Lilto

STM32F103_外部RAM用作运存

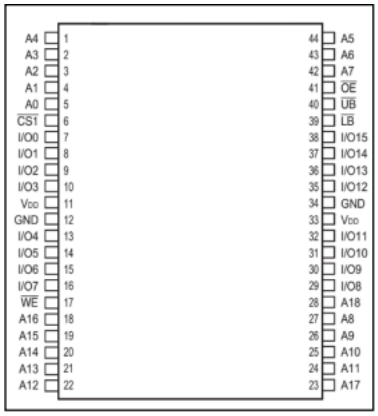
概述

SRAM的简介

折腾过电脑的朋友都知道，当电脑运行比较卡的时候，我们可以通过给电脑加装内存条来改善电脑的性能。那么号称微型计算机的单片机能不能像电脑一样加装内存条呢？装内存条倒是不行，但是我们可以给单片机外加和内存条效果一样的SRAM来提升单片机的性能。下面以STM32F407ZGT6单片机来讲解一下来扩展外部SRAM。

原理：给STM32芯片扩展内存与给PC扩展内存的原理是一样的，只是PC上一般以内存条的形式扩展，内存条实质是由多个内存颗粒(即SRAM芯片)组成的通用标准模块，而STM32直接与SRAM芯片连接。

SRAM，型号IS62WV51216，管脚图如下：

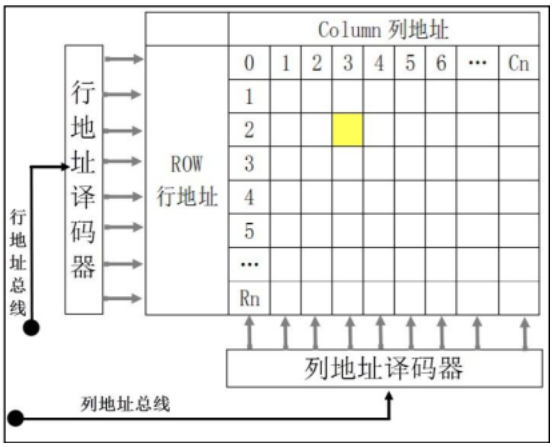


IS62WV51216的管脚总的来说大致分为：**电源线、地线、地址线、数据线、片选线、写使能端、读使能端和数据掩码信号线。**

信号线	类型	说明
A0-A18	I	地址输入
I/O0-I/O7	I/O	数据输入输出信号，低字节
I/O8-I/O15	I/O	数据输入输出信号，高字节
CS 和 CS1#	I	片选信号，CS 高电平有效，CS1#低电平有效，部分芯片只有其中一个引脚
OE#	I	输出使能信号，低电平有效
WE#	I	写入使能，低电平有效
UB#	I	数据掩码信号 Upper Byte，高位字节允许访问，低电平有效
LB#	I	数据掩码信号 Lower Byte，低位字节允许访问，低电平有效

从这个图中我们可以看出IS62WV51216有19根地址线和16根数据线，从这些数据中我们可以分析出IS62WV51216的存储大小为1M，那么这个1M是怎么分析出来的呢?

我们得来说说IS62WV51216的存储原理。首先，我们来谈一谈一般的SRAM的存储原理：



sram的存储模型我们可以用矩阵来说明：

SRAM内部包含的存储阵列，可以把它理解成一张表格，数据就填在这张表格上。和表格查找一样，指定一个行地址和列地址，就可以精确地找到目标单元格，这是SRAM芯片寻址的基本原理。这样的每个单元格被称为存储单元，而这样的表则被称为存储矩阵。地址译码器把N根地址线转换成2的N次方根信号线，每根信号线对应一行或一列存储单元，通过地址线找到具体的存储单元，实现寻址。如果存储阵列比较大，地址线会分成行和列地址，或者行、列分时复用同一地址总线，访问数据寻址时先用地址线传输行地址再传输列地址。

