内存图：

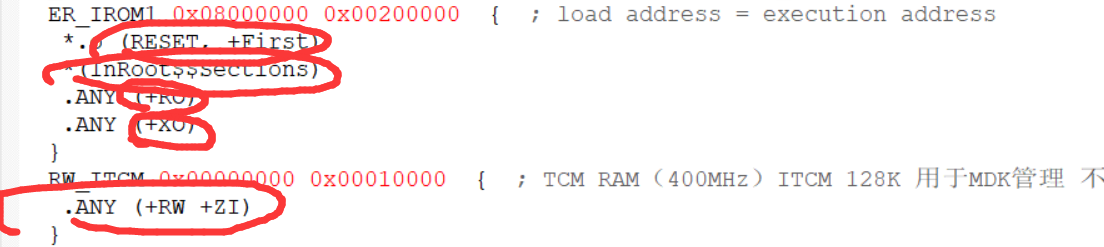
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **序号** | **名称** | **位域及速度** | **起始地址** | **大小** |
| 1 | AXI SRAM | D1域 200MHz | 0x2400 0000 | 512KB |
| 2 | AHB SRAM 1 | D2域 200MHz | 0x3000 0000 | 128KB |
| 3 | AHB SRAM 2 | D2域 200MHz | 0x3002 0000 | 128KB |
| 4 | AHB SRAM 3 | D2域 200MHz | 0x3004 0000 | 32KB |
| 5 | AHB SRAM4 | D3域 200MHz | 0x3800 0000 | 64KB |
| 6 | BKUP SRAM | D3域 200MHz | 0x3880 0000 | 4KB |
| 7 | DTCM RAM | 400MHz | 0x2000 0000 | 128KB |
| 8 | ITCM RAM | 400MHz | 0x0000 0000 | 64KB |

根据之前教程的内容，我们使用了ITCM作为MDK管理的内存，分配MDK在编译链接过程中产生对程序内各类变量等的分配，那么，剩下的空间我们是如何管理的呢？我们打开工程的分散加载文件（在day6工程中提供的），可以看到如下使用的地址分配：

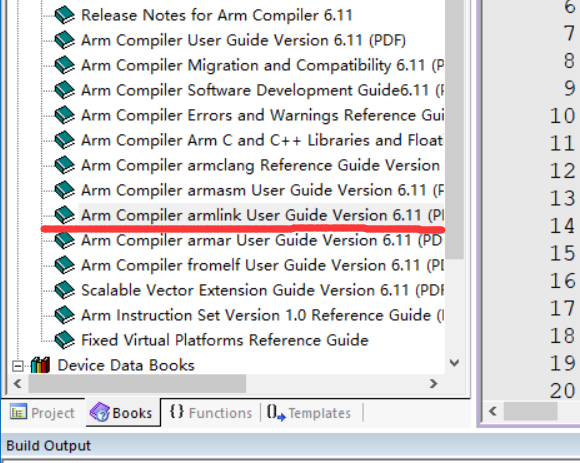


那么，我们是如何使用这些地址分配的呢？

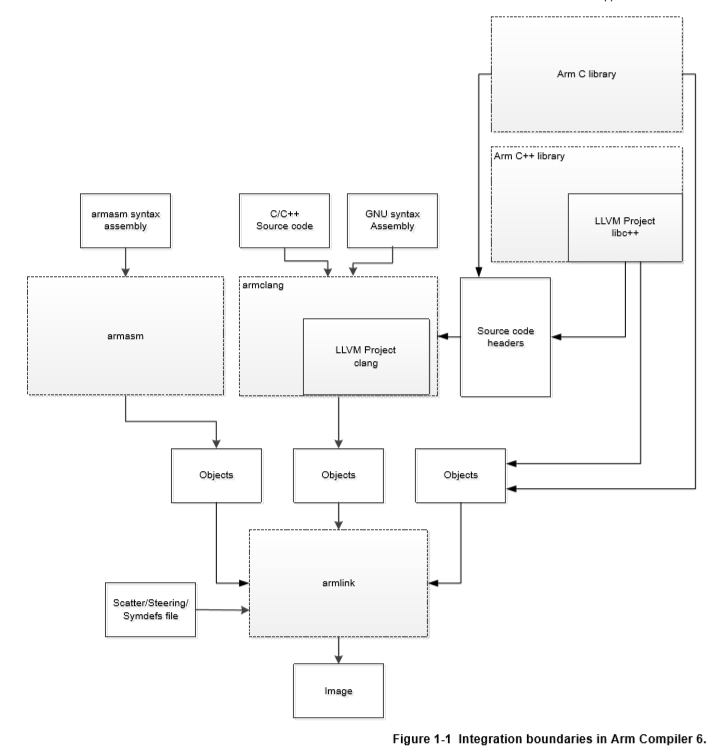
在步入正式内容之前，我们先了解一些基本概念，在上一节所描述到的分散加载文件中，有两块区域有特别的标识符：



打开ARMCC提供的文件：

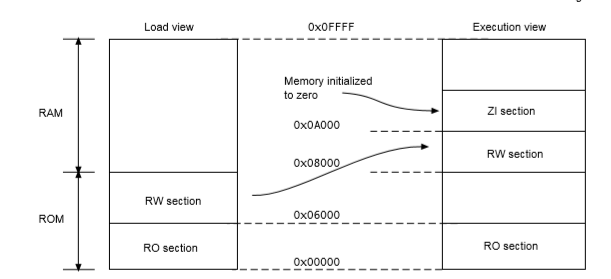


先看二十三页这张图：



在如上的图中，显示了ACC6的完成模块和大概的工作流程，我们所编写的分散加载文件在armlink这一模块的也就是链接部分起作用。

接下来看36页的这张图：



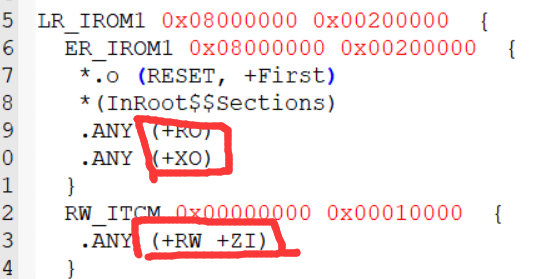
最终我们用MDK编译链接生成的文件为ELF镜像（可执行文件，相当于windows下的exe文件），ELF镜像包括sections、regions、segments这几个部分，每个链接阶段ELF的构成都有所差别。

上图显示的就是在加载阶段也就是执行用户代码之前完成的工作，把ROM中的RW拷贝到RAM中去，同时在RAM中建立ZI。这里RO代表常量所占空间，一般就是const定义的变量；RW代表程序中已经初始化的变量所占空间，就是在代码中定义的已经初始化的变量；ZI代表未初始化的static变量和全局变量以及堆栈所占的空间。

