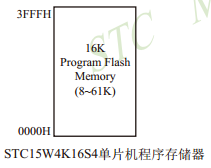
**STC15系列单片机存储器**

STC15系列单片机的程序存储器和数据存储器是**各自独立编址**的。STC15系列单片机的所有程序存储器都是片上FLASH存储器，不能访问外部程序存储器，因为没有访问外部程序存储器的总线。

1. **程序存储器**

程序存储器用于存放用户程序、数据和表格等信息。如STC15W4K16S4系列单片，其内部集成了16K字节的FLASH程序存储器，其地址空间如下图所示：



**单片机复位后，程序计数器(PC)的值为0000H**。从0000H单元开始执行程序。另外中断服务程序的入口地址(也称中断向量)也位于程序存储器单元。

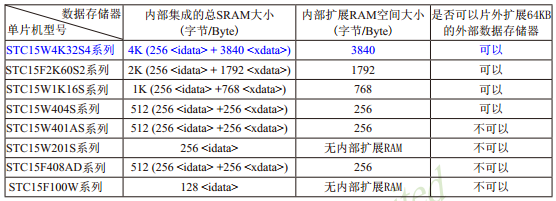
程序FLASH存储器可在线反复编程擦写10万次以上。

1. **数据存储器**

STC15系列单片机的**数据存储器在物理上和逻辑上都分为两个地址空间：内部RAM和内部扩展RAM**。其中内部RAM的高128字节数据存储器与特殊功能寄存器(SFRs)共用相同的地址范围，实际使用时通过不同的寻址方式加以区分。如STC15W4K32S4系列单片机内部集成了4096字节RAM内部数据存储器，其在物理和逻辑上都分为两个地址空间：内部RAM(256字节)和内部扩展RAM(3840字节)。

另外，STC15系列单片机40-pin及以上的单片机还可以访问片外扩展的64KB外部数据存储器。

下表总结了STC15系列单片机内部数据存储器的空间大小以及是否可以扩展片外数据存储器：

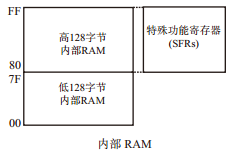


STC15系列单片机内部集成的RAM可用于存放程序执行的中间结果和过程数据。

**2.1、内部RAM (idata，不包含内部扩展RAM)**

内部RAM共256字节，可**分为3个部分：低128字节RAM(为了与传统8051兼容)、高128字节RAM(Intel在8052中扩展了高128字节RAM)、特殊功能寄存器区**。低128字节的数据存储器既可直接寻址也可间接寻址。高128字节RAM与特殊功能寄存器区共用相同的地址范围，都使用80H~FFH，地址空间虽然重叠，但是物理上却是独立的，使用时通过不同的寻址方式加以区分：高128字节RAM只能间接寻址，特殊功能寄存器区只可直接寻址。

内部RAM的结构如下图所示，地址范围是00H~FFH：



**2.1.1、内部低128字节RAM**

低128字节RAM也称通用RAM区。通用RAM区又可分为工作寄存器组区、可位寻址区、用户RAM区和堆栈区。

**A、工作寄存器组区：**

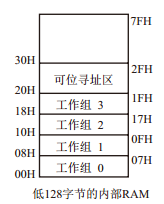
地址从00H~1FH共32字节单元，分为4组，每一组称为一个寄存器组，每组包含8个8位的工作寄存器，编号均为R0~R7，但属于不同的物理空间。通过使用工作寄存器组，可以提高运算速度。提供4组是因为1组往往不够用。程序状态字PSW寄存器中的RS1和RS0组合决定当前使用的工作寄存器组。

**B、可位寻址区：**

可位寻址区的地址从20H~2FH共16个字节单元。20H~2FH单元既可像普通RAM单元一样按字节存取，也可以对单元中的任何一位单独存取，共128位，所对应的地址范围是00H~7FH。位地址范围是00H~7FH，内部RAM低128字节的地址也是00H~7FH，从外表看，两者地址是一样的，实际上二者具有本质区别：位地址指向的是一个位，而字节地址指向的是一个字节单元，在程序中使用不同的指令区分。

