第9章

“高楼大厦，各有功用”存储器结构及功能

章节导读：

从本章开始，我们就正式进入 STC8 系列单片机片内资源的学习了。考虑到学习脉络的顺序性和完整性，我们首先“拿下”单片机存储器资源内容，虽说这部分的知识非常“枯燥”，但是小宇老师构造了“宿舍楼”和“双峰教学楼”给大家，相信大家一定能轻松愉快地掌握。本章共分为 7 节。9.1 节正面讲解了存储器知识点的必要性，并不是“鸡肋”的存在；9.2 节带领大家回到了 8032 微控制器时代，“忆苦思甜”地感受下单片机存储器的发展与变化；9.3 节～9.5 节以“建楼”为故事引入讲解了 RAM 及 ROM 区域结构、功能及内部划分；9.6 节讲解了 Keil C51 环境中的常规存储器配置及相关选项卡功能；9.7 节通过两个基础项目讲解了 STC8 系列单片机存储器单元的操作方法、特殊参数及字节数据的简单处理。希望朋友们活学活用，快乐进阶。

9.1 存储器难道不是“鸡肋”知识点吗

从本章开始，我们就正式学习 STC8 系列单片机的片内资源了，之前的章节中我们接触过 RAM、ROM、EEPROM、IAP 和 ISP 等名词，这些名词都与存储器资源相关，所以片内资源篇的第一章就给大家讲清楚 STC8 系列单片机的存储器资源。

可能有不少朋友对本章的内容感到疑惑，因为除单片机原理书籍之外，很多单片机应用类书籍都不会单独去写单片机内部存储器的相关内容，单片机的开发往往都是在编程环境中写好代码，经过编译器的处理后直接“烧录”就行了，谁也不用去关心程序和数据是怎么放置的，我们只需要去关心程序代码即可。所以有不少朋友觉得存储器知识是“鸡肋”的（鸡肋的意思就是肉少骨多，吃的过程比较麻烦，食之无肉，弃之可惜，多形容没什么太大意义的事物）。但是小宇老师并不这样认为，就拿人体来说，我们健康的时候压根儿感觉不到心脏有什么太大的“作用”，等到高血压、心绞痛的时候才会发现这种基础“资源”的重要性。

光是嘴巴说意义不大，直接上例子吧！假设我是用 A51 语言编写 STC 系列单片机代码，我可能会在初学的时候产生一大堆的疑问，随便列举 3 个疑问如下。

疑问 1：使用数据传送类指令时，为啥要区分 MOV、MOVX 和 MOVC 等指令头，干吗要有“变址寻址”这种方法？“MOVC A，@A＋DPTR”的形式如何理解？不能直接跨界访问数据吗？

疑问 2：某程序源码的第一句话是“ORG 0000H；”，我知道 ORG 是一个伪指令，它可以让程序从 ORG 指令指定的地址处开始执行。我也知道 0000H 表示一个十六进制地址，但是 0000 这个地址在哪儿？是在 RAM 还是 ROM？为什么程序都要从这里开始？

疑问 3：有的编程者在初始化程序时非要写“MOV SP，#80H”这样的语句，我不理解的是 SP 是个堆栈指针，它的默认地址就在 07H。既然有个默认值，为什么那么多的编程者非要把它迁移到 80H 地址后面去呢？

有的朋友忍不住了，站出来和小宇老师说：老师啊，我根本就不打算学习汇编语言，你说的这些指令、地址、堆栈指针什么的太偏硬件底层了，现在有了 C51 语言之后谁还用 A51 语言写代码啊！别人 Keil C51 的开发环境已经在 C 编译器的调控下把程序优化得很好了，程序存储压根儿不用我操心，我们就算没有存储器的知识也能把单片机玩得“飞起来”。所以我认为这本书添加这个“存储器”章节确实有点儿“鸡肋”。

好！朋友们说得非常在理，Keil C51 环境做得确实很好，特别值得一提的是，Keil C51 内部的编译器确实能合理分配相关资源，用户最关心的问题可能是“程序是否装得下”而不是“程序是怎么装下的”。长久的学习习惯让我们忽略了存储器资源的相关内容，感觉不到存储器的存在，但是这样的学习还是会有问题的，若是不信，我们再来看 4 个疑问。假设我们现在不用 A51 语言，换成 C51 语言来编程，看看下面的疑问你能否解释。

疑问 1：我是 51 初学者，我看别人 C51 程序中写了句“#pragma COMPACT”，然后紧接着写了“unsigned char xdata i;”和“unsigned char code NUM[10];”，这三句话中的“#pragma”“COMPACT”“xdata”“code”是什么意思呢？我查了一遍，它们不属于 C51 数据类型啊！在标准 C 语言中也没有这些关键字啊！

疑问 2：我是刚入职的初级技术员，公司前辈遗留了一个项目给我，打开工程后看到了很多类似于“unsigned char data COMBUF[8] _at_ 0x20;”这样的语句，这句话中的“data”“_at_”“0x20”是什么意思呢？要是去掉这 3 个东西我是能看懂的，这是在定义数组，但是加上这些字符后这个数组 COMBUF[8]去哪儿了呢？

疑问 3：我是学过 51 的，看了公司“大牛”写的一个中断服务函数“void Int0() interrupt 0 using 1”我就纳闷了，我知道“interrupt 0”是外部中断 1 的意思，但是这个“using 1”是什么？为什么我找了几本书都没有这样的写法？最奇怪的是 Keil C51 居然不报错？我自己改成“using 2”居然也没有什么问题，这个参数用来干嘛的？

疑问 4：我用 Keil C51 开发环境编写 51 程序都好几年了，但还是觉得不“踏实”。为啥呢？请看如图 9.1 所示的项目 Target 选项卡，这些黑色箭头指向的地方我居然都不明白是什么含义，显得我自己特别外行，我都不敢说我会用 Keil C51 开发环境了。

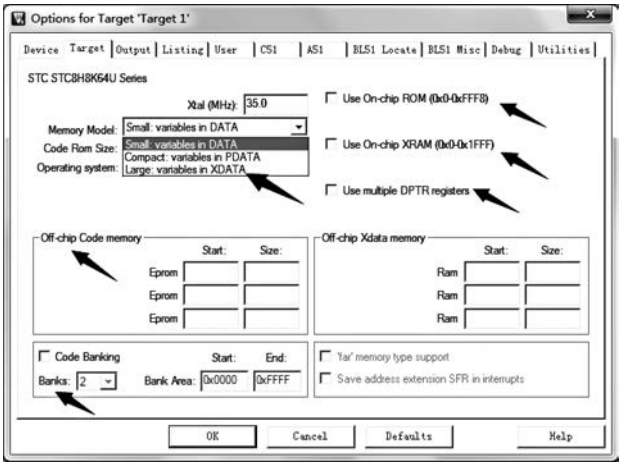


图 9.1　Keil C51 项目 Target 选项卡界面

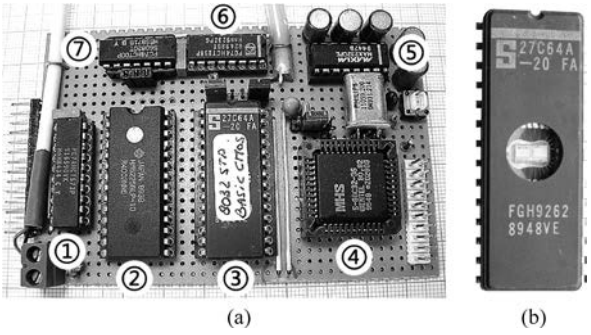
怎么样，现在心里舒服多了吧？是不是觉得存储器不那么“鸡肋”了？当然，也不要被小宇老师举的例子吓到了，单片机还是很简单的，毕竟这个内核是几十年前的，我们肯定能顺利地拿下相关知识点。

存储器结构其实就相当于去景点游玩时拿到的“地图”，假设我要去故宫游玩，最理想的参观路线是从午门进入紫禁城，然后沿着中轴线依次参观内金水桥、太和门、太和殿、中和殿、保和殿、乾清门、乾清宫、交泰殿、坤宁宫、御花园。参观完御花园，可以通过御花园左侧的门进入西六宫，依次参观储秀宫、翊坤宫、永寿宫、咸福宫、长春宫、太极殿，然后出内右门回到乾清门广场，东行进入内左门，可依次参观延禧宫、永和宫、景阳宫、乘乾宫、钟粹宫。参观完东六宫可沿东长安街再回到乾清门广场，向东穿过景运门进入锡庆门，然后再进入皇极门，可以参观皇极殿、宁寿宫、扮戏楼、畅音阁、养性殿、乾隆花园、珍妃井，最后出顺贞门西行出神武门离开故宫。咋样？听了小宇老师的讲解，是不是感觉头有点儿晕、腿有点儿抖？所以说，51 单片机的存储器结构要比故宫的“景点分布”简单太多，市面上的单片机存储结构能比“故宫”还复杂的几乎没有。经过本节的讨论，我和读者的认知应该一致了，接下来我们就可以快乐地学习存储器内容了。

9.2 让人“头疼不已”的 8032 微控制器时代

现在想想，STC8 系列单片机的存储资源较之传统的 51 单片机来说，那是提升了太多太多，读者朋友们在单片机的入门阶段就能使用到 STC8 系列单片机实在是一种“幸福”。为啥我会有这样的感慨呢？小宇老师想起了宋丹丹老师在春晚小品中的一段台词，台词是这么说的：“我都畅想好了，我是生在旧社会，长在红旗下，走在春风里，准备跨世纪。想过去，看今朝，我此起彼伏。于是乎，我冒出了个想法。”从这段话里真的能看出新中国的巨大变化，这和单片机产品的进化与升级是一样的，其发展速度实在是太快了。碰巧的是，我和宋丹丹老师的想法是一致的，那就是要写本书，宋丹丹老师的书名叫《月子》，然而我写的书名不叫《伺候月子》，而叫《深入浅出 STC8 增强型 51 单片机进阶攻略》。我也畅想好了，为了让大家体会下单片机存储器资源的发展和变化，我也要带着大家回到 1974 年，回到那个 Intel 公司推出了基于 MCS-51 系列内核的 80C32 微控制器的时代。

在那个时代，80C32 微控制器已经算是“明星”产品了，80C32 内置了 8 位 CPU 核心单元、具备 256B 大小的内部数据存储器 RAM、具备 32 个准双向 I/O 口、拥有 3 个 16 位定时/计数器和 5 个中断源、片内具备一个全双工串行通信口。但 80C32 产品最为“奇葩”的一点是片内没有程序存储器 ROM，这是什么意思呢？也就是说，程序开发人员编写好的代码没有办法直接烧录到单片机中，必须要自己先把程序烧录到专门的 ROM 芯片，然后再把 ROM 芯片搭建到单片机的最小系统中去，要是程序的执行需要占用较大的 RAM 空间，那还得再扩展专门的 RAM 芯片，搭载了相关的内存芯片后，再利用地址总线和数据总线进行数据的交换和读写。这就太麻烦了！真是同情当年的工程师们。接下来，让我们看看那个年代的单片机“最小”系统，其实物样式如图 9.2(a)所示。



(a) (b)

图 9.2 基于 80C32 微控制器的最小系统正面样式

放眼望去，小宇老师怎么都不相信这个板子仅仅是个 80C32 的最小系统，那么多的芯片在一个洞洞板上，感觉应该是要实现很“厉害”的功能，然而它真的只是个最小系统而已。而且还是个主频低、功能少、内存小、性价比低的最小系统。