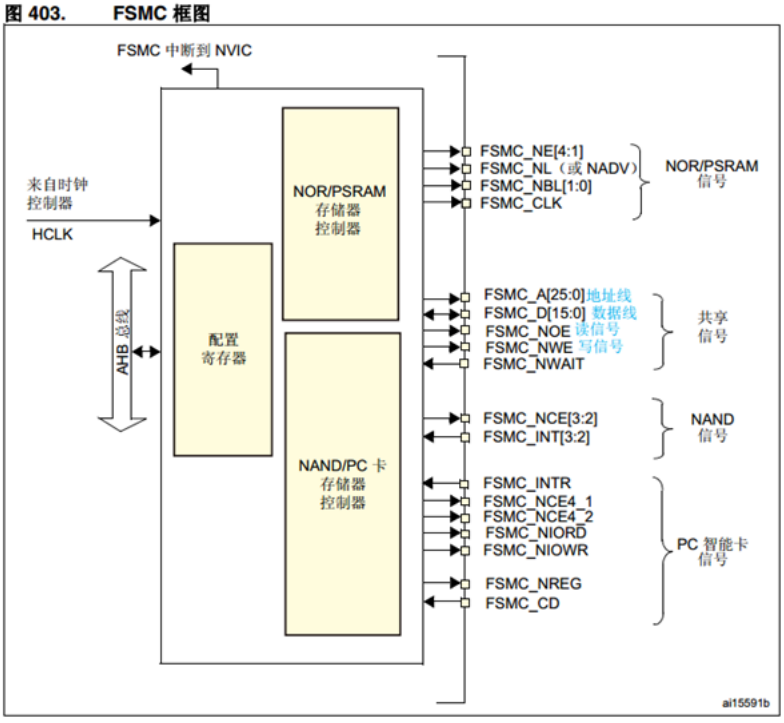
但是呢？你会发现，这个原理好像不太适用于IS62WV51216，为什么呢？

其实不然，因为我们使用的SRAM比较小，IS62WV51216没有列地址线。它只有19根行地址线，那么，我们就可以这么来解释：IS62WV51216有16根数据线，也就是说它的数据宽度为16位，一个行地址也就对应16位，即2字节空间。好，那现在来计算一下IS62WV51216有多少个行地址。2的19次方等于512K，在512K的基础之上在乘我们之前计算的2字节，不正好是1024K，也就是1M吗？1M后面的单位是B，即Byte，而不是Bit哦。这样的话你就会发现IS62WV51216这个名字中本身就包含了大量的信息：IS62WV51216共有512K个行地址，数据宽度为16位，再加以计算就可以得到它的存储大小为1M啦，有趣吧！

FSMC的简介

FSMC是Flexible StaticMemory Controller的缩写，就是灵活的静态存储控制器。它可以用于驱动包括SRAM、NOR FLASH以及NAND FLSAH类型的存储器。其他我们不用管，从上面我们可以总结的是，stm32雇佣FSMC这个管家来管理我们的IS62WV51216。来来来，我们来看看FSMC的庐山真面目：

图 403. FSMC 框图



蒙了吧！又是这么多信号线，不要怕，我们还是来总结归纳一下。我们FSMC控制SRAM为例来说明：通过查看STM32F103系列的参考手册：

2019年5月(5)
2018年9月(1)
2018年8月(3)

阅读排行榜
1. 建立时间和保持时间关系详
2. STM32F103_外部RAM用
03)
3. SPI协议解析(207)
4. printf的封装与实现(198)
5. 载域和运行域的理解（AR
么运行的）(197)

推荐排行榜
1. 建立时间和保持时间关系详

PSRAM/SRAM，非复用 I/O

表 192. 非复用 I/O PSRAM/SRAM

FSMC 信号名称	I/O	功能
CLK	O	时钟（仅用于 PSRAM 同步突发）
A[25:0]	O	地址总线
D[15:0]	I/O	数据双向总线
NE[x]	O	片选，x = 1..4（在 PSRAM 应用中被称作 NCE（Cellular RAM，即 CRAM））
NOE	O	输出使能
NWE	O	写入使能
NL(= NADV)	O	仅用于 PSRAM 输入的地址有效信号（存储器信号名称：NADV）
NWAIT	I	PSRAM 发送给 FSMC 的等待输入信号
NBL[1]	O	高字节使能（存储器信号名称：NUB）
NBL[0]	O	低字节使能（存储器信号名称：NLB）