在编译完成时，MDK会有如下提示：

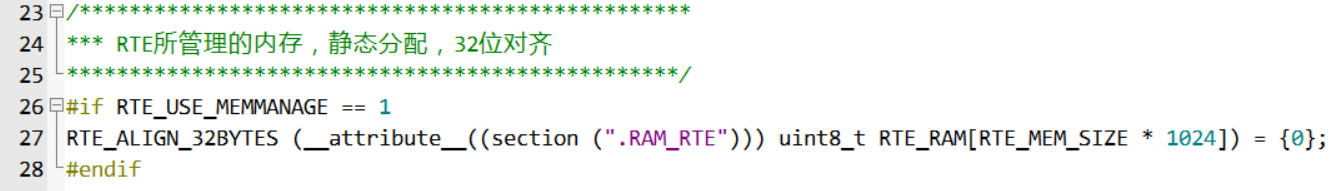


回到分散加载文件中



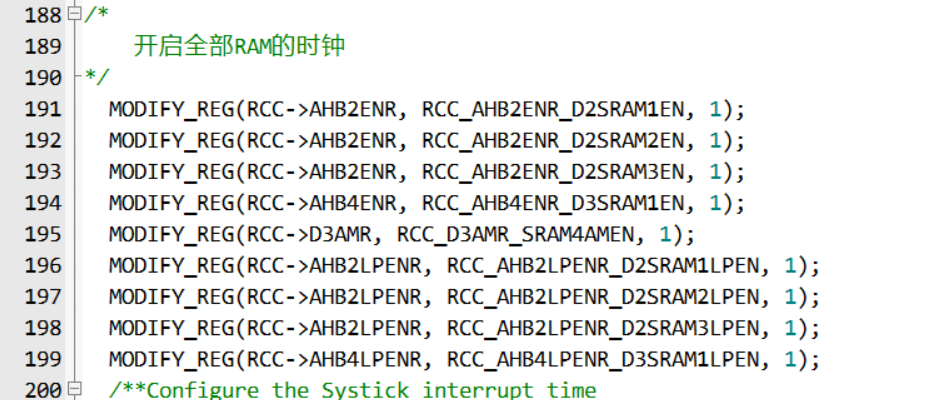
这里就可以比较清楚做个了解了，对于0x08000000的FLASH，也就是ROM区域，存放了RO和XO内容（XO实际没有），同时还包括了\*.o (RESET, +First)代表的启动代码的首次执行地址以及InRoot$$Sections代表的COPY RW区到RAM，然后再RW区后面创建ZI区。对于0x00000000指定的ITCM，作为RW和ZI区使用。剩下来的若干块，直接定义别名，指定使用，那么如何使用剩下来的几块区域呢？

可以打开RTE\_Port看一下：

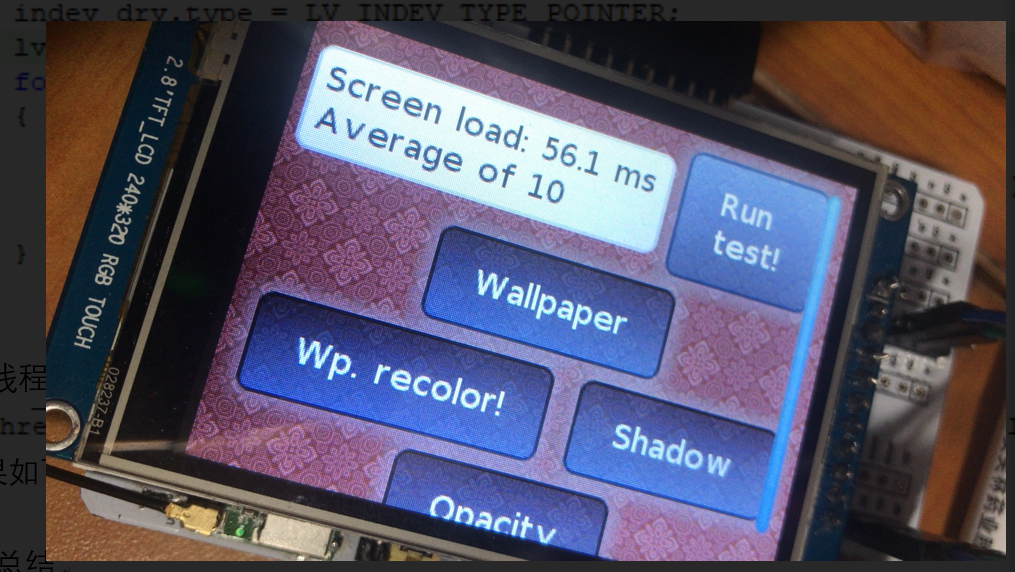


例如这里我们通过把一个RTE\_MEM\_SIZE大小的数组定义到.RAM\_RTE区，在通过动态内存管理模块对其完成分配使用。

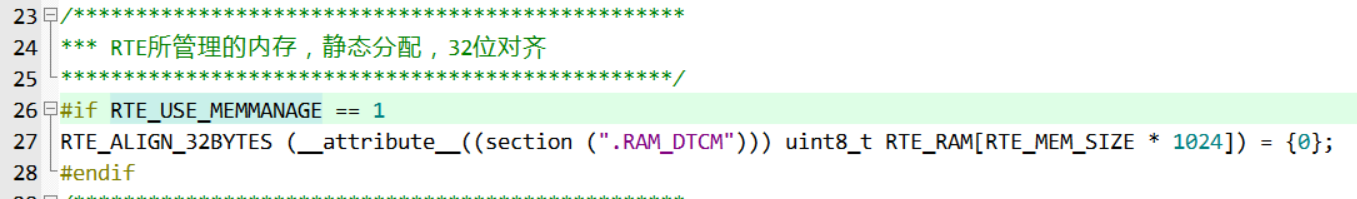
当然有的RAM需要打开时钟，我们在Board\_Init.c中完成了这项工作：



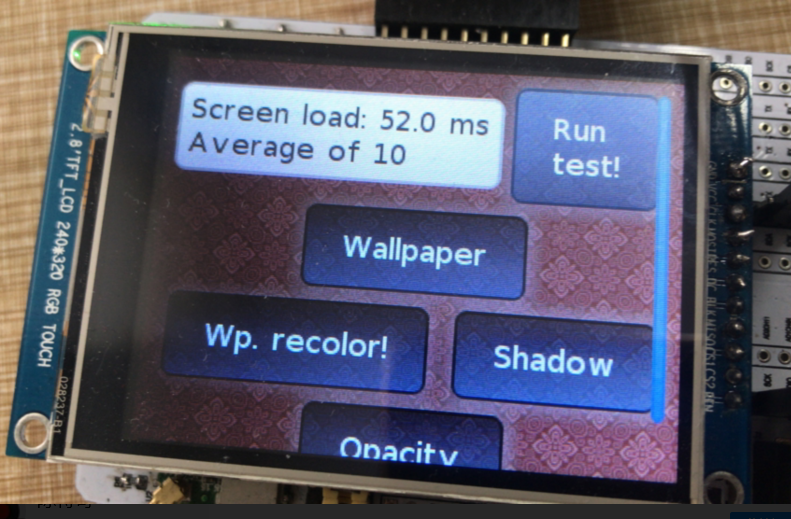
在day6工程中，我们使用了benchmark程序对工程做性能评估，当时特效全开的刷图速率如下，（MDK编译优化等级如下是oz-image size）：