板子实物上一共有 7 个功能芯片，有半数以上都是为了扩展单片机内存的。芯片 1 是 74HCT573，这是一个拥有 8 路输出的透明锁存器，用于锁存和分离地址数据和一般数据。芯片 2 是 HM62256，这是一个具备并行接口的 32KB 大小 SRAM 存储器，有了它就能让 80C32 控制器的 RAM 资源得到扩充。芯片 3 是 27C64，这是一个 8KB 容量的可接受 UV 紫外线擦除的 EPROM 芯片，这种只读存储器使用起来非常麻烦，细心的读者朋友肯定发现了，板子上的 7 个芯片中就只有这个芯片表面贴了一层不透明的胶布，为啥要这样做呢？是因为这种芯片表面的玻璃窗口如图 9.2(b)所示，当紫外线(哪怕是日常的太阳光线中也含有紫外线)照射到芯片内部晶圆后，数据或者程序信息就会被缓慢擦除。所以该芯片内部一旦存在编程数据，就必须要用胶布(最好是黑色不透明的胶布)贴住玻璃窗口，使用上确实不便。芯片 4 就是整块板的核心，即 80C32 微控制器芯片。芯片 5 是 MAX232，这是一种电平转换芯片，可以实现 RS-232 电平转换为 TTL 电平，通过该芯片就能实现 PC 的 DB-9 端口与单片机 UART 引脚间的通信。芯片 6 是 74HCT138，这是一个 3 线 8 态译码器电路，用于实现各内存芯片的地址译码。芯片 7 是 74HCT00，这是一个 2 输入与非门电路，该芯片用于控制各内存芯片的使能和读写操作。

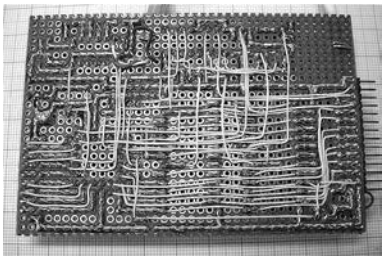


图 9.3 基于 80C32 微控制器的最小系统背面布线

把板子上的芯片这么一讲解，我们心里就清楚多了，这么大块板子其实也“没什么”。这就是 8032 那个时代的单片机产品，片内居然连 ROM 都不具备，片内资源也是少得可怜，内存容量也是小得不行。添加额外的芯片进行内存扩展都还不是最麻烦的，最麻烦的是芯片变多后布线难度就非常大，如果将如图 9.2(a)所示的最小系统翻一个面就会看到如图 9.3 所示的背面飞线，这些线接错一根就会导致访问错误，若需要在这个小系统上扩展其他外围芯片又得经过复杂的考虑，以免造成访问的冲突和引脚的占用。

怎么样？朋友们经过对比，是不是觉得现在的 STC 系列单片机非常好用？完全不用自己外扩 RAM 和 ROM 单元就可以轻松地编写和烧录程序了。现代单片机的封装形式非常多样，片上资源也较为丰富，STC 公司的很多 51 单片机产品甚至把时钟电路单元和上电复位电路做到了芯片内部，单片机本身仅需要 VCC 和 GND 两根电源线就可以工作了，剩下的 I/O 引脚都可以让开发人员随意使用。单片机产品经过不断的发展，其形态和性能相比以往已经有了很大的变化，我们可以基于单片机芯片做出更多有意义的电子模块和电子产品，所以小宇老师感慨：“能生在这样一个时代，是一件幸福的事情。”

9.3 你若是校长，教学楼和宿舍楼怎么修

接下来，咱们就来看看 51 单片机的存储结构，说白了，就是要知道 RAM 和 ROM 的区域划分问题，要说明白这个问题就要搞清楚普林斯顿结构(也可以叫作冯·诺依曼结构，因为该结构是由普林斯顿大学开发的)和哈佛结构的特点(哈佛结构是哈佛大学的研究成果)。但是直接讲述结构内容确实太枯燥，按照小宇老师的风格，我们先从“建楼”说起吧！假设读者朋友们新办了一所学校，你就是校长，在资金问题不用愁的情况下如何建立教学楼和宿舍楼这两个主要大楼呢？有很多朋友为你出谋划策，提出了如图 9.4 所示的两种方案，其实就是围绕两种楼是要建在一起还是分开的问题。

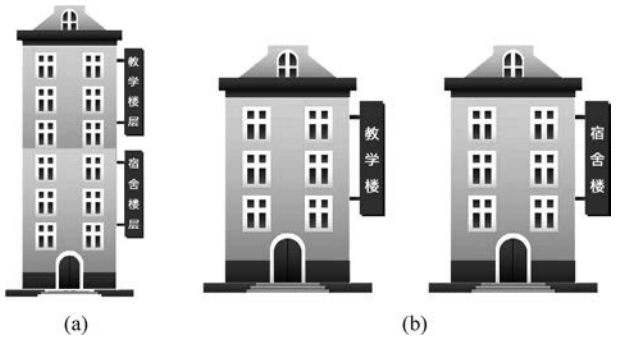


图 9.4 两种建楼方案示意图

图 9.4(a)主张建立“摩天大楼”(RAM 和 ROM 统一编址)，按照楼层号划分教学区域和宿舍区域(按照物理地址分界)，每层楼的房间数是确定的，从上到下都是严谨对齐的(指令数据位宽与数据位宽需一致)，建好之后一定很“气派”！图 9.4(a)方案一经提出后也有一些反面的“声音”，有不少朋友觉得图 9.4(b)方案可能更为“合理”。常理上讲，教学楼和宿舍楼从性质上讲还是不相同的(RAM 和 ROM 功能不同)，应该把两个楼单独建立(RAM 和 ROM 分开编址)，要是贸然把两种区域合并在一起可能产生一些麻烦。有人说：一整栋楼的电梯怎么装呢？有人要上楼有人要下楼，会不会冲突(访问瓶颈问题，取指令和取数据可能冲突)。又有人说：学生和老师那么多人，宿舍和教室又都叠加在一起了，那这个“摩天大楼”要高耸入云了吧？这会不会有什么安全隐患呢(最大寻址范围问题)？还有人说：学校以后扩招的话宿舍区需要容纳更多学生，宿舍房间应该多于教室房间吧(指令数据位宽可能和数据位宽不一致问题)！

两种建楼方案一经推出就引起了广泛讨论，其实两个方案各有优势。图 9.4(a)就是普林斯顿结构，其结构示意如图 9.5(a)所示，该结构下的 RAM 和 ROM 是统一编址的，受 51 单片机地址总线位宽限制(一般是 16 位地址总线)，其存储器能被寻址的最大空间是 64KB(即 2^{16}B=64KB)，该结构的组织形式非常简单，但是该结构一般要求指令数据位宽与数据位宽一致，数据指针和程序指针在访问相关数据内容时可能遇到效率低下或者冲突问题。

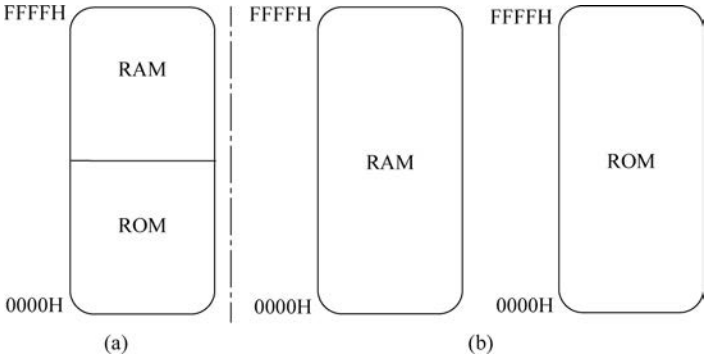


图 9.5 普林斯顿与哈佛存储结构示意图

图 9.4(b)就是哈佛结构，其结构示意如图 9.5(b)所示，该结构下的 RAM 和 ROM 是分开编址的，RAM 区域和 ROM 区域又有“片内”和“片外”的说法，片内就是芯片内部自带的，“片外”就是用户在芯片外部自己扩展的(比如添加专门的存储器芯片去获得更大的容量)。外部 RAM 存储器(也就是我们即将要学习的 xdata 区域)和 ROM 存储器(也就是我们即将要学习的 code 区域)的最大空间都可以支持到 64KB。从设计上看，哈佛结构要比普林斯顿结构复杂一些(从各类总线的连

接和分配上就能明显感觉到分开编址后提升了存储器设计的复杂度),该结构不强制要求指令数据位宽与数据位宽一致,在存储单元的组织上比较灵活,访问效率也比较高。因为哈佛结构的访问优势,现代微控制器芯片(包括 MCS-51 内核的单片机产品)的存储器结构一般都用哈佛结构。

需要说明的是,STC8 系列单片机中的不少型号都具备充足的 ROM 区域,STC8H8K64U 型号的单片机甚至把 ROM 区域做到了“打顶”的 64KB 大小,所以 STC8 系列单片机不再提供访问外部扩展 ROM 的总线,也就是说,该系列单片机仅支持片内 ROM 单元。但是 RAM 区域就要另说了,由于 STC8 系列单片机的片内扩展 RAM 区域还没有做到 64KB 这么大,所以针对 40 个引脚数量及以上的单片机型号,还可以根据实际需求扩展外部 RAM 单元,进一步增加 RAM 容量。

虽说 ROM 和 RAM 区域里面都是存放的“0”和“1”,但是这两种区域中的数据含义和功能特性还是有较大区别的,ROM 中存放的数据大都是些程序代码、固有数据和常数,RAM 中的数据多是一些临时的变量数据。RAM 和 ROM 的区别还不只是数据层面,随着区域的划分,各内存区域支持的寻址方式和操作方法也有不同。以汇编语言为例,针对不同内存区域使用数据传送类指令时“指令头”就要发生变化,访问片内 RAM 区域的时候可以用“MOV”,访问片外扩展的 RAM 区域时就要用“MOVX”,访问整个 ROM 区域时就要用“MOVC”才行。其实,这和我们的交通出行是一样的!如果我家住重庆,我需要到对街买个零食,那“骑车”去就可以了(即 MOV 指令头),要是我需要到重庆的其他市区,坐个“轻轨”也很合适(即 MOVX 指令头),要是我需要去北京出差,那估计只能是飞机或者火车了(即 MOVC 指令头)。

接着“指令头”的话题稍加扩展,指令的具体形式其实还与寻址方式有关,所谓“寻址方式”就是以什么样的方式去“找到”数据,找寻数据的方法越多,那数据访问能力就越强,一定程度上也能反映出单片机的性能。MCS-51 内核的单片机就支持直接寻址、寄存器寻址、寄存器间接寻址、立即寻址、变址寻址、位寻址和相对寻址等 7 种寻址方式。所以说,打铁还需自身硬,MCS-51 内核能够经久流传且被称作“经典”还是有自身原因的。

以“变址寻址”为例,这种寻址方式是用“基址+变址”的组合形式去表达新的地址,好比有朋友初来重庆,人生地不熟地想来找我,我就给他说“××小区喷泉左边的第一栋楼就是我家”,这句话里的“××小区喷泉”是个基础地址,“左边第一栋”就是个“变动地址”,使用变址寻址后的语句类似于“MOVC A,@A+DPTR;”,若 DPTR 中存放了 0120H,A 中存放了 05H,那“@A+DPTR”将会把 ROM 区域的 0125H 地址中的数据传送给 A。综上所述,朋友们也应该对汇编指令头和寻址方式有个大概认识了,这些内容就足够解决 9.1 节用 A51 编程的第 1 个小疑问了。虽说本书不讲汇编,但是小宇老师还是推荐朋友们在闲暇时间里看看汇编语言程序设计,说不定会反向加深我们对 C 语言的相关理解。

9.4 “宿舍区”就类似于程序存储器 ROM

建完了单独的教学楼和宿舍楼之后,我们就来看看两者的内容和特点。一个学校的主体一定是学生,没有学生就谈不上育人了。与学生关系最密切的肯定是“宿舍区”,这相当于孩子们求学阶段的“家”了。在宿舍区里,每位同学都有一个固定的房间或者床位,里面的物品是同学们自己规划和存放的,这个“宿舍区”就相当于单片机的程序存储器 ROM,在这些存储单元中存放着掉电非易失的数据(也就是说,单片机断电后,ROM 中的数据不会丢失),就好比学生在寝室居住的时间里,宿舍里的东西是不会凭空消失的。

ROM 存储器的类型有很多,常见的有掩膜型 ROM、EPROM、EEPROM 和 Flash 类型,在 STC 系列单片机中就是用的 Flash 这种类型的 ROM(有个别朋友一直搞不清楚 Flash 和 ROM 的关系,其实 ROM 是个大类别,Flash 是 ROM 中的一种)。有的朋友肯定好奇,在单片机 ROM 中究竟存

放着哪些东西呢？ROM 中一般存放了程序内容、固有数据（比如 C51 语言中定义的数据数组或者是汇编语言中的 DB“表格”数据等）和一些常量数据（例如 π 的取值），这些数据一旦“烧录”到 ROM 之后一般不会发生变化，就好比我们的寝室，新生入住之后寝室就是归自己所有了，别人也不会去动你的物品。