低电平有效

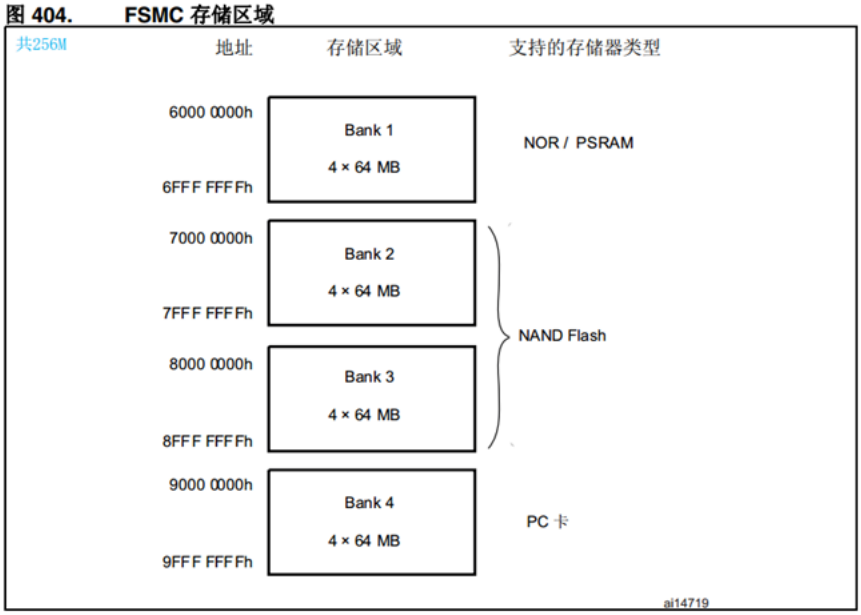
你会发现居然和SRAM中的线居然高度统一（那是当然啫，我们就是讲的FSMC嘛！）

1. **FSMC_NBL[1:0]分别对应于LB#、UB#，有什么用呢？提供数据掩码信号。具体是怎么回事呢？还记得前面提到的行地址线吗？**

   一根行地址线对应16位的数据，我们可以把16位的数据分为高字节和低字节。当要访问宽度为16位的数据时，使用行地址线指出地址，然后把UB#和LB#线都设置为低电平（FSMC_NBL0和FSMC_NBL1为低电平），那么I/O0-I/O15线（FSMC_D0到FSMC_D15）都有效，它们一起输出该地址的16位数据(或者接收16位数据到该地址)；当要访问宽度为8位的数据时，使用行地址线指出地址，然后把UB#（FSMC_NBL0）设置为低电平，I/O8-I/O15（FSMC_D8到FSMC_D15）会对应输出该地址的高8位，I/O0-I/O7的信号无效（或者把LB#（FSMC_NBL1）设置为低电平，I/O0-I/O7（FSMC_D0到FSMC_D7）会对应输出该地址的低8位，I/O8-I/O15的信号无效。这样是不是有一部分信号没有用呢？好像被掩盖了。因此它们被称为数据掩码信号。

2. **FSMC_NE[1:4]是个很有趣的东西，它决定了FSMC可以控制多个存储器。**这里就要提及FSMC的地址映射啦！

图 404. FSMC 存储区域



首先，有一点我们必须明白，对于32位的stm32单片机来说，它能够管理的地址大小为4GB，而stm32将4GB的地址空间中的0x60000000到0x9FFFFFFF共1GB的空间分给外部内存，所以这1GB的空间就成了我们的小天地，供我们自由玩耍。然后强势的FSMC就接管了这1GB的空间，FSMC将图中的1GB大小的External RAM存储区域分成了4个Bank区域，每个Bank对应于stm32内部寻址空间的不同地址范围。那么为什么要分为不同的Bank区域呢？因为不同的Bank可以来管理不同的外部存储设备，比如NOR Flash及SRAM存储器只能使用Bank1的地址，NAND Flash存储器只能使用Bank2和Bank3的地址,等。

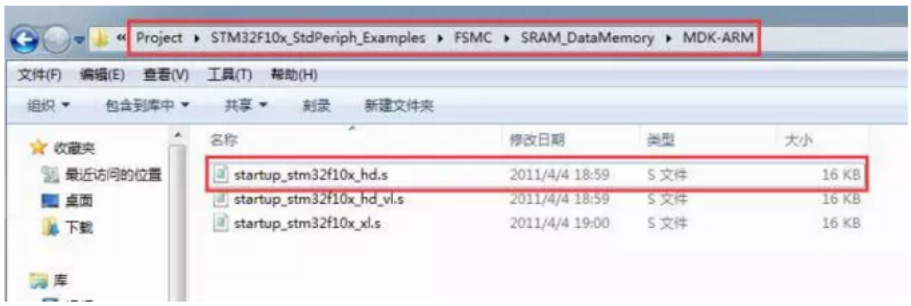
细心的你肯定还会发现，每个Bank中居然还有4x64MB这种文字，这是什么意思呢？