或许有的同学可能更慢，我们尝试下不同速率RAM对程序的影响。在RTE\_Port中，我们定义RTE内存到DTCM区：



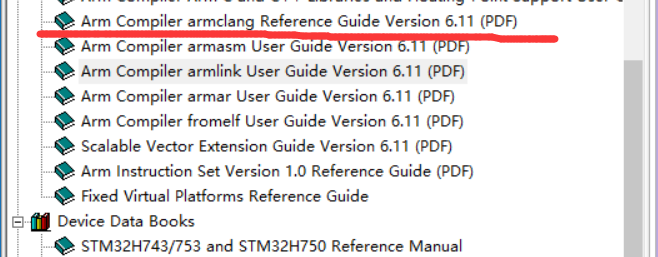
编译下载：



可以看到速率明显提升了，当然，由于我们的串口所使用的RingQuene也是从这里分配内存，并且串口用的是DMA方式，所以现在串口接收应该无效了（DMA不能访问TCM内存）。

3、编译器优化等级对程序的影响

同样地打开ARMCC的文档（根据第二小节的ACC框图，armclang就是编译阶段对非汇编源代码编译的编译器）：

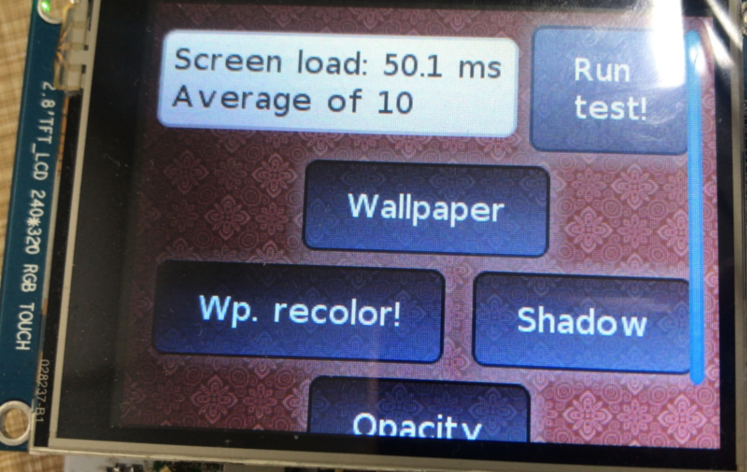


翻到第58页，可以看到编译器共有如下几个优化等级：

-O0 | -O1 | -O2 | -O3； -Os| -Oz；

之所以用分号间隔，是因为前后两组优化等级优化方向不一样，前一组重于性能而后一组重于生成code的大小。同时各组内从左到右优化程度越来越高。

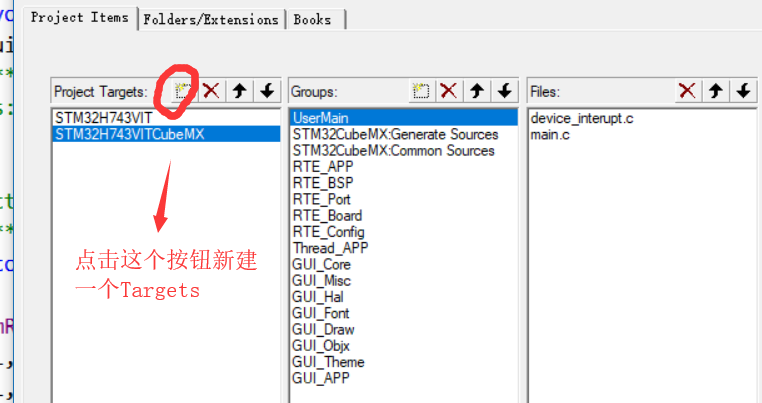
我们把原来的oz优化改成o3优化（这之前把RTE内存改回原有配置），编译下载：

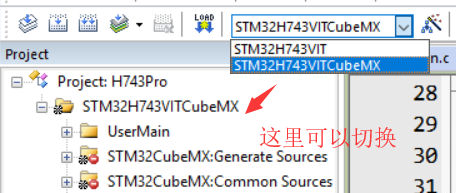


可以看到速度提升了，但是生成的code大小也提升了。根据手册的建议，在后面的开发中，统一使用o1等级优化，开发完成后，改成o3等级编译工程。

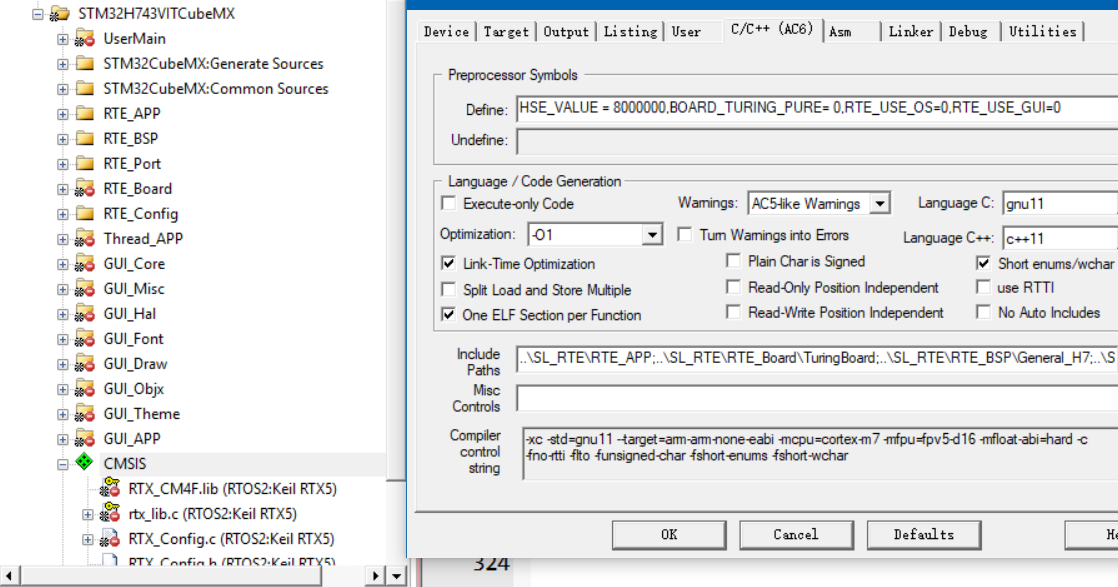
4、在RTOS环境下使用DMA优化CPU使用率

如果按照day6的做法，使用CubeMX的时候需要回复裸机运行，很麻烦，这里给出一劳永逸解决的方法：

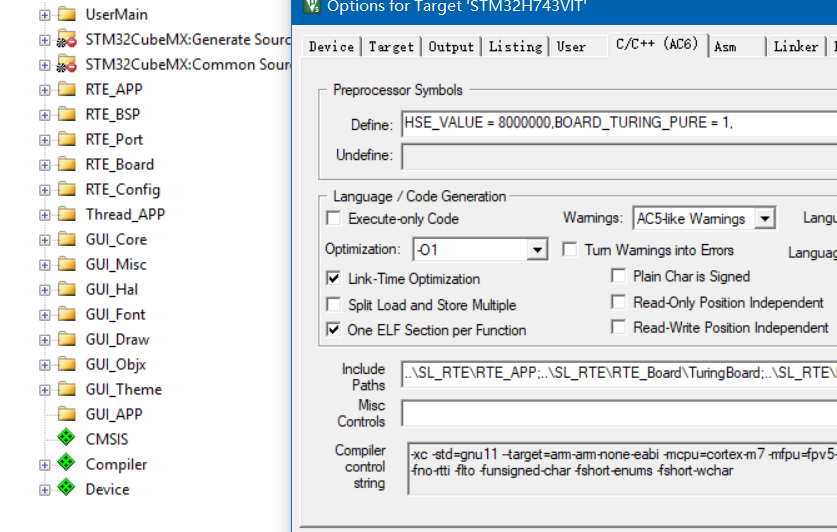




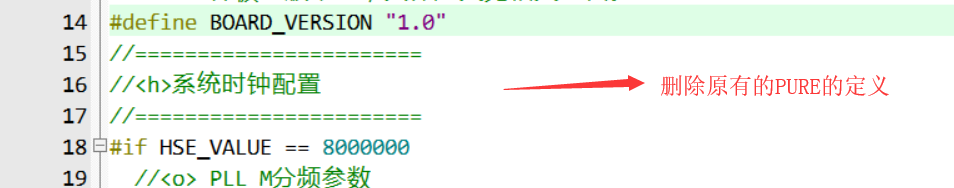
在CubeMX工程中，我们修改如下：



在纯净工程中，我们修改如下：



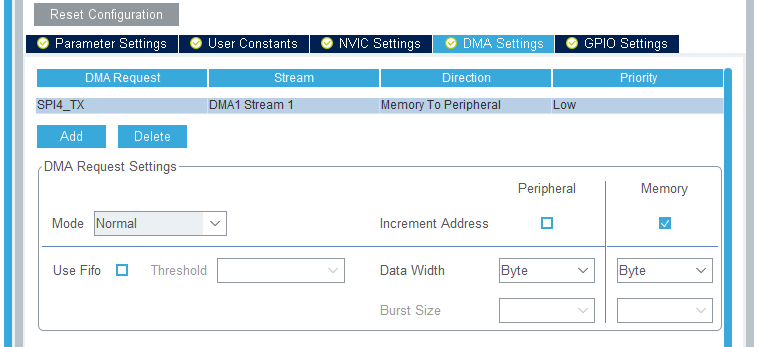
在Board\_Config.h中，修改如下：



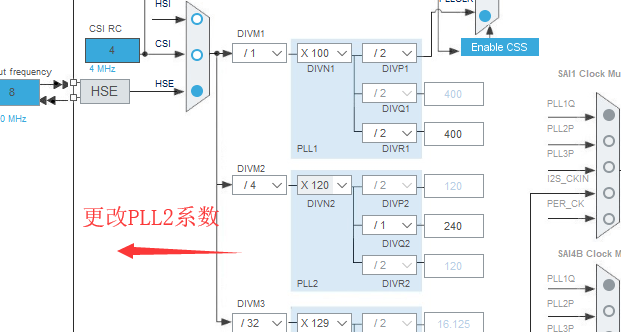
美中不足的是，EventRecoder的配置依然需要在切换后在MDK中重新修改。