单片机的 ROM 区域一般要比 RAM 区域大很多，但是结构上却比 RAM 简单很多，所以我们先来学习 ROM 区域的结构。为了便于大家理解，先看如图 9.6(a)所示的“宿舍区”结构。每个学校的宿舍区都有个大门，我们将其称为“区域入口”，不管是上/下课还是放假离校、开学返校，我们总会由宿舍区大门进入宿舍，这个入口会天天和我们打交道。进入宿舍区后一般都会有一些生活“业务区”，比如饭卡充值的地方、小超市、洗衣服的洗衣房、宿管阿姨的办公室、缴纳电费的办公室、ATM 取款机等。这些地方不是经常去，只有等到特定事件发生的时候才会去（也就是单片机的中断“事件”产生时才会去对应入口），比如寝室电费用完了，造成了寝室断电，这时候才会去“业务区”缴纳电费。要说占用“宿舍区”最多的地方还是宿舍了，这才是主体部分。

有了“宿舍区”的相关概念做铺垫，再来看看如图 9.6(b)所示的 STC8 系列单片机 ROM 结构，其结构也可以大致分为 3 部分，原来的宿舍“区域入口”就是“起始地址”，原来的生活“业务区”就是各种“中断向量”的入口，这些中断源大多都是 STC8 系列单片机的片上资源。剩下的“宿舍”就相当于 ROM 存储单元了。

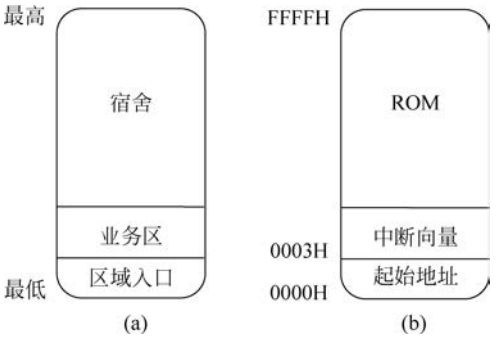


图 9.6 宿舍区结构与单片机 ROM 结构类比图

光是进行类比肯定是不够的，我们需要深入理解单片机程序的执行过程及 ROM 区域作用，在讲解这些内容之前，先来补充点单片机 CPU 有关的基础知识。从大的结构上讲，STC8 系列单片机 CPU 内包含控制器和运算器这两部分，运算器就负责数据运算，主要是由算术逻辑部件 ALU、累加器 ACC（数据运算的主要场合）、通用寄存器 B、程序状态字寄存器 PSW（数据运算的状态反映）和一些辅助电路共同构成。

控制器实现取指令、译码和逻辑控制的作用，主要是由程序计数器 PC（也称为 PC 指针，专门指向 ROM 区域的程序代码）、指令寄存器 IR、指令译码器 ID、数据指针 DPTR（专门指向 ROM 区域或 RAM 区域内的数据内容）和一些逻辑电路共同组成。CPU 的知识这里不过多展开，只需要关注下 PC 指针和 DPTR 指针就行。在单片机中，这两个指针都很“忙”，PC 一天到晚都在指向下一条要执行的指令地址，而 DPTR 也在 ROM 和 RAM 间来回跳转，指向相关的数据内容。当单片机复位时这两个指针都会被自动赋值为“0000H”，这个数值是不是在哪里见到过呢？没错！它就是 ROM 区域的起始地址。

这个起始地址就相当于宿舍区大门，单片机的 PC 指针在上电或者整体复位的时候就会强制性地回到 0000H 这个起始地址，从这个入口开始执行程序代码。所以 9.1 节用 A51 编程的第 2 个小疑问就可以得到解答了，“ORG 0000H;”语句的作用就是让程序从 0000H 开始执行。

宿舍区中的“生活业务区”其实就是中断向量入口，什么叫“中断向量”呢？其实就是一些处理“突发事件”的入口地址，这些事件大多来自于芯片上的相关资源，中断源越多，一定程度上就反映出了单片机性能的“强大”。比如外部中断引脚上出现了一个下降沿，在使能外部中断资源的情况下就可以产生一次外部中断请求。话又说回来，正常的生活中肯定不会有人天天往缴纳电费的办公室跑，肯定是等到寝室没电了才会意识到要充电费，所以，这些特定入口平日里都不会随意进入，除非是产生了相关的“中断”请求且 CPU 响应了请求，然后把 PC 指针强制性“拽”到对应入口

去执行相关内容。

以 STC8H 系列单片机的最高款 STC8H8K64U 单片机为例，其中断源及向量入口信息如表 9.1 所示。需要说明的是，中断源的数量会受单片机具体型号的影响产生差异，STC8 系列单片机中断源数量根据不同型号，会在 16～22 范围内波动，朋友们在确定好型号之后一定要去看手册，以免弄错。现在请朋友们仔细观察表 9.1，从初学的角度看，我们能发现哪些信息呢？

表 9.1 STC8H8K64U 单片机中断源及向量入口列表

向 量 号	中断向量	中 断 源	向 量 号	中断向量	中 断 源
0	$(0003)_H$	INT0	20	$(00A3)_H$	Timer4
1	$(000B)_H$	Timer0	21	$(00AB)_H$	CMP
2	$(0013)_H$	INT1	24	$(00C3)_H$	I^2C
3	$(001B)_H$	Timer1	25	$(00CB)_H$	USB
4	$(0023)_H$	UART1	26	$(00D3)_H$	PWMA
5	$(002B)_H$	ADC	27	$(00DD)_H$	PWMB
6	$(0033)_H$	LVD	35	$(011B)_H$	TKSU
7	$(003B)_H$	PCA	36	$(0123)_H$	RTC
8	$(0043)_H$	UART2	37	$(012B)_H$	P0
9	$(004B)_H$	SPI	38	$(0133)_H$	P1
10	$(0053)_H$	INT2	39	$(013B)_H$	P2
11	$(005B)_H$	INT3	40	$(0143)_H$	P3
12	$(0063)_H$	Timer2	41	$(014B)_H$	P4
16	$(0083)_H$	INT4	42	$(0153)_H$	P5
17	$(008B)_H$	UART3	43	$(015B)_H$	P6
18	$(0093)_H$	UART4	44	$(0163)_H$	P7
19	$(009B)_H$	Timer3			

第一个发现就是：STC8H8K64U 单片机的中断源有 33 个之多，可以从侧面反映该型号单片机的强大。有的朋友可能早期学过 STC89C52 型号的单片机产品，该款芯片仅有 5 个中断源，片上资源非常基础，功能上也比较单一。

第二个发现就是：STC8H8K64U 单片机的中断向量号貌似“不连续”，从 0 到 44 缺少了 13、14、15、22、23、28、29、30、31、32、33 和 34(实际上只有 33 个中断向量号可用)，这是为什么呢？缺失的向量号其实并不影响开发者使用，这几个缺失的向量号早期是为了预留给其他片上资源设计的，编程时忽略这些向量号即可。

第三个发现就是：中断向量从$(0003)_H$这个地址开始，按照地址递增的方式安排其他的中断向量，每两个相邻的中断向量之间有 8B 的空间大小(不考虑缺失的中断向量)。这 8B 是用来做什么的呢？有的朋友肯定要说：当然是为了存放用户编写的中断代码啊！话是不错，但是代码要是很大，8B 都不够“装”怎么办呢？这确实是个问题，所以这 8B 中并不是用于存放真正的中断代码，而是存放了一条能“找到”中断代码的无条件“跳转”指令，这个指令会带着 PC 指针去往真正存放中断代码的存储区块，然后再去执行中断代码程序(即中断服务函数中的代码内容)。

9.5 “教学区”就类似于数据存储器 RAM

说完了“宿舍区”，再来看看“教学区”。一般的学校里，教学区的复杂度都要比“宿舍区”大一些，一个大学肯定有多功能学术报告厅、授课教室、二级学院的办公室、资料档案室、专业实验室、行

政管理办公室等。这些区域功能各异，管理和服务于日常教学、学生实践和学院发展，要是把这些内容都建立到一栋“大楼”中去，那这栋大楼一定非常高，当然，也会很气派！小宇老师也当一次大楼的“设计师”，经过精心安排，我设计出的“双峰”综合教学楼如图 9.7 所示。

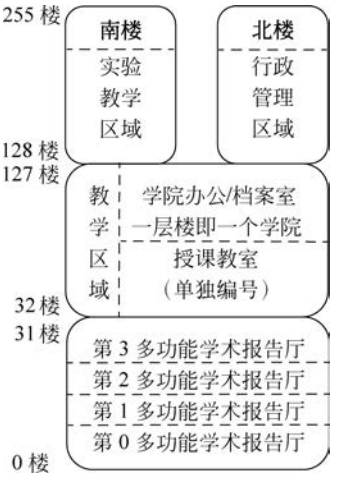


图 9.7 小宇老师的“双峰”综合教学楼结构图

我们先来说一说“双峰”综合教学楼的构成及区域作用。这栋楼有 256 层高，有的朋友可能会担心：这么高的楼肯定不安全，倒了咋办？这……我确实没想过，我们就假设它很“牢靠”吧！这个楼的设计仅仅是为了引出我们即将要学习的“知识”，大家不必太较真。我们将这栋综合教学楼从 0 开始编号，可以得到第 0～255 楼。

从 0 楼往上到 31 楼，我设计了 4 个“多功能学术报告厅”，每个报告厅由 8 层楼大小的区域组成，总共就是 32 层楼这么大。多功能学术报告厅是用来干吗的呢？在大学里，经常会有校内外专家学者前来授课和分享学术知识，在这种情况下，总得找个地方让专家讲课吧？所以这种大厅很有用处。除了讲座之外，做个新生入学教育或者毕业生的毕业典礼也是不错的。那为什么小宇老师要设计 4 个报告厅呢？因为很多时候单独的一个报告厅不足以满足使用需求，有一些时间段可能会出现好几个二级学院都想占用报告厅的情况，这时候多几个备用报告厅也是有必要的。

32～127 楼就是通用教学区了，该区域又细分为学院办公/档案室和授课教室这两部分。授课教室占用了 16 个楼层，每层楼有 8 个教室，总计 128 间教室，这些教室都是单独编号的，所以授课教室区域是以“房间号”去区分，同学们可以按照编号去找到任何一间教室。学院办公/档案室占用了 80 个楼层，主要是分配给学校的二级单位，为了方便管理，一个单位就占用一层楼，所以学院办公/档案室是以“楼层号”去区分。

再往上就是 128～255 楼了，为了设计出综合教学楼的特色，小宇老师构造了“双峰”形式，在 127 楼的平台上为大楼修建了“两个耳朵”。南楼是 128～255 楼，主要是实验教学区域，也是按照一个专业一个楼层的分配方法布置的。需要说明的是，学生不能随意进入南楼实验区，只能是获得实验审批后才能进入南楼开展实验。北楼也是 128～255 楼，也是按照一个机构一个楼层的分配方法布置的。需要说明的是，北楼的进入并不需要授权，但凡是有教学需求都可以直接找到相关办公室。“两个耳朵”的功能是不同的，但是楼层号是一致的，所以在平时交流时，一定要说清楚是哪一个子楼，可以用“南楼 128 层”和“北楼 128 层”的说法加以区分。

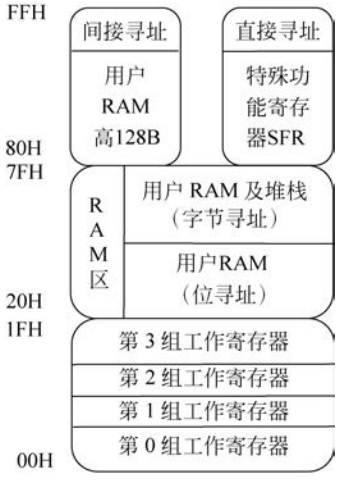


图 9.8 STC8 系列单片机 RAM 结构图

了解这栋楼的设计理念和区域功能后，朋友们就能轻松地理解 STC8 系列单片机 RAM 区域的构成和含义了，要是把 STC8 系列单片机的 RAM 区域拿出来和“双峰”综合教学楼进行区块类比，就能得到如图 9.8 所示的样子。