Bank内部的256MB空间又被分成4个小块，每块64M，各自有相应的控制引脚用于连接片选信号。FSMC_NE[4:1]信号线就分别对应图中的FSMC bank1 NOR/PSRAM4到FSMC bank1 NOR/PSRAM1。当STM32访问0x6C000000-0x6FFFFFFF地址空间时，会访问到Bank1的第3小块区域：FSMC bank1 NOR/PSRAM3相应的FSMC_NE3信号线会输出控制信号（即片选信号），如果这个时候FSMC_NE3处刚好接上IS62WV51216的CS端，那么IS62WV51216就可以任由我们摆布啦。因此，对于你使IS62WV51216来说，一定要注意你的CS端是接的FSMC的哪个FSMC_NE端，这决定你在程序访问哪个地址范围。

下面来说一下在stm32F407中SRAM的硬件连接：对于FSMC来说，它已经集成到了单片机内部，它的提供给的管脚已经确定了，是不能改动的，这个可以参考STM32对应芯片的Datasheet。唯一具有灵活性的就是FSMC_NE，具体用哪个FSMC_NE管脚来和你的SRAM相连，当然是你的自由，但是不要忘了，你要找到你选的FSMC_NE所对应的地址范围，不然写程序的时候就搞不清啥！

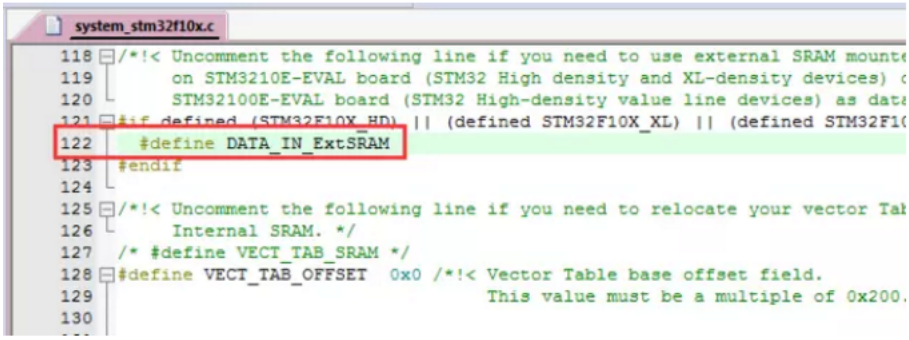
一、配置启动文件

我们使用官方标准库，拷贝标准库FSMC例程里面的"startup_stm32f10x_hd.s"文件（工程使用103ZE，若使用互联型芯片拷贝对应文件），替换掉我们之前工程的启动文件，如下图：



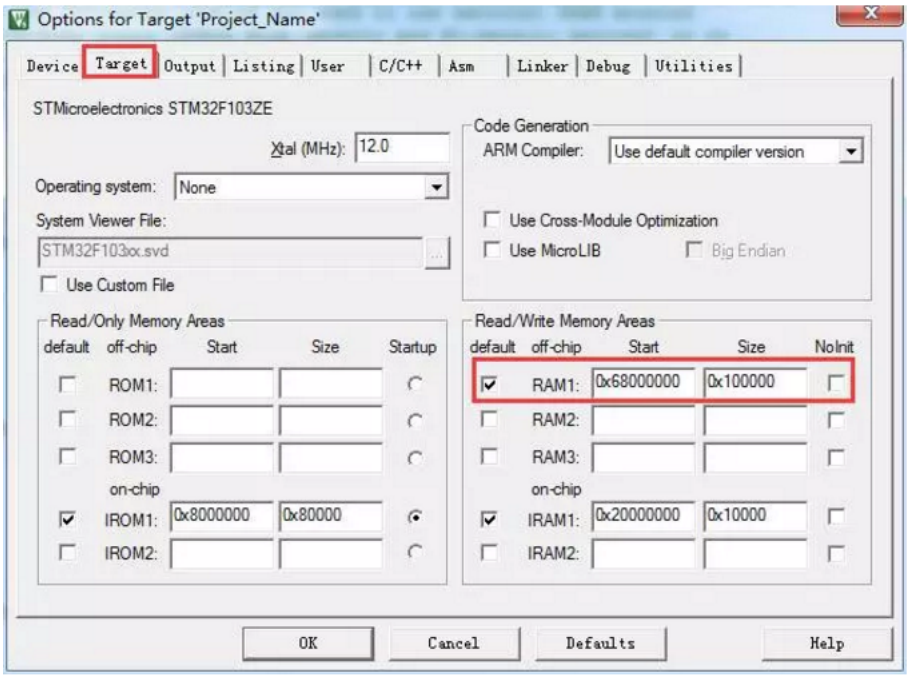
二、配置FSMC

我们使用官方标准库里面"system_stm32f10x.c"文件里面现成的函数接口(使用寄存器配置)来配置FSMC，只需要打开"system_stm32f10x.c"文件里面第122行的宏"DATA_IN_ExtSRAM"，见下图：