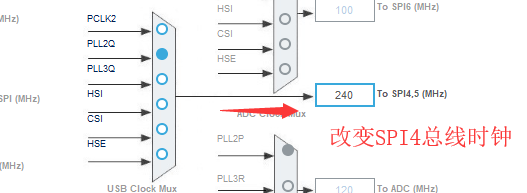
老样子，打开CubeMX：

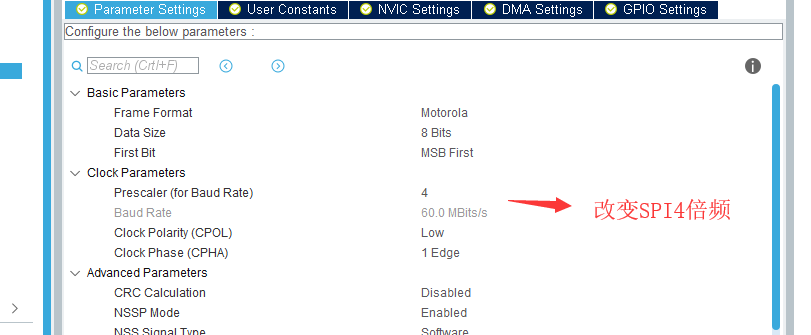
在SPI4的配置中为其添加TX\_DMA：



然后稍微给SPI超频一下：



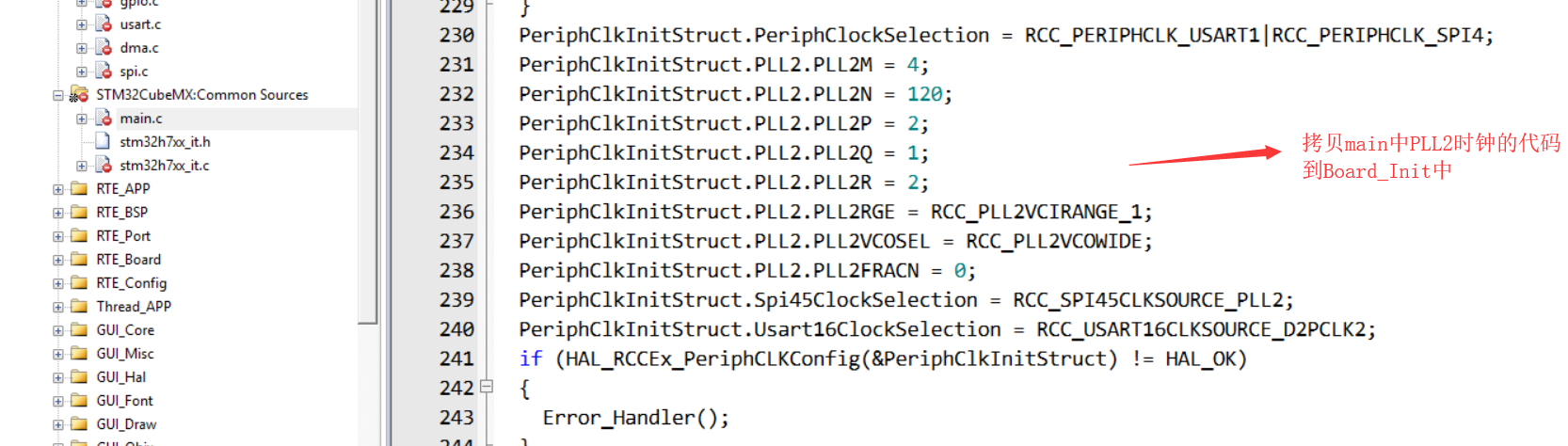


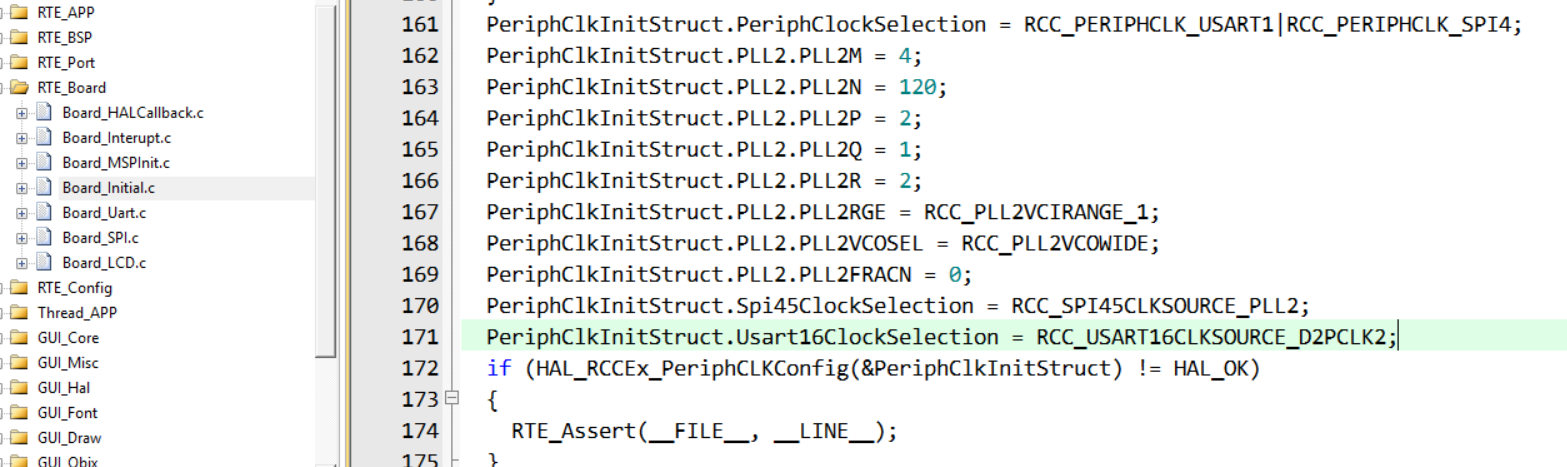


生成代码，编译通过。

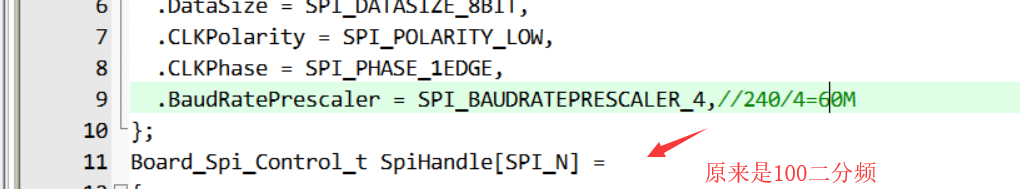
切换到Pure工程中。

这里我们先把超频的部分应用过来：





然后改变Board\_SPI的配置：

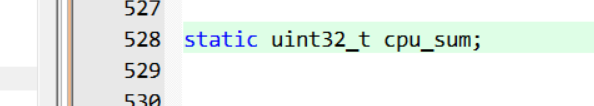


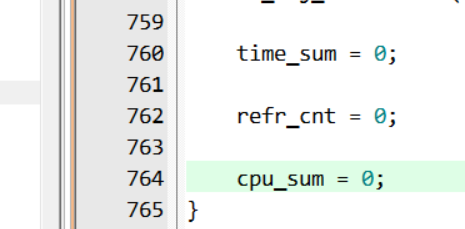
编译下载：

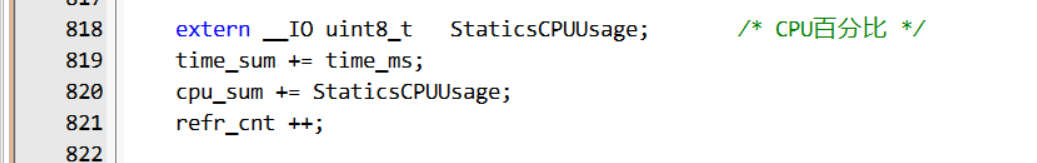