整个楼层还是 0～255 楼，只是表达方式上写成了 00H～FFH 的十六进制地址形式。从 00H 往上到 1FH 是 4 个工作寄存器组，这个区域对应了之前的“多功能学术报告厅”。每个工作寄存器组又是由 8 个寄存器组成，也就是 R0～R7 寄存器，全部加起来就有 32 个寄存器单元。有的朋友可能会疑惑了，这 4 个

工作寄存器组都有R0～R7寄存器，那岂不是有4个R0寄存器？都叫这个名字不会冲突吗？朋友们确实心细，在该区域中确实存在4个R0寄存器，虽说名字相同，但是物理空间是有差异的。与生活中的报告厅不一样的是：任一时刻下，CPU只能从这4个工作寄存器组中选定1个作为当前工作寄存器组，所以不必担心冲突问题。工作寄存器组可以用于暂存运算的临时状态或者临时数据(第11章的中断资源就会用到，届时将展开细致的讲解)，采用寄存器名称直接编程和暂存参数，使用上十分灵活且有助于提高运算速度，提升代码的执行效率。MCS-51内核中提供了4个工作寄存器组就是为了避免1个工作寄存器组不够用的情况。所以9.1节用C51编程的第3个小疑问就可以得到解答了，“void Int0() interrupt 0 using 1”语句中的“using 1”就是选择了第2个工作寄存器组存放相关临时数据，“using”后面的数值支持0～3，正好可以对应这4个工作寄存器组。抛开C51的上层代码，从单片机的底层上看，工作寄存器组的具体选择是由单片机内部程序状态寄存器PSW中的“RS1”和“RS0”位决定的。那PSW这个寄存器又有什么用呢？简单来说，它就是一个反映CPU数据运算状态的寄存器，这个寄存器中的大部分功能位都是一些标志位，在实际编程中一般不会用到，这里只看“RS1”和“RS0”位即可，程序状态寄存器的相关位定义及功能说明如表9.2所示。

表9.2 STC8单片机程序状态寄存器

程序状态寄存器(PSW)								**地址值：$(0xD0)_H$**
位 数	位7	位6	位5	位4	位3	位2	位1	位0
位名称	**CY**	**AC**	**F0**	**RS1**	**RS0**	**OV**	**F1**	**P**
复位值	0	0	0	0	0	0	0	0
位 名	位含义及参数说明							
CY 位7	借位/进位标志							
	0	最高位无借位或者进位						
	1	最高位借位或者进位						
AC 位6	辅助借位/进位标志							
	0	低4位没有向高4位发生借位或者进位						
	1	低4位向高4位发生借位或者进位						
F0 位5	用户标志位0							
	0	清零用户标志位0						
	1	置位用户标志位0						
RS1-RS0 位4:3	工作寄存器组选择位							
	00	第0组工作寄存器			01	第1组工作寄存器		
	10	第2组工作寄存器			11	第3组工作寄存器		
OV 位2	溢出标志位							
	0	累加器ACC中的运算结果没有超出8位二进制数据范围						
	1	累加器ACC中的运算结果超出8位二进制数据范围						
F1 位1	用户标志位1							
	0	清零用户标志位1						
	1	置位用户标志位1						
P 位0	奇偶标志位							
	0	累加器ACC中“1”的个数为偶数个						
	1	累加器ACC中“1”的个数为奇数个						

也可以将这4个工作寄存器组的配置信息和地址信息单独拿出来，做成如表9.3所示内容。

表 9.3 工作寄存器组选择及地址分配

配置位		选定组号	内部寄存器组成							
RS1	RS0		R7	R6	R5	R4	R3	R2	R1	R0
0	0	0	07H	06H	05H	04H	03H	02H	01H	00H
0	1	1	0FH	0EH	0DH	0CH	0BH	0AH	09H	08H
1	0	2	17H	16H	15H	14H	13H	12H	11H	10H
1	1	3	1FH	1EH	1DH	1CH	1BH	1AH	19H	18H

在“工作寄存器组”相关知识点的末尾一定要讲解下“SP”这个 8 位宽度的堆栈指针，这个指针固定指向 RAM 中的堆栈区。堆栈是什么呢？简单来说就是一种用于组织数据在内存中存放的结构。“栈”可以用来暂存数据，特别是用于保存中断发生时的“现场数据”，该结构有个特点就是先进后出(First In Last Out，FILO)，这个特点正好保证了中断返回时的断点数据恢复流程，在本章不做深入展开。我们把堆栈想象成一个“羽毛球筒”就可以了，生活中见到的羽毛球筒都是底面封死的，只能从一个出口取出羽毛球，若用户想要最下面的羽毛球，就必须把全部的羽毛球都拿出来才行，这就是堆栈的特点。那堆栈和工作寄存器组之间有什么联系呢？干吗要在中途插入 SP 的知识点呢？

这是因为 SP 这个堆栈指针在单片机上电后有个默认值，它会指向单片机内部 RAM 的 07H 这个地址，朋友们是不是有点儿熟悉这个地址？这不就是第 0 组工作寄存器的末尾吗？那指向这个地址有什么不好吗？我们来假设一下，假设单片机的程序启用了这 4 个工作寄存器组，同时又具备相关中断服务函数(也就是说需要用到堆栈区)，这时候一旦发生中断，则相关数据就会从 07H 地址往后开始写入“现场数据”，这些数据会怎么写入呢？当然是“无情”地覆盖第 1 组工作寄存器中的相关内容！这就麻烦了！若 SP 指针指向的地址不恰当，就会造成一些重要数据的“覆盖”，所以编程者在程序代码执行之前都会将 SP 指针重新指向到 80H 单元以后，以免产生数据覆盖。所以 9.1 节用 A51 编程的第 3 个小疑问就可以得到解答了，迁移 SP 指针就是为了避免重要数据被误覆盖。

20H～7FH 就是通用 RAM 区了，这个区域对应了之前的“通用教学区”，该区域又细分为两部分，第一部分是仅支持字节寻址的用户 RAM 及堆栈区(即学院办公/档案室)，第二部分是支持位寻址的用户 RAM 区(即授课教室)。

“授课教室”占用了 16 个楼层(即 16 字节地址)，每层楼有 8 个教室(即 8 个位地址)，总计 128 间教室(即 128 个位地址)，这些位地址可以被单独地“找到”，它的用处很多，在编程中可以把某个位地址进行“变量”化处理，将其当成一个标志位，用于指示一些事件或者内部动作的产生，这个“变量”的“1”或者“0”就表示“有事件”或者“无事件”，使用起来非常方便，20H～2FH 区域不仅支持位寻址，也支持常规的字节寻址。

“学院办公/档案室”占用了 80 个楼层(即 80 字节地址)，但这个区域不能用“教室号”去找，必须是按照“楼层号”去区分，所以 30H～7FH 这个区域仅支持字节寻址。简单地说，位地址指向的就是一个二进制位，而字节地址指向的是 1 字节，即 8 个二进制位。小宇老师这样说可能并不直观，所以列了一个如表 9.4 所示的分配情况。

表 9.4 位寻址区的字节地址和位地址分配

字节地址	位地址							
	位 7	位 6	位 5	位 4	位 3	位 2	位 1	位 0
20H	07H	06H	05H	04H	03H	02H	01H	00H
21H	0FH	0EH	0DH	0CH	0BH	0AH	09H	08H
22H	17H	16H	15H	14H	13H	12H	11H	10H

续表

字节地址	位 地 址							
	位 7	位 6	位 5	位 4	位 3	位 2	位 1	位 0
23H	1FH	1EH	1DH	1CH	1BH	1AH	19H	18H
24H	27H	26H	25H	24H	23H	22H	21H	20H
25H	2FH	2EH	2DH	2CH	2BH	2AH	29H	28H
26H	37H	36H	35H	34H	33H	32H	31H	30H
27H	3FH	3EH	3DH	3CH	3BH	3AH	39H	38H
28H	47H	46H	45H	44H	43H	42H	41H	40H
29H	4FH	4EH	4DH	4CH	4BH	4AH	49H	48H
2AH	57H	56H	55H	54H	53H	52H	51H	50H
2BH	5FH	5EH	5DH	5CH	5BH	5AH	59H	58H
2CH	67H	66H	65H	64H	63H	62H	61H	60H
2DH	6FH	6EH	6DH	6CH	6BH	6AH	69H	68H
2EH	77H	76H	75H	74H	73H	72H	71H	70H
2FH	7FH	7EH	7DH	7CH	7BH	7AH	79H	78H

有的朋友看完表格后又有疑问了！为啥字节地址和位地址还有一样名称的啊？大家看看字节地址的“24H”这一行，为啥还有个位地址也是“24H”？其实这很简单，在生活中也有“4楼4号房”的说法，这个楼层的“4”和房间号的“4”虽然数值一样，但是含义不同，朋友们无须纠结。

再往上就是80H～FFH了，这个区域就是综合教学楼的“双峰”部分。南楼就是“实验教学区域”，是用户RAM区域的高128字节部分，该区域仅支持间接寻址方式（需要得到授权才能进入）。北楼就是“行政管理区域”，该区域仅支持直接寻址方式，这里面装载的不是一般的内容，分布其中的都是一些特殊功能寄存器（也称为SFR，在第4章学习的I/O部分的PxM0、PxM1、Px、PxPU、PxNCS、PxSR、PxDR和PxIE都是特殊功能寄存器，它们决定了I/O引脚的相关功能，在后续的章节学习中还会遇到更多，需要说明的是STC8系列单片机内还存在一些“扩展SFR”单元，这些单元位于片内扩展RAM区，其访问和操作方法不同于常规SFR，我们学过的Px类寄存器就属于“常规SFR”，我们学的PxPU这种寄存器就属于“扩展SFR”，此处做简要说明）。这些寄存器关系到单片机的内部电路、资源选择、参数配置和整体性能，这些作用其实和“行政管理”是一样的，一个学校缺少了行政管理也将会是一盘散沙。

说到这里，小宇老师长舒了一口气，STC8系列单片机的片内256B的RAM算是讲解完毕了，但是256B的RAM是否够用呢？如果用这个容量大小的RAM去“跑”一些简单的程序，那肯定是用不完的，但是如果程序中构建了一些算法，或者加入了一些协议栈、小型调度器或者实时操作系统，256B RAM就不够用了。

那怎么扩展RAM空间呢？是不是要像9.2节讲解的那样，在一块板子上用乱七八糟的飞线去连接一片专用RAM芯片呢？这倒不用，STC8系列单片机自带了内部扩展RAM单元，其容量支持2～8KB，以STC8H8K64U型号单片机为例，这款单片机就自带了8KB大小的片内扩展RAM。这还不算厉害的，最厉害的是内部扩展RAM居然不占用芯片内部的相关总线（例如P0和P2），也不需要额外的控制线路（例如RD、WR和ALE线路），大家操作该区域的方法就按照传统51扩展RAM的形式即可。

STC8系列单片机片内扩展RAM区域在上电后是默认启用的，当然，朋友们也可以按照需求禁用该区域，只需配置辅助寄存器AUXR中的“EXTRAM”功能位即可，为了方便讲解，小宇老师将辅助寄存器AUXR中的其他功能位做了“屏蔽”，若只看“EXTRAM”功能位，其含义说明如表9.5所示。

表 9.5 STC8 单片机辅助寄存器

辅助寄存器(AUXR)								地址值：$(0x8E)_H$
位 数	位 7	位 6	位 5	位 4	位 3	位 2	位 1	位 0
位名称	**T0x12**	**T1x12**	**UART_M0x6**	**T2R**	**T2_C/T**	**T2x12**	**EXTRAM**	**S1ST2**
复位值	0	0	0	0	0	0	0	1
位 名	位含义及参数说明							
EXTRAM 位 1	内部扩展 RAM 访问控制位							
	0	访问内部扩展 RAM，当访问地址超过了内部 256B 的 RAM 区域时，系统会自动切换到内部扩展 RAM 区域来						
	1	禁用内部扩展 RAM 区域						

如果从 RAM 区域的划分上严格地去界定，STC8 系列单片机的片内扩展 RAM 区域其实属于“外部扩展 RAM”的一部分，外部扩展 RAM 的最大寻址范围是 64KB，若是启用了片内扩展 RAM 之后，外部还能扩展的 RAM 容量将会变小。以 STC8H8K64U 型号单片机为例，若启用该单片机的 8KB 片内扩展 RAM 后，外部还能扩展的 RAM 大小将会是 64KB 减去 8KB，即 56KB，也就是如图 9.9(a)所示的构成。如果编程者不启用片内扩展 RAM 区域，则外部可扩展的 RAM 容量如图 9.9(b)所示。