三、分配RAM

RAM地址的分配是由编译器完成的，因此需要对工程进行相应配置，就是使用外部RAM,见下图：



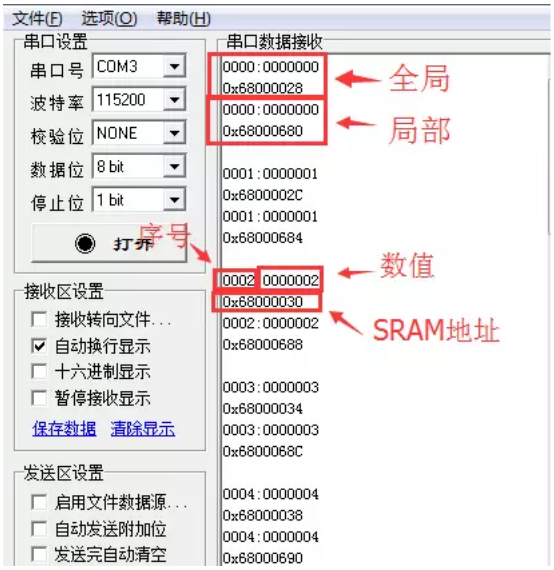
四、测试函数说明

```
#define BUF_LENGTH              256             //256*4 = 1024
uint32_t Global_Buf[BUF_LENGTH];                //全局BUF

/*********************************************
函数名称 :  SRAM_Check
功    能 :  对SRAM进行检测
参    数 :  无
返 回 值 :  无
作    者 :  strongerHuang
*********************************************/
void SRAM_Check(void)
{
  uint32_t i;
  uint32_t Local_Buf[BUF_LENGTH];               //局部BUF

  for(i=0; i<BUF_LENGTH; i++)                   //填充数值
  {
    Global_Buf[i] = i;
    Local_Buf[i] = i;
  }

  for(i=0; i<BUF_LENGTH; i++)
  {
    USART1_PrintSequence(i);                    //打印(全局)序号、BUF值、BUF地址
    USART1_PrintNumber(Global_Buf[i]);
    USART1_PrintSRAMAddr((uint32_t)&Global_Buf[i]);   全局

    USART1_PrintSequence(i);                    //打印(局部)序号、BUF值、BUF地址
    USART1_PrintNumber(Local_Buf[i]);
    USART1_PrintSRAMAddr((uint32_t)&Local_Buf[i]);    局部

    LED_TOGGLE;
    TIMDelay_Nms(100);                          //延时
  }
}
```

该函数位于main.c文件下面；

这个函数主要就是对上面配置及整改工程的测试。定义一个全局变量和一个局部变量，通过串口打印出他们的地址就可以判断运行内存是使用外部还是内部。

五、打印(测试)结果



看了测试函数就知道依次打印出来的数据是什么，这里我们很明显的可以看到打印出的地址是0x6800xxxx，这里的0x6800xxxx地址数据就是外部SRAM地址(不懂的话，请看昨天的讲解)，说明运行内存确实是外部SRAM.

六、变量定位定义

对于一个使用单片机内部 RAM 的访问相当容易，基本上定义变量是不需要思考其定位问题的，当把外部 SRAM 考虑进来时，则需要考虑内部及外部的问题；比如，如何让一个变量定位在内部或者外部 RAM；定位于内部是如何访问，定位于外部时又如何访问。这里说的是一个变量，或者一个数组的定位问题，当涉及到一个文件或者多个文件其内部所有变量的定位问题就复杂得多了。变量定位定义的一般方法（使用__attribute__）。