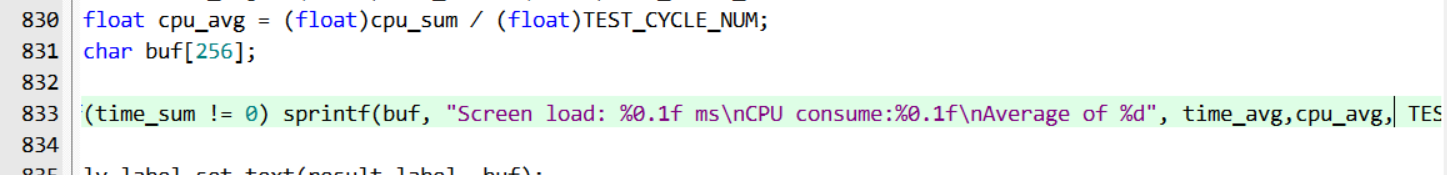
速度再次提升。

首先我们修改benchmark程序，使其能够显示CPU平均利用率：

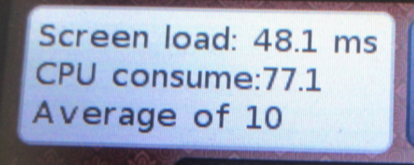








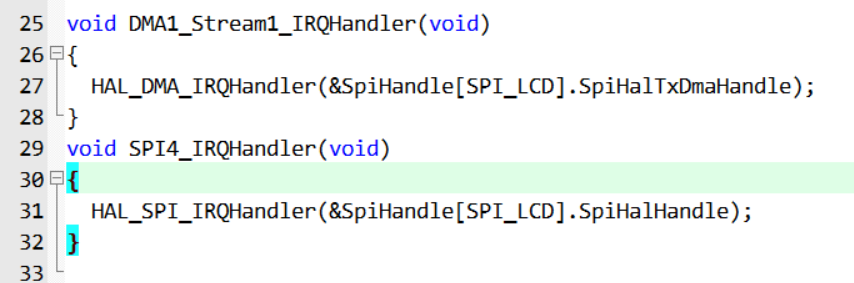
编译下载测试：



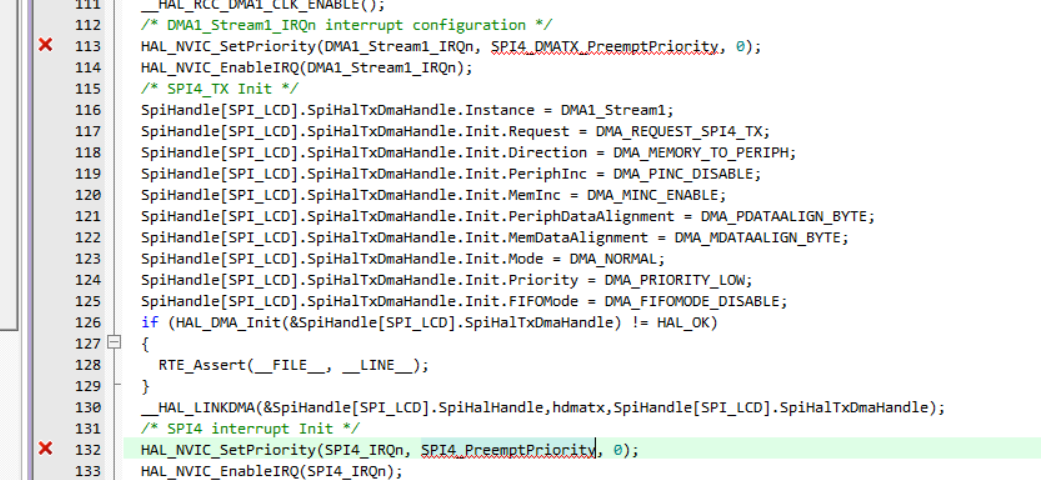
可以看到，特效全开下，未开启DMA前CPU占用率高达百分之77。

做如下修改：

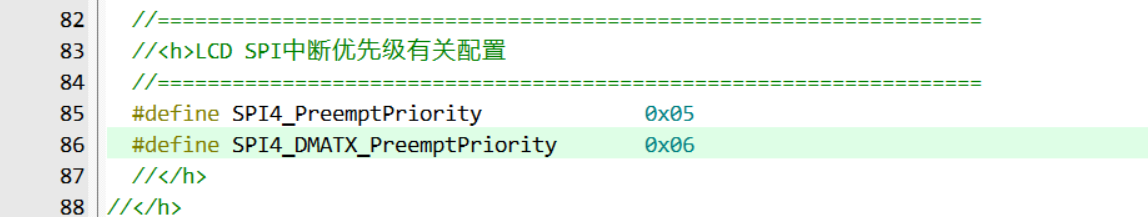
拷贝stm32h7xx\_it.c中的DMA1\_Stream1中断和SPI4中断到Board\_Interrupt中，略加修改：



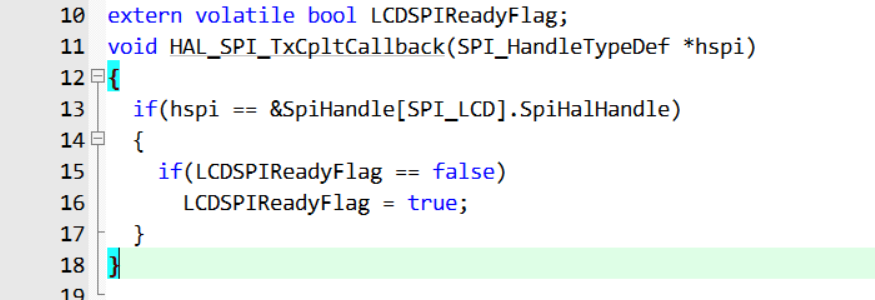
拷贝spi.c中HAL\_MSP\_Init相关内容到Board\_MSPInit.c文件中，略加修改：