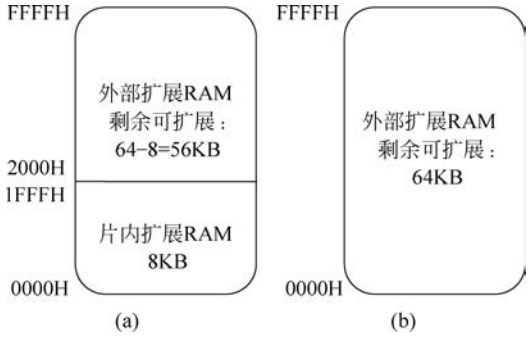


图 9.9 EXTRAM 位配置与扩展 RAM 区域构成图示

需要注意的是，在 STC8 系列单片机中只有那些引脚数量在 40 个或以上的型号才具备外部扩展 RAM 的能力，若需要启用外部扩展 RAM 功能就必须有所“牺牲”了，毕竟是外挂了专用芯片，必须要提供数据总线，还要提供一些引脚去做控制(例如 WR、RD 和 ALE 信号引脚)。值得一提的是，STC8 系列单片机还专门为外部扩展 RAM 功能做了一个总线速度控制寄存器 BUS_SPEED，该寄存器可以对 RD/WR 引脚功能做选择，也能对总线读写速度做配置，该寄存器的相关位定义及功能说明如表 9.6 所示。

表 9.6 STC8 单片机总线速度控制寄存器

总线速度控制寄存器(BUS_SPEED)								地址值：$(0xA1)_H$
位 数	位 7	位 6	位 5	位 4	位 3	位 2	位 1	位 0
位名称	**RW_S[1:0]**		—	—	—	**SPEED[2:0]**		
复位值	0	0	x	x	x	0	0	0
位 名	位含义及参数说明							
RW_S[1:0] 位 7:6	RD/WR 控制线选择位							
	00	P4.4 为 RD，P4.2 为 WR			01	保留		
	10	保留			11	保留		
SPEED[2:0] 位 2:0	总线读写速度控制 读写数据时控制信号和数据信号的准备时间和保持时间							

至此，我们就讲完了 STC8 系列的 RAM 区域和 ROM 区域组织形式和区域作用了，是不是觉得其实也不难？MCS-51 架构下的单片机距离现在已经好几十年了，所以从资源的复杂度上来说绝对不算复杂。为了梳理相关知识点，我们还是以 STC8H8K64U 单片机为例，默认启用该单片机 8KB 内部扩展 RAM 单元后的存储器资源如表 9.7 所示。

表 9.7　STC8H8K64U 单片机存储器资源信息

存储器区域		内部划分	再次划分	容量大小	起始地址	结束地址
RAM 区域	内部 RAM	低 128 字节区域	工作寄存器组	32B	00H	1FH
			位寻址区	16B	20H	2FH
			用户 RAM 及堆栈区	80B	30H	7FH
		高 128 字节区域	—	128B	80H	FFH
	外部扩展 RAM	内部扩展 RAM	—	8192B 即 8KB 占用	0000H	1FFFH
		外部扩展 RAM	—	57 344B 即 56KB 剩余	2000H	FFFFH
ROM 区域	—	—	程序入口	—	0000H	0000H
			中断向量	每个向量间隔 8B 区域	0003H	—
			普通 ROM	除去之间占用的剩余区域	—	FFFFH

需要说明的是，在表格末尾中断向量和普通 ROM 这两行的“起始地址”及“结束地址”中使用了“—”标记，这是提醒朋友们 STC8 系列单片机的中断向量空间根据型号的不同会有差异，所以在中断向量的结束地址和普通 ROM 的起始地址处没有写明具体的地址，要按照实际单片机的型号去查阅相关地址。

9.6　在 Keil C51 中看似“无用”的配置项

学习了单片机的存储资源之后，小宇老师就要开始连环“发问”了！RAM 区域中的数据是如何组织存放的呢？RAM 中有多个区域，我们要按照什么策略选择变量存放的位置呢？这些变量的地址在使用过程中是否会发生变动呢？ROM 中的程序代码又是从哪里开始存放的呢？程序中要是有中断函数的话，中断部分的代码又会存到哪里呢？要解释清楚这些问题就要看看 Keil C51 环境下的存储器资源是怎么配置的，程序代码是怎么写的，烧录到单片机中的“固件”文件是怎么得到的。

一般来说，在初学 STC 系列单片机时，只会触及 Keil C51 环境中的一些基本功能，比如通过 STC-ISP 软件添加 STC 系列单片机型号和头文件到 Keil 环境中，利用 Keil 新建项目工程，或者配置项目选项卡中的相关项，让工程代码编译后可以得到“.Hex”文件，最后再用 STC-ISP 软件下载 Hex 文件到单片机内部即可。在开发环境中，我们貌似没有接触到与“存储器资源”相关的任何地址或者区块配置，我们只管写代码，至于变量怎么放？区域怎么选？这些问题我们貌似都“不关心”。

再者，初学时我们编写的 C51 代码并不复杂，其风格与标准 C 差不多，写来写去貌似都是那几个常用数据类型和语句结构，特别是简单的实验，从代码上貌似也找不到一丁点儿与“存储器资

源”相关的关键字或者语句。

那就奇怪了！为什么初学阶段的朋友们在不了解51单片机内部存储器资源的情况下也能顺利开展相关实验呢？为什么Keil C51环境没有出现复杂的“存储器配置项”呢？这都是因为Keil C51环境的C51编译/链接器太“宠爱”编程者了！生怕编程者为底层操作而伤脑筋。就拿变量的分配来说，Keil C51的编译器内部支持3种编译模式(SMALL模式，COMPACT模式和LARGE模式)，常规的变量分配、存储器规划等工作在编译时就已经做好了，难怪我们感觉不到内部存储器资源的存在。要是把开发语言从C51语言换成A51语言，那C51编译器就帮不了我们了，一旦失去了“宠爱”我们的好帮手，我们就只能乖乖地从底层资源开始学起了。这就解释了一个问题，A51的编程者不懂存储器那就寸步难行，C51的编程者在初级阶段不懂存储器貌似也没什么太大影响，但是遇到实际问题时，还是需要回头补充相关知识的。

接下来就谈一谈Keil C51的3种编译模式，啥叫“编译模式”呢？简单来说，就是一组可以按照实际单片机内存情况去调配变量存储区域的配置项。例如，现有程序的变量数量非常少，不需要那么多的RAM空间，这时候就可以把变量区域放在单片机片内RAM的低128字节范围内，因为这个区域的变量访问速度是最快的。那我们就可以为工程文件选择Keil C51编译模式中的SMALL模式，该模式下的变量默认都分配到片内RAM的低128字节中去了。

有的朋友可能对存储区域的描述感到困惑，特别是初学者，一听到什么“片内RAM”“片外扩展RAM”“低128字节”“高128字节”就觉得头疼，Keil C51其实也发现了这个问题，所以Keil对MCS-51内核的单片机存储器区间进行了“关键字”形式的“二次重命名”。啥意思呢？就是把这些区域的叫法改变一下，变成一些C51中的扩展关键字(标准C中不存在这些关键字)，方便编程者去使用。例如“片内RAM的低128字节范围”，直接二次命名为“data”区域，“data”就是C51语言中的一个扩展关键字，也是一种“存储类型”，这样一来就很好理解了。类似的存储类型还有bdata、idata、xdata、pdata、code等。具体的存储空间与存储类型对照情况如表9.8所示。

表9.8 STC8系列单片机存储空间及存储类型对照

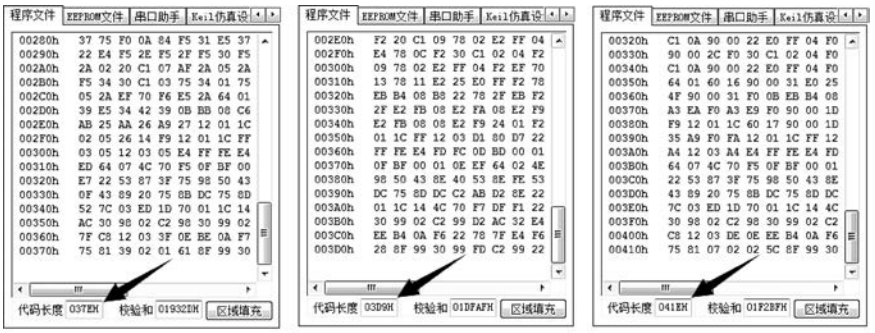
存储器区域		内部划分	再次划分	起始地址	结束地址
RAM区域	内部RAM(data)	低128B区域	工作寄存器组	00H	1FH
			位寻址区(bdata)	20H	2FH
			用户RAM及堆栈区	30H	7FH
		高128B区域(idata)	—	80H	FFH
	外部扩展RAM(xdata)	外部扩展RAM的低256B区域(pdata)	—	0000H	FFFFH
ROM区域	整个ROM区(code)	—	程序入口	0000H	0000H
			中断向量	0003H	—
			普通ROM	—	FFFFH

了解了存储类型与存储空间的对应关系之后再来看Keil C51的编译模式就很好理解了，3种编译模式的特点及变量存放区域如表9.9所示，SMALL模式也叫作“小编译模式”，适合变量不多的情况，也是Keil C51项目选项卡的默认配置。COMPACT模式可以叫作“紧凑编译模式”，当变量数量适中时可以选这个选项。LARGE模式就是“大编译模式”了，适合变量较多的情况。

表 9.9 Keil C51 环境的 3 种编译模式及特点

编译模式	变量存放区域	存储类型	模 式 说 明
SMALL	内部 RAM 的低 128B	data	该 RAM 区域访问数据的速度是最快的，但是空间大小有限，要是程序中的变量和临时空间需求较大，就不太适合
COMPACT	外部扩展 RAM 的低 256B	pdata	该 RAM 区域位于外部扩展区域，访问效率介于 data 和 xdata 之间
LARGE	整个外部扩展 RAM 区域(最大 64KB)	xdata	该 RAM 区域的数据访问效率就稍微低一些，但是容量很大，可以满足临时空间需求较大的情况

需要特别说明的是，编译模式的调整会影响代码内容、代码结构和代码大小，这样一来就可能导致程序产生运行差异，胡乱调配的话可能导致程序无现象，所以要合理选择。如果遇到某种模式下程序无现象，可以尝试调配到别的模式继续观察。为了方便理解，小宇老师对同一个程序工程进行了编译模式的调整，得到了如图 9.10 所示的代码内容，使用默认 SMALL 模式时的代码长度为如图 9.10(a)所示的$(0x037E)_H$，COMPACT 模式下的代码长度为如图 9.10(b)所示的$(0x03D9)_H$，LARGE 模式下的代码长度为如图 9.10(c)所示的$(0x041E)_H$。通常情况下，对于同一个程序代码，在不同的编译模式下，编译得到的代码量大小关系是：SMALL 模式 < COMPACT 模式 < LARGE 模式。



(a) (b) (c)

图 9.10 不同编译模式下的固件代码差异

通过学习，我们了解了 Keil C51 环境中的 3 种编译模式，也了解了 C51 语言还具备标准 C 语言所没有的 21 个扩展关键字，有了这些知识的铺垫，再来看看以下几条语句就很好理解了(假定这些语句都在同一个代码文件中)。

```
#pragma COMPACT
//写给编译器看的,调整本项目编译模式为 COMPACT 模式
unsigned char a;
//定义无符号字符型变量 a,受编译模式影响,将其分配到 pdata 区域
unsigned char xdata i;
//定义无符号字符型变量 i,编程人员将其指定分配到 xdata 区域
unsigned char code NUM[10];
//定义无符号字符型数组 NUM[10],编程人员将其指定分配到 code 区域
unsigned char data COMBUF[8] _at_ 0x20;
/*定义无符号字符型数组 COMBUF[8],编程人员将其指定分配到 data 区域并通过 C51 语言扩展关键字
"_at_"指定数组 COMBUF[8]从绝对空间地址 0x20 开始存放*/
void Int0() interrupt 0 using 1{【略】}/*外部中断 0 的中断服务函数："interrupt 0"表示中断向量为 0,即函数入口为 INT0 的入口地址 0003H(请参考表 9.1 内容),"using 1"表示中断函数占用第 1 组工作寄存器(即 9.5 节中的"多功能学术报告厅"区域)*/
```