一般的定义方法如下图：

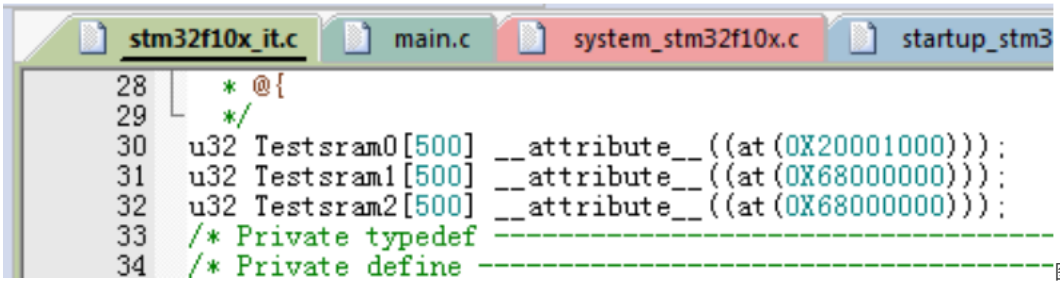


图1

定义了 3 数组（属于公共变量），现在检查下对应的 map 文件如下图所示：

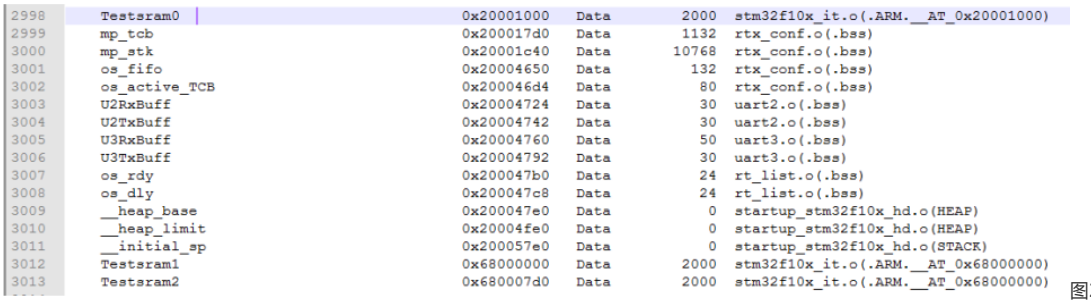


图2

如上图的 2998 行 3012 行 3013 行可见与上面定义的位置是对应的，所以这样实现了变量定义的定位功能；当内部 RAM 不足时或者有意定义一个变量定位到外部 RAM 中就可以采用这种方法（使用外部 RAM 有前提条件这里就不说了）。

七、批量定义变量到外部 SRAM

如何实现批量的变量定义到外部 RAM 呢？除了批量地使用__attribute__定义变量，还是有更快捷的方法的。

配置外部 SRAM 可用起始地址及大小一如下图：

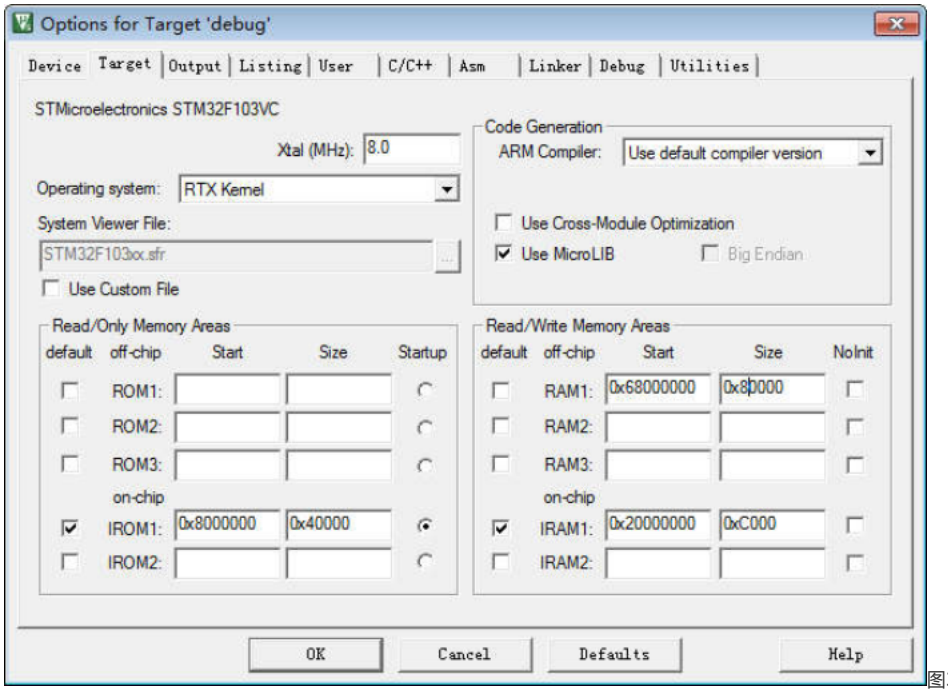


图3

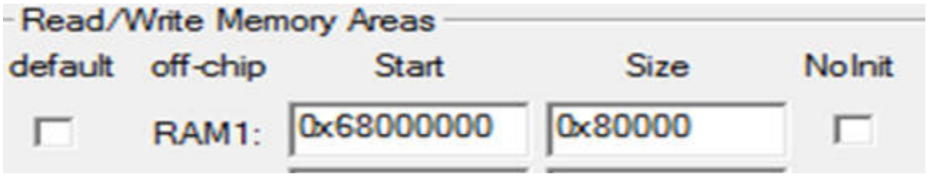


图4

如上图 所示，最左最右边的小方框不要打勾...千万不要打勾，开始地址及大小必须如实填写（Size 的值可以小于但绝对不能大于实际外置芯片值），开始地址安装原理图连线确定其值。