在Board\_Confg.h中添加两个宏定义：

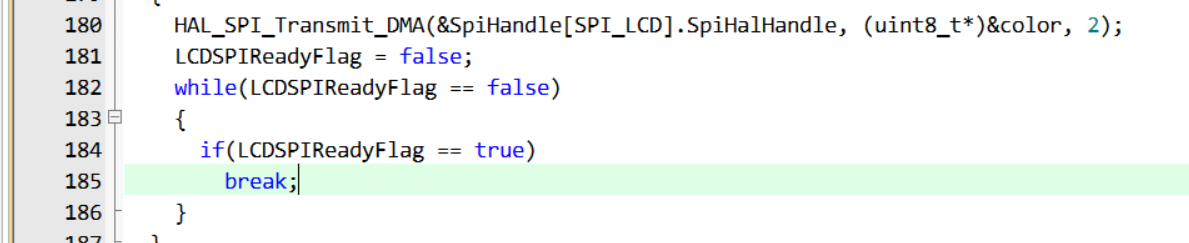


如此即完成了初始化的配置，然后类似串口，我们添加中断回调函数：



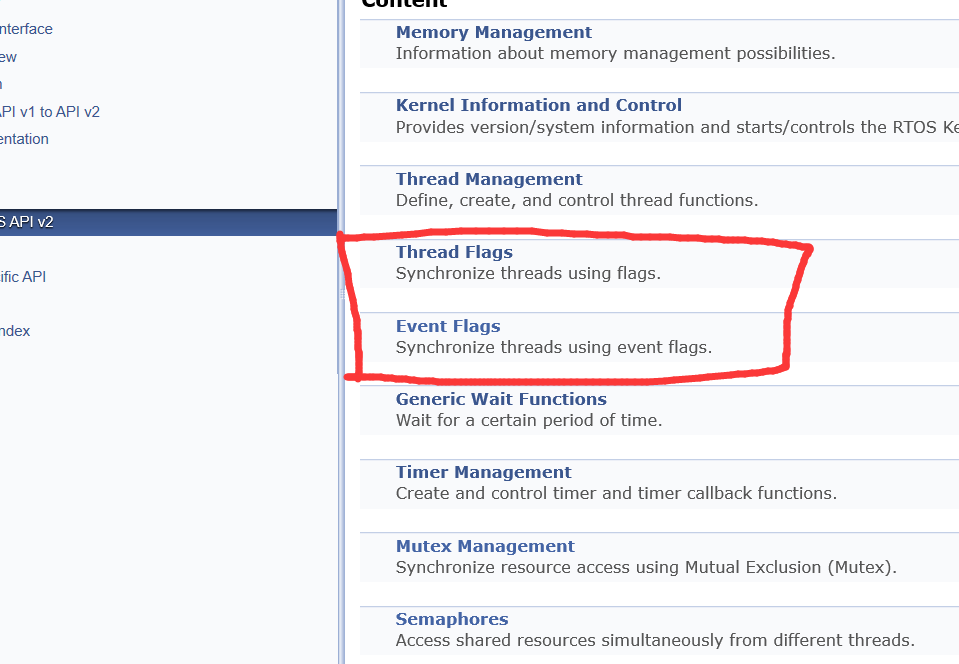
这里说一下为什么要这么做，在没有使用操作系统接口前，我们需要测试DMA的功能。在配置完DMA之后，CPU是可以去做其他事情的，因为没有使用操作系统接口，所以我们依然死等一个标志位，当SPI发送完成，会产生一个中断，这个中断意味着发送完毕，我们在这里置位如上标志位。

然后修改LCD驱动如下：



编译下载测试，哈哈CPU占用率不会变的。

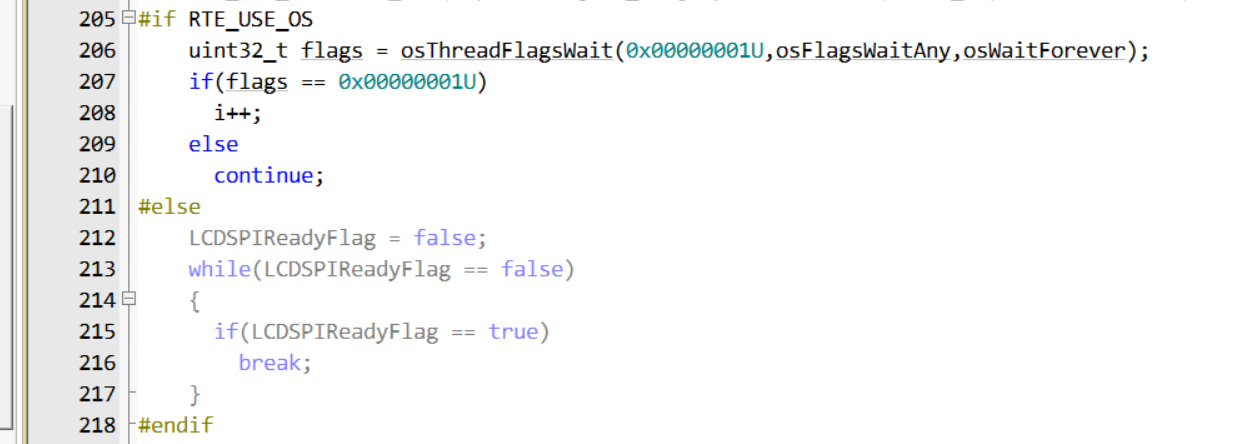
那么，如何在完成DMA配置后让CPU去做其他事情然后发生中断后再回来处理呢？这里就涉及到操作系统调度了，我们打开CMSIS\_RTOS2的文档：