说到这里，9.1 节用 C51 编程的疑问 1 和疑问 2 也得到解答了。接下来，再来看看第 4 个小疑问，也就是 Keil C51 环境 Target 选项卡的内容。这个选项卡的内容一定要了解，我们对 Keil C51 环境越是熟悉，编程起来就越是轻松。我们可以按照图 9.11 所示的样式将选项卡的相关配置进行区域划分，然后逐一“拿下”这 6 个区域的作用。

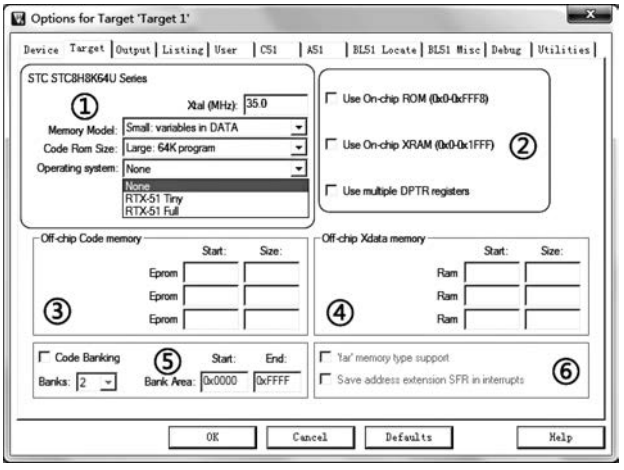


图 9.11 Keil C51 环境 Target 选项卡内容

第 1 区域：用于配置仿真频率、编译模式、代码空间及实时操作系统的支持。

该区域左上角显示出的“STC STC8H8K64U Series”是指当前工程所用单片机型号为 STC 公司生产的 STC8H8K64U，朋友们需要在 Keil C51 环境搭建完毕后，用 STC-ISP 工具添加 STC 相关信息到 Keil 安装目录中（具体方法在 3.2.2 节讲解过，此处不再赘述）。

Xtal(MHz)选项用于设定工程仿真调试时所用的单片机工作频率，配置频率的大小只会影响仿真调试模式下的程序执行速度，这个配置与最终产生的目标代码无关。我们可以把这个频率设定为单片机实际运行的工作频率，这样就可以在调试模式下看到程序的大致执行时间。当然，如果不用仿真功能，不设定这个值也可以，不影响实际的代码内容。

Memory Model 选项用于设定 Keil C51 环境下的 3 种编译模式，也就是之前学习的 SMALL 模式（变量优先存放在 data 区）、COMPACT 模式（变量优先存放在 pdata 区）和 LARGE 模式（变量优先存放在 xdata 区）。这个选项的默认配置是选用 SMALL 模式，也可以根据实际需求灵活调整。

Code Rom Size 选项用于选择代码空间的大小，也就是设置 ROM 空间的使用，该选项里面支持 3 种配置，第一个是 Small 配置，这个配置适合于那些 ROM 空间小于等于 2KB 的单片机型号（比如 Atmel 公司早期推出的 AT89C2051 单片机就只有 2KB 大小的片内 ROM），因为 ROM 区域很小，所以这种配置下就会影响 Keil C51 编译时候的一些策略，比如不要用长跳转指令，尽量选择短跳转指令，要是“跳猛了”到 2KB 外面去了就会发生程序错误。第二个就是 Compact 配置，这个配置下的工程代码大小可以支持到 64KB，但是单个函数的代码大小不能大于 2KB，也可以理解为全局的程序执行可以利用长跳转，但是在一些局部的子函数代码内就要用短跳转，这个配置最好不要乱选，除非非常确定单个函数的代码大小才能确保该配置下不会出现程序异常。第三个就是 Large 配置，这个配置下允许程序使用全部的 64KB 空间，而且不会产生单个函数代码大小的限制，通常情况下都选 Large 配置。

Operating system 选项用于选择实时操作系统的支持。MCS-51 内核的单片机还能装操作系统？是 Windows 7 还是 Windows 10？小宇老师要告诉朋友们，MCS-51 内核的单片机当然能“跑”操作系统，但是这里的操作系统并非 PC 上的大型操作系统，51 单片机上用的都是些轻量级的实时

操作系统，这些系统的代码量不大，能实现 CPU 时间片的合理分配、多任务调度和多任务处理，在单片机的提升学习中务必要掌握（在本章中暂时不用展开太多，这部分内容会在第 22 章进行展开讲解）。

Keil C51 环境为我们提供了两种轻量级的操作系统支持，分别是 RTX-51 Tiny 版本和 RTX-51 Full 版本。在一般的实验工程中都不需要选择操作系统的支持，所以该选项默认配置为 None，即不使用操作系统。

第 2 区域：选择片上 ROM 空间、XRAM 空间、双 DPTR 指针支持。

这个区域内有 3 个选项，默认情况下都不用勾选。

第一个选项是 Use On-chip ROM，这个选项决定是否使用单片机片内 ROM 资源，由于实际使用的是 STC8H8K64U 单片机，该系列单片机只能使用片内的 ROM 资源（不支持 ROM 外扩），所以这个选项是否勾选都不会影响到单片机的程序运行。

第二个选项是 Use On-chip XRAM，这个选项决定是否使用单片机外部扩展 RAM 空间，在 STC8H8K64U 型号的单片机中本身就具备 8KB 大小的外部扩展 RAM，而且在默认情况下，这个 8KB 空间都是被启用的，所以这个选项也可以不勾，这里说的 8KB 空间其实设计在 STC8H8K64U 单片机的内部，但是从严格意义上说，该空间属于“外部扩展 RAM”，也就是属于 xdata 区域。

第三个选项是 Use multiple DPTR registers，这个选项决定是否使用单片机内部的双数据指针功能，一般情况下，勾选这个选项之后会在最终代码中多出一些汇编语句来实现双数据指针的启用，建议合理选择。因为双数据指针如果没能正确使用，会造成程序功能混乱，一般都不用勾选该项。就以 STC8 系列单片机为例，其芯片内部确实集成了两组 16 位宽度的数据指针（DPTR0 和 DPTR1），通过内部相关寄存器的调配可以实现数据指针的一些基本功能，比如自增、自减或者切换等。两个数据指针好比一栋楼有两部“电梯”，使用上确实灵活，但是配合不好也会有问题。在 STC8 系列单片机中，与双数据指针有关的寄存器就有 6 个（第一组数据指针低 8 位寄存器 DPL、第一组数据指针高 8 位寄存器 DPH、第二组数据指针低 8 位寄存器 DPL1、第二组数据指针高 8 位寄存器 DPH1、指针选择寄存器 DPS 和时序控制寄存器 TA 等）。这些寄存器的内容，我们做个了解即可，如果是用 A51 语言开发程序的话就需要继续深入和扩展学习了。

第 3 和第 4 区域：规划外部扩展内存资源区域及地址范围。

这两个区域的作用主要是管理和配置外扩内存资源区间，有的朋友可能要问了，哪里来的什么外扩内存资源区间呢？就拿台式计算机来说，如果系统程序的运行非常卡顿，可能是内存容量不足，这时候需要在计算机主板上安装更大容量的内存条，这时候主板就必须要区分不同插槽上的内存条情况及容量。又比如有大量的资料需要存放在计算机硬盘中，这时候就要在主机内“挂”上不止一个硬盘，多个硬盘同时存在的时候也要区分硬盘的主从、分区及大小。这些二次添加的“内存条”和“硬盘”就是“外扩内存资源区域”。

第 3 区域是 Off-chip Code memory，主要是管理外部扩展的 ROM 资源区域的，Keil 软件最多能支持 3 个外部 ROM 扩展区域。需要特别说明的是，STC8H8K64U 单片机是不支持外部扩展 ROM 单元的，所以这里的选项不用填写相关地址参数。如果我们开发的是其他 51 单片机型号且支持外部扩展 ROM 单元，那就可以把外部 ROM 单元的起始编址和容量大小“告诉”Keil，这样就能将这些 ROM 区域统一管理和使用了。

第 4 区域是 Off-chip Xdata memory，主要是管理外部扩展的 RAM 资源区域的，其配置方法与第 3 区域类似，不再赘述。

现在的单片机产品内存资源越做越大，一般的项目应用都是能够满足的，就算是有大容量内存需求，也多是选择大容量内存的单片机芯片即可，很少需要自己外扩内存资源，所以第 3 和第 4 区域的相关参数一般都不用填写，留空即可。

第 5 区域：启用代码分页技术，实现更大的代码空间支持。

我们都知道，MCS-51 内核的单片机受内部总线位宽的影响，通常情况下，其内存资源的寻址范围最大就是 64KB，所以小宇老师之前说 STC8H8K64U 单片机的 ROM 资源已经“打顶”了。但是 64KB 大小的 ROM 空间是不是一定够用呢？肯定不是的！有的项目中的程序代码需要建立中文字库，或者是做一个液晶的菜单内容、图形内容或者动画效果之类的，这些“固有数据”就会很大，几个数组编译下来，不一会儿就把 ROM“吃光了”。那这种情况下的代码大小可能比 64KB 大很多，咋办呢？

这让小宇老师想到了“切西瓜”，夏天到来时，我们都喜欢吃西瓜，冰箱的格子也装不下那么大一个西瓜，那怎么办呢？直接用刀把西瓜切成两半就能放得下了。这个技术其实就是“Code Banking”，这里的“Bank”不是银行的意思，而是一种对代码段进行分页的方法。代码分页的机理就是将代码文件分成小于或等于 64KB 大小的代码段，然后装到不同的 ROM 区域里面去，通过片选的方式实现程序在不同代码空间的跳转。Keil C51 环境下就可以支持这个技术，代码分页支持 2、4、8、16、32 和 64。通过合理的代码分页，可以让系统达到最大 2MB 的代码空间。虽说这个技术确实很“厉害”，但是在本书的项目中，还不会遇到代码大于 64KB 的情况，所以不必分页，这个选项也不用选取。

第 6 区域：添加“far”变量访问支持及保存中断里的扩展 SFR 参数内容。

这个区域的两个选项在 STC8H8K64U 单片机项目中是灰色的状态，意思是不能勾选，所以简单了解即可，“‘far’memory type support”的意思是添加对 far 变量访问的支持，这个“far”也是 C51 的一个扩展关键字。“Save address extension SFR in interrupts”的意思就是在进入中断服务函数之前保存扩展特殊功能寄存器 SFR 中的相关参数。

学完了 Keil C51 环境 Target 选项卡的 6 个区域内容有什么感觉呢？小宇老师的感觉是 Keil C51 环境也得认真学习，只有熟悉了开发环境，才能让我们的开发更为顺利。需要说明的是，Keil C51 环境并不是只为 STC 公司的 51 单片机设计的，这款环境支持众多厂家的成百上千款主流型号单片机产品的开发和调试。所以其中的选项卡覆盖了 MCS-51 内核单片机的诸多功能和参数，在学习软件界面的同时也拓宽了我们对不同厂家 51 单片机性能的认知。所以希望朋友们有空时研究一下 Keil C51 环境的 Help 文档和使用手册，一定可以获得很多知识细节。

9.7 藏匿于存储器单元中的“特殊”参数

说完了 Keil C51 环境中的存储器有关内容，再来看看 STC-ISP 软件中的相关提示。在每次烧录程序时，STC-ISP 工具的右下方调试信息栏中就会显示出很多附加内容，这些内容代表什么含义呢？有的朋友要说：管它呢，这些信息不影响用户编程，我一般就只关心程序下载好了没，下载信息不看也罢！朋友们说的也有道理，对于 STC 公司早期单片机而言，下载信息的内容并不太多，也不复杂，但是对于 STC8 系列来说，可能就有一些内容值得一看了。以 STC8H8K64U 型号的单片机为例，下载调试信息过程如图 9.12 所示。