八、定义一个文件内的所有变量于外部SRAM

首先定一个小目标：确定你要一个所有变量需要定位于外部 SRAM 的文件，接着按照下图来配置（这里让 main.c 这个文件，让里面定义的所有变量均定位于外部SRAM 中）在工程窗口选择 main.c 点鼠标右键如下图：

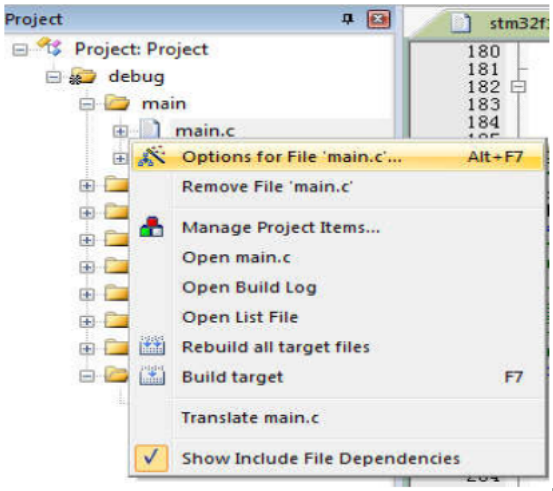


图5

菜单选择第一行"Options for File 'main.c' "之后显示如下图：

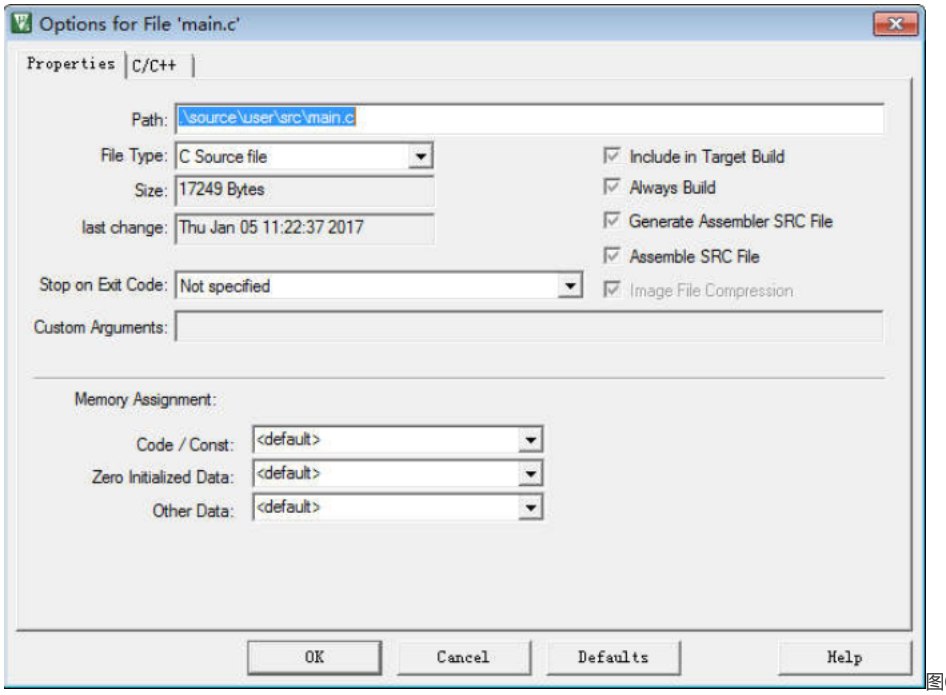


图6

如上图我们仅需关注"Memory Assignment"组，Code/Const 定义代码及 const 的定位，Zero Initialized Data 及 Other Data 是变量的定位，此处我们关心的仅数据（变量）部分的定位。定位设置如下图：