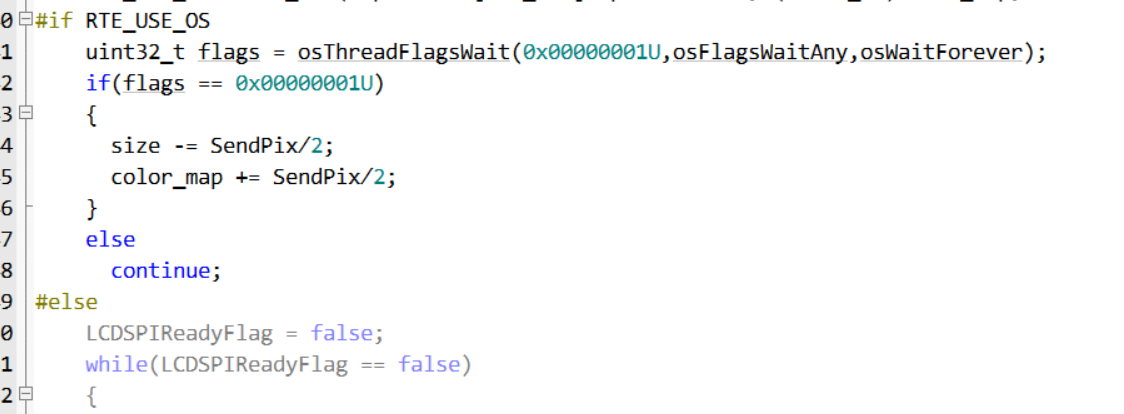


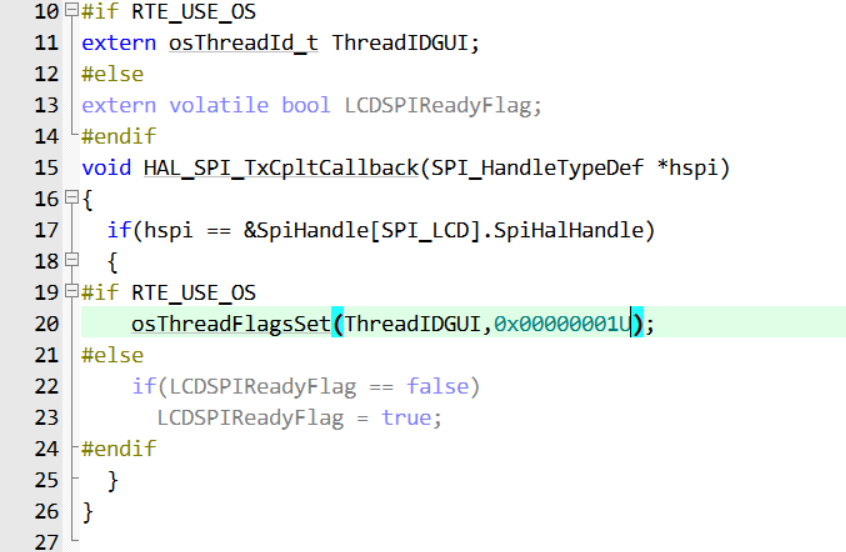
如图所示，RTOS的很多模块包括线程标志位、事件标志位、互斥量、信号量都可以用来同步线程，我们这里使用线程标志位以及事件标志位，首先我们讲解线程标志位：

线程标志位类似事件标志位，事件标志位用于几个线程之间的全局信号，线程标志位用于单一线程。每个线程可以收到自己的线程标志位，不需要定义新的内容。

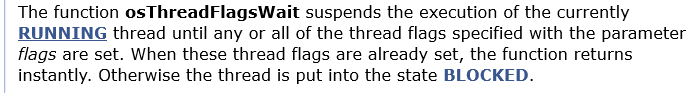
我们添加如下代码:



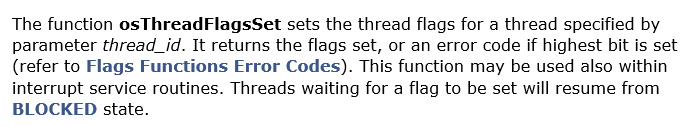




解释一下如上代码。在刷图的函数中，配置好DMA后就等待线程标志位，这时根据RTOS的手册：

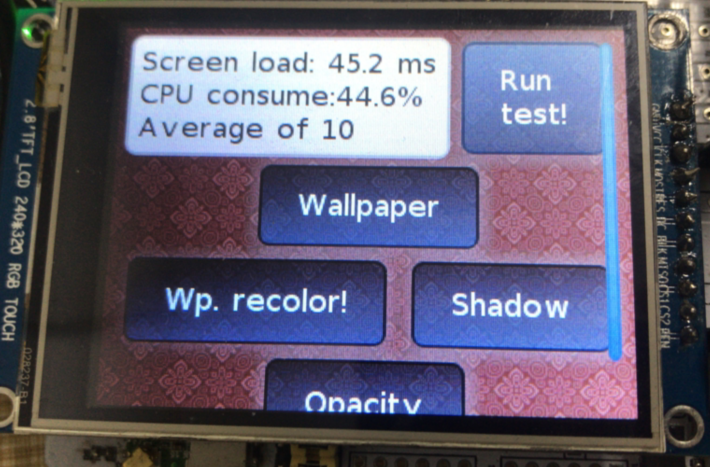


OsThreadFlagsWait会阻塞当前运行的线程，因此，其他线程会获取当前CPU占有权。当发送完毕后，中断中调用osThreadFlagsSet：



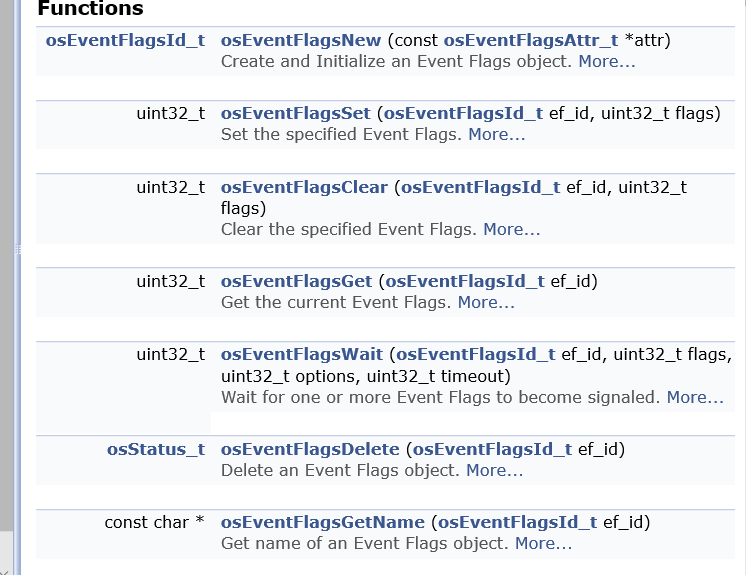
之前被阻塞的线程将被恢复。同时，如果等到的标志位是等待的标志位，将会增加下一轮DMA搬运的地址，否则，重复之前的搬运。