分析下载信息可以看出，这些内容包含下载过程、单片机硬件选项配置、单片机内部参数等信息。“硬件选项”的相关内容来自于 STC-ISP 软件左侧的硬件功能选项卡，可以通过打钩的方式去配置，但在下载信息里也有一些不是我们配置的内容，比如单片机的固件版本号、内部参考电压值、芯片实际的内部 IRC 振荡器频率值、芯片实际的掉电唤醒定时器的频率值和芯片出厂序列号等。这些内容是从哪里来的呢？难道是在下载的过程中从目标单片机里读出来的吗？的确是的！这些内容就是今天的“主角”，即存储器单元中的“特殊”参数，这些参数包括芯片全球唯一 ID、32K 掉电唤醒定时器时钟频率值、内部参考电压值（即 Bandgap 电压值）和 IRC 内部时钟参数等。

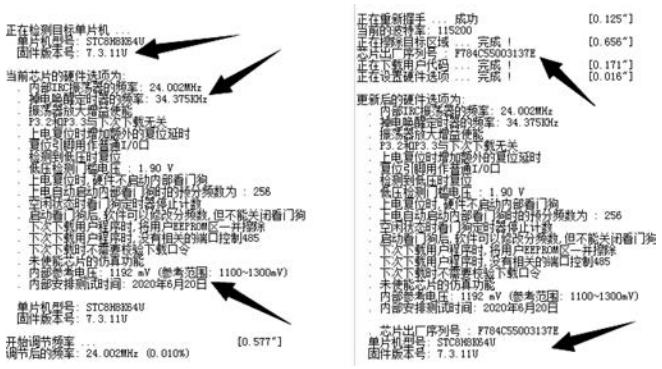


图 9.12 STC-ISP 工具中的下载信息

这些参数存在于单片机的 RAM 区域和 ROM 区域，只需要定义相关类型的指针变量指向特定地址，然后按照顺序把地址中的内容读取出来就可以了。需要特别说明的是，STC8 系列单片机的型号较多，不同系列和型号下的参数地址是不一样的，一定要在编程前进行手册查阅才行。在本节中，我们就选择 STC8H8K64U 单片机作为实验对象，尝试读取该单片机的芯片全球唯一 ID 和内部参考电压值(为了不影响学习脉络和进阶难度梯级，小宇老师将 32K 掉电唤醒定时器时钟频率值和 IRC 内部时钟参数的知识点放在第 10 章中讲解，此处不做展开)。

在实验之前，需要进行实验设计和实验准备，如果读取到了芯片全球唯一 ID 或者内部参考电压值应该怎么看到结果呢？这时候我们学过的 1602 字符型液晶就可以派上用场了，可以将相关信息读取后进行简单转换(变成字符形式)，然后在 1602 液晶上显示出来。实验电路按照图 9.13 进行搭建即可。

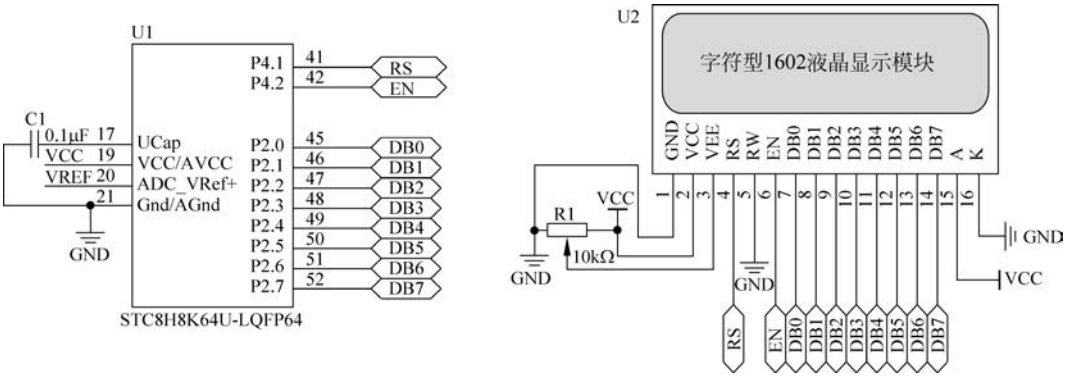


图 9.13 存储器特殊参数读取实验电路原理图

9.7.1 基础项目 A 读取 STC8 系列单片机的“身份证”号

我们第一个要读取的是“芯片全球唯一 ID”，这个 ID 就像是单片机芯片的“身份证”，朋友们可以基于这个固定序列做一些实际应用，比如利用已知芯片的 ID 做数据加密或者程序加密，又或者把这个 ID 当成实际产品的运行许可证，要是 ID 验证不通过就不准给设备升级、不准设备联网、不准设备向外供电、不准设备体现功能，等等。只要加以想象，这个 ID 就能有很多“玩法”。

以 STC8H8K64U 单片机为例，该型号单片机的 ID 同时存在于 RAM 区域和 ROM 区域中，RAM 区域的 ID 参数保存地址为 $(F1)_H$～$(F7)_H$(共 7B)，ROM 区域的 ID 参数保存地址为 $(FDF9)_H$～$(FDFF)_H$(也是 7B)。说到这里，问题就来了，怎么定义参数地址呢？其实很简单，直接用如下两条宏定义语句就可以。

```
#define RAM_AD 0xF1                          //ID 序列在 RAM 空间的存放地址
#define ROM_AD 0xFDF9                        //ID 序列在 ROM 空间的存放地址
```

那怎么读取这 7B 的内容呢？当然是要用 C51 语言中的“利器”：指针。但是今天要用的指针有点儿“讲究”，我们必须要区分指针变量的存储类型，如果是定义指向 RAM 区域的指针，就要用到 idata 关键字去修饰，如果是定义指向 ROM 区域的指针，就要用到 code 关键字去修饰。为了方便程序的操作，在程序中定义如下的 RAM_P 和 ROM_P 指针。

```
u8  idata *RAM_P;                            //定义指针 RAM_P,指向 idata 区域
u8  code *ROM_P;                             //定义指针 ROM_P,指向 code 区域
```

有了指针可能还不行，虽说指针可以指向 ID 序列所在的地址，但是取回来的 7B 总要有地方“安放”才可以。那也好办，直接定义如下所示的 AID[]和 OID[]数组就行。这两个数组分别存放 RAM 和 ROM 区域中取回的 ID 序列数据即可。

```
u16 AID[7] = {0x00,0x00,0x00,0x00,0x00,0x00,0x00};
u16 OID[7] = {0x00,0x00,0x00,0x00,0x00,0x00,0x00};
```

ID 序列参数地址明确了，指针也定义好了，数据取回来也有地方存放，这就算是具备基础条件了，剩下就是理清编程思路了。单片机上电后，需要进行相关资源的初始化（例如液晶的初始化），然后定义指针并赋值，让指针指向 ID 序列参数的地址，然后利用两个 for 循环实现指针自增和数据存储，最后把数据显示到 1602 液晶上就可以了。以 RAM_P 指针为例，先让 RAM_P 指针指向 RAM 区域的$(F1)_H$ 地址，然后在 for 循环中实现将$(F1)_H$ 地址中的数据存放到 AID[0]，然后指针自增，又把$(F2)_H$ 地址中的数据存放到 AID[1]，ROM_P 指针也是一样的道理。这样一来，AID[]和 OID[]数组中就可以装满 ID 序列数据了，剩下的就是对数据稍作处理，变成“字符形式”再送到 1602 液晶中显示即可。

理清了思路就开始编写程序吧！利用 C51 语言编写的源码如下。

```
//芯片型号: STC8H8K64U(程序微调后可移植至 STC8A/F/C/G/H 系列单片机)
//时钟说明: 单片机片内高速 24MHz 时钟
/*------------------------------------------------------------------*/
#include "STC8H.h"                           //主控芯片的头文件
/*--------------------------- 常用数据类型定义 -----------------------*/
【略】为节省篇幅,相似定义参见相关章节或源码工程即可
/*--------------------------- 端口/引脚定义区域 ----------------------*/
sbit  LCDRS = P4^1;                          //LCD1602 数据/命令选择端口
sbit  LCDEN = P4^2;                          //LCD1602 使能信号端口
#define  LCDDATA  P2                         //LCD1602 数据端口 D0～D7
/*--------------------------- 用户自定义数据区域 ---------------------*/
u16 AID[7] = {0x00,0x00,0x00,0x00,0x00,0x00,0x00};
//用于存放 RAM 中读出的 ID 序列(7B)
u16 OID[7] = {0x00,0x00,0x00,0x00,0x00,0x00,0x00};
//用于存放 ROM 中读出的 ID 序列(7B)
#define  RAM_AD 0xF1                         //ID 序列在 RAM 空间的存放地址
#define  ROM_AD 0xFDF9                       //ID 序列在 ROM 空间的存放地址
/*------------------------------ 函数声明区域 ------------------------*/
void delay(u16 Count);                       //延时函数
void LCD1602_Write(u8 cmdordata,u8 writetype);   //写入液晶模组命令或数据函数
void LCD1602_init(void);                     //LCD1602 初始化函数
/*------------------------------ 主函数区域 --------------------------*/
void main(void)
{
    u8  idata  *RAM_P;                       //定义指针 RAM_P,指向 idata 区域
    u8  code   *ROM_P;                       //定义指针 ROM_P,指向 code 区域
```

```
    u8 i;                                          //定义循环控制变量 i
    //配置 P4.1 - 2 为准双向/弱上拉模式
    P4M0& = 0xF9;                                  //P4M0.1 - 2 = 0
    P4M1& = 0xF9;                                  //P4M1.1 - 2 = 0
    //配置 P2 为准双向/弱上拉模式
    P2M0 = 0x00;                                   //P2M0.0 - 7 = 0
    P2M1 = 0x00;                                   //P2M1.0 - 7 = 0
    delay(5);                                      //等待 I/O 模式配置稳定
    LCD1602_init();                                //LCD1602 初始化
    RAM_P = RAM_AD;                                //让指针 RAM_P 指向 0xF1
    for(i = 0;i < 7;i++)                           //读 7B(高字节在前)
    {
        AID[i] = * RAM_P++;                        //取回 RAM 中的 ID 序列存入 RAM_ID[]
    }
    ROM_P = ROM_AD;                                //让指针 ROM_P 指向 0xFDF9
    for(i = 0;i < 7;i++)                           //读 7B(高字节在前)
    {
        OID[i] = * ROM_P++;                        //取回 ROM 中的 ID 序列存入 ROM_ID[]
    }
    LCD1602_Write(0x80,0);                         //选择第一行
    LCD1602_Write('A',1);                          //写入字符 A,表示 RAM 中取出的 ID 序列
    LCD1602_Write(':',1);                          //写入字符冒号,用于区分数据
    for(i = 0;i < 7;i++)
    {
        LCD1602_Write((AID[i]/16 > 9)?AID[i]/16 - 10 + 'A':AID[i]/16 + '0',1);
        //写入单字节高 4 位,通过三目运算转换为 ASCII 形式
        LCD1602_Write((AID[i] % 16 > 9)?AID[i] % 16 - 10 + 'A':AID[i] % 16 + '0',1);
        //写入单字节低 4 位,通过三目运算转换为 ASCII 形式
    }
    LCD1602_Write(0xC0,0);                         //选择第二行
    LCD1602_Write('O',1);                          //写入字符 O,表示 ROM 中取出的 ID 序列
    LCD1602_Write(':',1);                          //写入字符冒号,用于区分数据
    for(i = 0;i < 7;i++)
    {
        LCD1602_Write((OID[i]/16 > 9)?OID[i]/16 - 10 + 'A':OID[i]/16 + '0',1);
        //写入单字节高 4 位,通过三目运算转换为 ASCII 形式
        LCD1602_Write((OID[i] % 16 > 9)?OID[i] % 16 - 10 + 'A':OID[i] % 16 + '0',1);
        //写入单字节低 4 位,通过三目运算转换为 ASCII 形式
    }
    while(1);                                      //程序停止于此
}
/ ********************************************************************** /
【略】为节省篇幅,相似函数参见相关章节源码即可
void  delay(u16 Count){}                           //延时函数
void  LCD1602_Write(u8 cmdordata,u8 writetype){} //写入液晶模组命令或数据函数
void  LCD1602_init(void){}                         //LCD1602 初始化函数
```