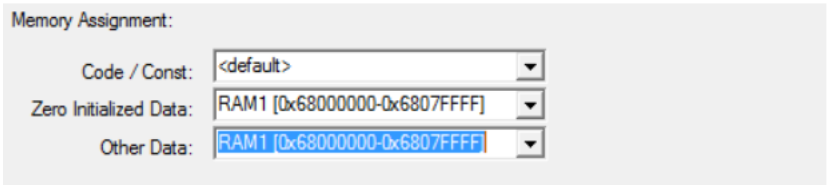


图7

至此，main.c 文件内部的所有变量均已定位到外部 SRAM 中（前提是没忘记点 OK 按钮），到这里应该会发现一个问题如图 7 每一项都有一个<default>选项；在 keil 的工程里每个文件的变量安排都会有一个默认选项，当这里选择<default>时则会启用如图3 所示的默认选项，可以看到图 4 那里，说到千万不要打勾的那里。打勾的话那里就变成了第一默认选项，那么图 7 的配置就多余了。这个是可以验证的。那么再回头验证一下图 7 的配置是否实现了将 main.c 文件中的变量定位到外部 SRAM.....同样查看MAP 文件验证一下。首先在 main.c 中定义变量如下图：

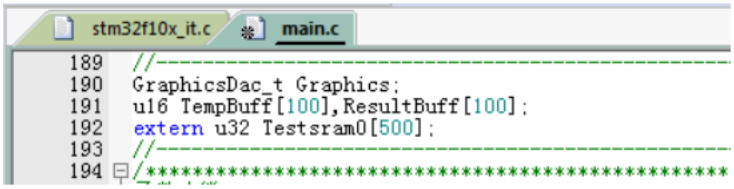


图8

如上图所示定义了一个结构体实例，两个 16 的数组，对应 map 文件如下图：

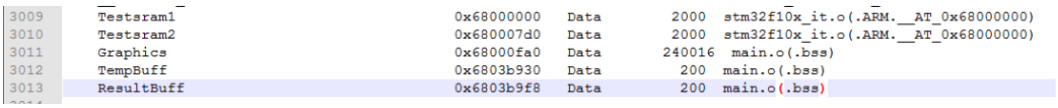


图9

如上图所示， 3009 行 3010 行使用__attribute__定义占用了外部 SRAM的部分地址，《main.c》文件中并没有使用__attribute__来定义变量，其定位也处于外部 SRAM 中，其数组的大小定义与 map

文件是一致的，在此证明经过配置之后，在 main.c 文件中定义的变量均会被定位到外部 SRAM 中。再验证 GraphicsDac_t 的大小。printf(" GraphicsDac_t size=%d \r\n",sizeof(GraphicsDac_t));

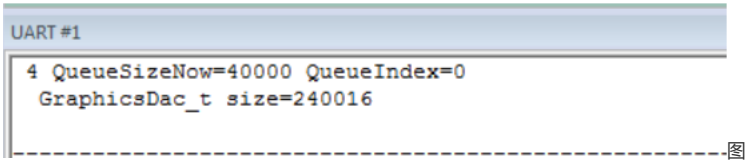


图 10（可见 printf 的大小与 map 的大小是一致的）

九、将变量定义到内部SRAM

参考图 3，将内部 SRAM 的《default》打勾，之后将图 7 的下两个选项配置为《default》即可，这样实现一个文件其变量定位的切换。多个外部 SRAM 芯片时适当参考配置。

十、定位到外部 SRAM的变量的访问方法

方法一、一般访问外部 SRAM 的方法

首先使用 SRAM_Init();之后使用下面两个函数读写外部 SRAM：

SRAM_ReadBuffer(uint16_t* pBuffer, uint32_t ReadAddr, uint32_t NumHalfwordToRead)

SRAM_WriteBuffer(uint16_t* pBuffer, uint32_t WriteAddr, uint32_t NumHalfwordToWrite)

方法二、编译器

如第 3 节变量定位定义方法，变量的访问就由编译器自己搞定了(关于这一点还没有实际硬件验证---这里仅是理所当然的推测，至于还要使用方法一是不可思议的)，变量的读写就和内部变量一样操作。

备注：文件内部函数内部的变量被定义在堆栈里，同时说明上面所说的定位到外部 SRAM 的变量均为全局变量，其文件内的局部变量还是在堆栈里（此处为我的推测，没经过验证）。

分类： STM32学习笔记

« 上一篇： 载域和运行域的理解（ARM程序是怎么运行的）
» 下一篇： 建立时间和保持时间关系详解

posted @ 2018-08-28 15:51 Lilto 阅读(5403) 评论(0) 编辑 收藏