编译下载测试：



可以看到CPU占用率大大下降了。

类似地，我们可以用事件标志位实现：



不同的是，我们需要先创建一个事件标志位。考虑到实际调用绘图函数的线程只有GUI线程，所以使用线程标志位代替事件标志位。

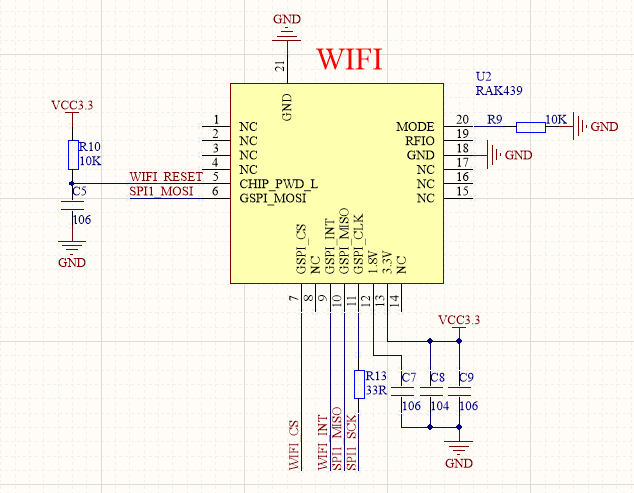
额外的优化：

在之前实现的刷颜色函数中，我们首先申请了内存，向其填充颜色，然后在释放，实际上在使用DMA搬运的过程中，我们可以配置DMA为内存不自增模式，这样就无需申请内存填色与释放了，但是在刷颜色末尾需要再次配置DMA为内存自增，因为在刷色块函数中需要DMA为自增模式，感兴趣的小伙伴可以自己尝试。

5、WIFI模块驱动的完成（建议直接使用demo里的驱动）

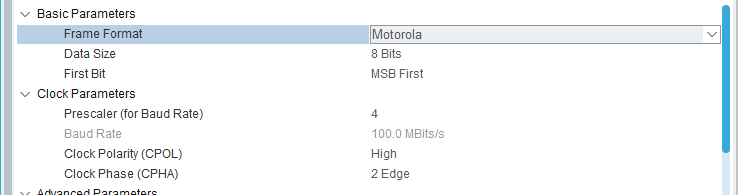


开发板上所使用的WIFI模块是RAK439，其跟LCD类似，也是用SPI总线通信的模块。其与MCU连接的原理图：

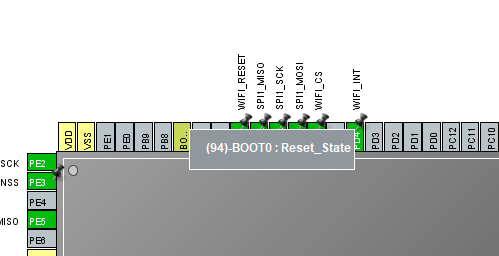


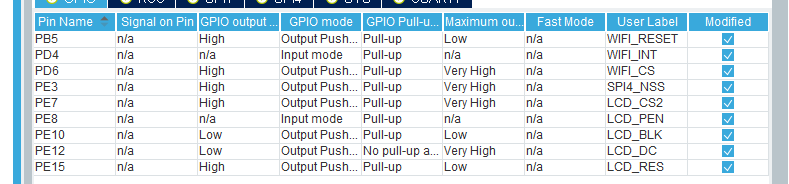
 

类似LCD我们现在CubeMX中完成SPI1的配置，需要注意的是，WIFI的SPI的CLK触发方式跟LCD的完全相反：



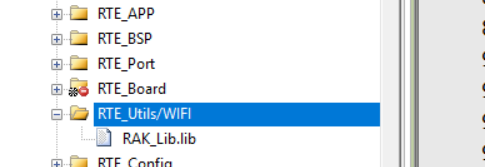
然后按照之前的教程完成GPIO的配置：





然后生成代码编译下载。

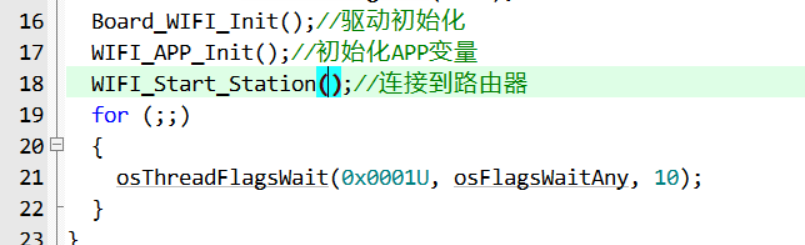
接下来我们使用RAK439提供的库，将其加入工程：



编译一下。

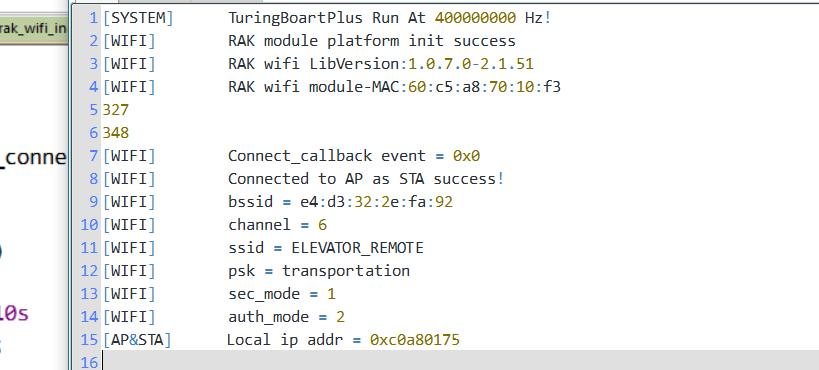
然后打开RAK439提供的文档与demo开始移植（资源见Document路径），我们这里不多赘述，直接提供移植好的驱动给大家测试：（如果有移植的小伙伴，请仔细阅读RAK提供的移植手册，同时需要注意其提供的库函数分os版和裸机版本，我们使用的是OS版）

完成驱动后，一样类似LCD，先在main中调用SPI初始化，然后建立WIFI线程。



每个人需要根据自己要连接的WIFI路由器信息修改rak\_wifi\_ap\_sta中的WIFI\_Start\_Station函数的有关内容。

编译下载后，串口输出相关信息：

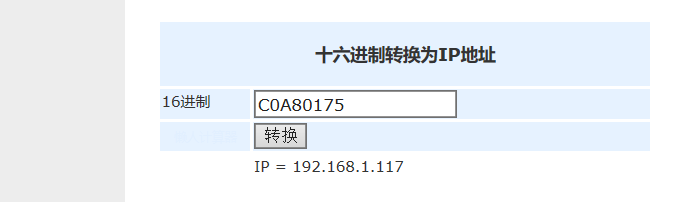


可以看到，最后输出了模块获取的IP地址，这里我们打开自己的路由器的管理页面：



可以看到，已经显示有模块连接到了路由，同时MAC地址一致。

我们打开一个16进制IP转换网页：



可以看到WIFI模块获取的IP跟路由器分配给他的IP是一致的。