**C、用户RAM区和堆栈区：**

内部RAM中的30H~FFH单元是用户RAM和堆栈区，一个8位的堆栈指针(SP),用于指向堆栈区。单片机复位后，堆栈指针SP为07H，指向了工作寄存器组0中的R7，因此，用户初始化程序都应对SP设置初值，一般设置在80H后的单元为宜。startup.a51文件会设置SP指针。



**2.1.2、内部高128字节RAM**

**A、特殊功能寄存器(SFRs):**

特殊功能寄存器(SFRs)是用来对片内各功能模块进行管理、控制、监视的控制寄存器和状态寄存器，是一个特殊功能的RAM区。STC15系列单片机内部RAM的高128字节数据存储器与特殊功能寄存器(SFRs)共用相同的地址范围，都使用80H~FFH，特殊功能寄存器必须用直接寻址指令访问。

**2.2、内部扩展RAM(xdata)**

**2.3、片外扩展RAM(xdata)**

使用51单片机内部扩展RAM时，同时存在片外RAM和内部扩展RAM，地址是重叠的。也就是说如果我们假设：

单片机内部有1K的内部扩展RAM，同时外部IO口上接有2K的片外RAM，那么实际上外部的2K RAM中前1K是不会被使用的，因为访问前1K的语句被执行到访问内部扩展RAM里了。（当然前提是初始化部分按照实际资源设置，而不是强制设定内部扩展RAM为0，设置部分参考一下keil里的startup文件），这种情况下单片机扩展RAM资源一共是2K而不是3K。这样就不难理解书上说的访问内部RAM和外部RAM用同一个方法了。可以访问外部RAM的IO口就不说了，看看接口定义就知道了。

在汇编语言里使用MOVX来表示访问扩展RAM部分，C语言在定义的时候使用xdata关键字，其实是一回事。51的外部访问地址空间是连续的，也就是说即使在IO口上挂接其他设备，这个设备也可以作为外部地址空间的一部分从而通过访问特定的地址来获取数据

在C语言中，可使用xdata声明数据的存储类型为内部扩展RAM，如”unsigned char xdata i=0;”

1. **data、idata、xdata、pdata、bit、bdata、code关键字**

**data**: 片内RAM直接寻址区，固定指内部数据存储器前面0x00-0x7f的低128个字节RAM，可以用ACC直接读写的，速度最快，生成的代码也最小。

**idata**：片内RAM间接寻址区，固定指前面0x00-0xff的256字节RAM，其中前128字节和data的128字节完全相同,只是访问的方式不同。idata是用类似C中的指针方式访问的。汇编中的语句为：mox ACC,@Rx.(不重要的补充：c中idata做指针式的访问效果很好)。

**xdata**: 64K片外RAM数据区(包括内部扩展RAM和片外扩展RAM)，一般指外部0x0000-0xffff空间,用DPTR访问。

**pdata**: 数据存储器外部扩展RAM的低256个字节,地址出现在A0-A7的上时读写,用movx ACC,@Rx读写。这个比较特殊,而且C51好象有对此BUG, 建议少用。但也有他的优点,具体用法属于中级问题,这里不提。

**bit**: 是在数据存储器内部低128字节存储空间中20H~2FH，可位寻址区域中一个位的地址。

**bdata**：片内RAM位寻址区，20H~2FH。

若程序需要8个或者更多的bit变量，如果你想一次性给8个变量赋值的话就不方便了，（举个例子说说它的方便之处，想更深入的了解请在应用中自己琢磨）又不可以定义bit数组，只有一个方法：

char **bdata** MODE;  
sbit MODE\_7 = MODE^7;  
sbit MODE\_6 = MODE^6;  
sbit MODE\_5 = MODE^5;  
sbit MODE\_4 = MODE^4;  
sbit MODE\_3 = MODE^3;  
sbit MODE\_2 = MODE^2;  
sbit MODE\_1 = MODE^1;  
sbit MODE\_0 = MODE^0;

8个bit变量MODE\_n 就定义好了。  
这是定义语句，Keil的特殊数据类型。记住一定要是sbit，不能bit MODE\_0 = MODE^0，赋值语句要是这么写C语言就视为异或运算。

**code**:code的作用是告诉单片机，我定义的数据要放在ROM（程序存储区）里面，写入后就不能再更改。如：unsigned char **code** table[3]。

1. **指针类型和存储区的关系**

对变量进行声明时可以指定变量的存储类型如:  
uchar data x和data uchar x相等价都是在片内RAM区分配一个字节的变量。

同样对于指针变量的声明，因涉及到指针变量本身的存储位置和指针所指向的存储区位置不同而进行相应的存储区类型关键字的使用，我们举例说明对应“\*”前后不同的关键字的使用在编译时出现什么情况。

首先我们在外RAM区开辟10个字节的内存空间，地址是外RAM的0x0000－0x0009地址：  
 uchar **xdata**  tmp[10];

然后定义指针变量指向它：  
第1种情况:  
 uchar **data\*** **data** pstr;  
 pstr = tmp;  
首先要提醒大家这样的代码是有bug的, 他不能通过这种方式正确的访问到tmp空间。 为什么？因为tmp是外空间RAM变量，访问外RAM需要2 byte来寻址64k空间，但因为使用data关键字(在"\*"号前的那个)，所以按KeilC编译环境来说，就把他编译成指向片内RAM的指针变量了，这也是初学C51的朋友们不理解各个存储类型的关键字定义而造成的bug。特别是当工程中的默认的存储区类为large时，又把tmp[10]声明为uchar tmp[10]时，这样的bug是很隐秘的不容易被发现。  
第2种情况:  
 uchar **xdata\* data**  pstr;  
 pstr = tmp;  
这种情况是没问题的，这样的使用方法是指在片内RAM分配一个指针变量("\*"号后的data关键字的作用)，而且这个指针本身指向xdata区("\*"前xdata关键字的作用)。这种情况应该是在这里所有介绍各种情况中效率最高的访问外RAM的方法了，请大家记住他。  
第3种情况:  
 uchar **xdata\* xdata** pstr;  
 pstr = tmp;  
这中情况也是对的，但效率不如第2种情况。  
这种方式一般用在片内RAM资源相对紧张而且对效率要求不高的项目中。  
第4种情况:  
 uchar **data\* xdata** pstr;  
 pstr = tmp;  
如果详细看了第1种情况的读者发现这种写法和第1种很相似，是的，同第1 种情况一样这样也是有bug的，但是这次是把pstr分配到了外RAM区了。  
第5种情况:  
 uchar \* **data** pstr;  
 pstr = tmp;  
大家注意到"\*"前的关键字声明没有了，这时pstr指针变量会用到3bytes空间了。大家会疑问为什么在前面的几种情况的pstr指针变量都用2 byte空间而到这里就用3 byte空间了呢？这是KeilC的一个系统内部处理，在KeilC中一个指针变量最多占用 3 byte空间，对于没有声明指针指向存储空间类型的指针，系统编译代码时都强制加载一个字节的指针类型分辩值。具体的对应关系可以参考KeilC的help中C51 User's Guide。  
第6种情况:  
 uchar \* pstr;  
 pstr = tmp;   
这是最直接最简单的指针变量声明，但他的效率也最低。这种情况很类似第5种和第3种情况的组合，既把pstr分配在外ram空间了又增加了指针类型的分辨值。

**注意**：在keil中可以指定变量和函数的绝对地址。

1. 指定变量的绝对地址：

unsigned char **idata**  myvar **\_at\_** 0x40; //将myvar变量的地址固定在RAM的0x40地址上

在 M51 文件中可以找到这麽一行：  
IDATA   0040H     0001H     ABSOLUTE      
表示有变量在 idata 的 0x0040 处绝对地址定位。

unsigned char **code**  myvar **\_at\_** 0x40; //将myvar变量的地址固定在ROM的0x40地址上

1. 指定函数的绝对地址：

在demo.c中编写两个函数myTest1和myTest2:  
     void myTest1(void) void myTest2(void)  
     { {  
        // Add your code here // Add your code here  
      } }

打开 Project -> Options for Target 菜单,选中 BL51 Locate 选项卡, 在 Code: 中输入:  
            ?PR?myTest1?DEMO(0x4000),?PR?myTest2?DEMO(0x4100)  
把函数 myTest1定位到程序区的0x4000处, 把函数 myTest2定位到程序区的0x4100处，再次编译就可以了。