通读程序也不是很难,稍微难以理解的地方可能是 ID 数据的显示处理部分。因为取到的 ID 数据是 7B,类似于$(F62802BCBB766F)_H$的形式,那第一字节内容就是$(F6)_H$,这里的“F”和“6”是十六进制的数值形式,要想显示到 1602 液晶上就必须要转换成对应的 ASCII 码形式(即十六进制内容转换成对应字符形式),$(F)_H$就是十进制的 15,那这个 15 怎么和 ASCII 码的字母“F”对应呢?很简单,只需要把 15－10 再加上“A”字母的 ASCII 码(即 65)就可以得到字母“F”的 ASCII 码了(即 15－10＋65,就是 70),这种方法适用于对待数值大于 9 的十六进制数据转换。要是十六进制数值为 0～9 怎么办呢?其实也是类似的,以$(6)_H$为例,要想转换成 ASCII 码形式的“6”(即 54)只需要加上“0”的 ASCII 码(即 48)就可以了。

找到了数据转换的方法，程序就很好写了，可以用 if-else 形式的 C51 语句去写，但是稍显麻烦，所以程序中用到了 C51 语言中的三目运算符(也可称为三元运算符)，其书写形式为“(表达式 1)?(表达式 2):(表达式 3);”，这个语句中有 3 个表达式内容，执行语句时先判断表达式 1 的值，如果为真就执行表达式 2，反之执行表达式 3。这种语句形式非常实用，按照之前讲解的转换方法，就可以轻松得到如下语句。

```
LCD1602_Write((AID[i]/16>9)?AID[i]/16-10+'A':AID[i]/16+'0',1);
//写入单字节高 4 位,通过三目运算转换为 ASCII 形式
LCD1602_Write((AID[i]%16>9)?AID[i]%16-10+'A':AID[i]%16+'0',1);
//写入单字节低 4 位,通过三目运算转换为 ASCII 形式
```

第一句话中的“(AID[i]/16>9)”就是通过除以 16 的方法取得第一字节的高 4 位(假设第一字节是$(F6)_H$，除以 16 后就得到$(F)_H$)，如果这个十六进制数是大于 9 的，那就执行“AID[i]/16－10＋'A'”(即转换为 A～F 字母形式字符)，反之执行“AID[i]/16＋'0'”(即转换为 0～9 数字形式字符)。最后通过循环，把这些转换后的字符送到 1602 液晶中显示即可。不管是 RAM 区域还是 ROM 区域中的 ID 序列，其转换方法是一致的，将程序编译后烧录到单片机中，可以得到如图 9.14 所示运行效果。液晶第一行的“A:”即为 RAM 区域读取的 ID 序列，第二行的“O:”即为 ROM 区域读取的 ID 序列，通过对比，这个序列号刚好就是如图 9.12 所示的芯片出厂序列号“F784C55003137E”。

图 9.14 单片机的“身份证”号实验效果

9.7.2 基础项目 B 片内 Bandgap 电压是多少

第二个要读取的参数就是片内参考 Bandgap 电压值了，什么是“Bandgap 电压”呢？简单来说就是带隙基准电压单元的简略叫法(英文全称是 Bandgap Voltage Reference)。该电压由单片机内部电路产生，电压值一般恒定为 1.19V，不管单片机供电电压和工作环境温度如何变化，其误差范围在 1%左右(一般为 1.11～1.3V)，通常情况下可以认为该电压是稳定不变的“基准”。该电压值可以作为 STC8 系列单片机片内比较器资源的输入电压量之一，也可以当作 ADC 模数转换单元的基准电压。通俗地讲，可以把 Bandgap 电压值当作量化功能中的“一把尺子”去用，用“固定”对比“变动”，用“已知”量化“未知”。

带隙基准的具体设计就稍微复杂些，在这里就简单了解下单元构成即可。我们可以把带隙基准单元看作一个与温度变化成正比的电压单元和一个与温度变化成反比的电压单元的组合单元，当温度变化时，两个单元的变化增量相互抵消，这样一来就实现了电压基准，这个电压约为 1.19V，因其电压值与硅材料的带隙电压差不多，所以也将其称为“带隙基准”，但是这个叫法也不完全对，因为现在很多 Bandgap 单元并不是利用带隙电压产生的，甚至允许 Bandgap 输出电压与带隙电压不相等，所以 STC8H 系列单片机手册中将其称为“内部参考信号源电压”。

还是以 STC8H8K64U 单片机为例，该型号单片机的 Bandgap 值同时存在于 RAM 区域和 ROM 区域中，RAM 区域的 Bandgap 参数保存地址为$(EF)_H$～$(F0)_H$(共 2B)，ROM 区域的 ID 参数保存地址为$(FDF7)_H$～$(FDF8)_H$(也是 2B)。我们在编程时可以用宏定义方式将其写为如下两条语句。

```
#define RAM_AD 0xEF                    //Bandgap 参数在 RAM 空间的存放地址
#define ROM_AD 0xFDF7                  //Bandgap 参数在 ROM 空间的存放地址
```

类似地，需要定义指向 RAM 区域的指针“RAM_P”和指向 ROM 区域的指针“ROM_P”，取回来的 4B 数据(RAM 区域中有 2B，ROM 区域中也有 2B)存放到一个数组中即可，可以定义如下语句。

```
u16 BGV[4]={0x00,0x00,0x00,0x00};      //Bandgap 参数暂存数组
```

有了特殊地址、指针和数组还不行，单片机上电时，还需要进行相关资源的初始化（例如液晶的初始化），然后给指针赋值，让指针指向 Bandgap 参数的地址，然后利用 for 循环实现指针自增和数据存储，最后把数据显示到 1602 液晶上就可以了。

以 RAM_P 指针为例，先让 RAM_P 指针指向 RAM 区域的$(EF)_H$地址，然后在 for 循环中实现将$(EF)_H$地址中的数据存放到 BGV[0]，然后指针自增，又把$(F0)_H$地址中的数据存放到 BGV[1]，ROM_P 指针也是类似的，取出的数据放在 BGV[2]和 BGV[3]。将编程思想转换为实际程序，就可以得到如下语句。

```
RAM_P = RAM_AD;                                    //让指针 RAM_P 指向 0xEF
for(i = 0;i < 2;i++)                               //读 2B(高字节在前)
{
    BGV[i] = * RAM_P++;                            //取回 RAM 中的 Bandgap 存入 BGV[0]和 BGV[1]
}
```

这样一来，RAM 和 ROM 区域中的 Bandgap 参数都得到了，剩下的就是数据处理，通过数据拼合和取位运算得到 Bandgap 参数的万位、千位、百位、十位和个位即可，最后将其变成“字符形式”再送到 1602 液晶中显示。需要特别说明的是，内存中有关 Bandgap 参数的两字节是高位在前低位在后，读取出来的数据单位是 mV，程序中可以在千位和百位之间人工添加个“小数点”，这样看起来就是××.×××V 了。

理清了思路就开始编写程序吧！利用 C51 语言编写的源码如下。

```
//芯片型号: STC8H8K64U(程序微调后可移植至 STC8A/F/C/G/H 系列单片机)
//时钟说明: 单片机片内高速 24MHz 时钟
/*************************************************************************/
#include "STC8H.h"                                 //主控芯片的头文件
/***************************** 常用数据类型定义 *****************************/
【略】为节省篇幅,相似定义参见相关章节或源码工程即可
/***************************** 端口/引脚定义区域 ****************************/
sbit   LCDRS = P4^1;                               //LCD1602 数据/命令选择端口
sbit   LCDEN = P4^2;                               //LCD1602 使能信号端口
#define  LCDDATA  P2                               //LCD1602 数据端口 D0～D7
/***************************** 用户自定义数据区域 ***************************/
u16 BGV[4] = {0x00,0x00,0x00,0x00};                //Bandgap 参数暂存数组
#define  RAM_AD 0xEF                               //Bandgap 参数在 RAM 空间的存放地址
#define  ROM_AD 0xFDF7                             //Bandgap 参数在 ROM 空间的存放地址
//注意: ROM 中的 BGV 参数需要在 STC-ISP 下载时添加"重要测试参数"
/***************************** 函数声明区域 *********************************/
void delay(u16 Count);                             //延时函数
void LCD1602_Write(u8 cmdordata,u8 writetype);     //写入液晶模组命令或数据函数
void LCD1602_init(void);                           //LCD1602 初始化函数
/***************************** 主函数区域 ***********************************/
void main(void)
{
    u8  idata  * RAM_P;                            //定义指针 RAM_P,指向 idata 区域
    u8  code   * ROM_P;                            //定义指针 ROM_P,指向 code 区域
    u8 i;                                          //定义循环控制变量 i
    u32 RAM_BGV,ROM_BGV;                           //定义变量 RAM_BGV 和 ROM_BGV
    //配置 P4.1-2 为准双向/弱上拉模式
    P4M0& = 0xF9;                                  //P4M0.1-2 = 0
    P4M1& = 0xF9;                                  //P4M1.1-2 = 0
    //配置 P2 为准双向/弱上拉模式
    P2M0 = 0x00;                                   //P2M0.0-7 = 0
    P2M1 = 0x00;                                   //P2M1.0-7 = 0
    delay(5);                                      //等待 I/O 模式配置稳定
```

```
    LCD1602_init();                                  //LCD1602 初始化
    RAM_P = RAM_AD;                                  //让指针 RAM_P 指向 0xEF
    for(i = 0;i < 2;i++)                             //读 2B(高字节在前)
    {
        BGV[i] = * RAM_P++;                          //取回 RAM 中的 Bandgap 存入 BGV[0]和 BGV[1]
    }
    ROM_P = ROM_AD;                                  //让指针 ROM_P 指向 0xFDF7
    for(i = 0;i < 2;i++)                             //读 2B(高字节在前)
    {
        BGV[i + 2] = * ROM_P++;                      //取回 ROM 中的 Bandgap 存入 BGV[2]和 BGV[3]
    }
    LCD1602_Write(0x80,0);                           //选择第一行
    LCD1602_Write('A',1);                            //写入字符 A,表示 RAM 中取出的 BGV 参数
    LCD1602_Write(':',1);                            //写入字符冒号,用于区分数据
    LCD1602_Write(0x85,0);                           //选择第一行第 5 个位置
    BGV[0] = BGV[0]<< 8;                             //将 BGV[0]数据内容移到高 8 位
    RAM_BGV = BGV[0]|BGV[1];                         //将 BGV[0]与 BGV[1]内容拼合后给 RAM_BGV
    LCD1602_Write(RAM_BGV/10000 + '0',1);            //写入万位
    LCD1602_Write(RAM_BGV % 10000/1000 + '0',1);     //写入千位
    LCD1602_Write('.',1);                            //写入小数点
    LCD1602_Write(RAM_BGV % 1000/100 + '0',1);       //写入百位
    LCD1602_Write(RAM_BGV % 1000 % 100/10 + '0',1);  //写入十位
    LCD1602_Write(RAM_BGV % 10 + '0',1);             //写入个位
    LCD1602_Write(' ',1);                            //写入空格
    LCD1602_Write('V',1);                            //写入电压单位(V)
    LCD1602_Write(0xC0,0);                           //选择第二行
    LCD1602_Write('O',1);                            //写入字符 O,表示 ROM 中取出的 BGV 参数
    LCD1602_Write(':',1);                            //写入字符冒号,用于区分数据
    LCD1602_Write(0xC5,0);                           //选择第二行第 5 个位置
    BGV[2] = BGV[2]<< 8;                             //将 BGV[2]数据内容移到高 8 位
    ROM_BGV = BGV[2]|BGV[3];                         //将 BGV[2]与 BGV[3]内容拼合后给 ROM_BGV
    LCD1602_Write(ROM_BGV/10000 + '0',1);            //写入万位
    LCD1602_Write(ROM_BGV % 10000/1000 + '0',1);     //写入千位
    LCD1602_Write('.',1);                            //写入小数点
    LCD1602_Write(ROM_BGV % 1000/100 + '0',1);       //写入百位
    LCD1602_Write(ROM_BGV % 1000 % 100/10 + '0',1);  //写入十位
    LCD1602_Write(ROM_BGV % 10 + '0',1);             //写入个位
    LCD1602_Write(' ',1);                            //写入空格
    LCD1602_Write('V',1);                            //写入电压单位(伏特)
    while(1);                                        //程序停止于此
}
/*********************************************************************/
【略】为节省篇幅,相似函数参见相关章节源码即可
void  delay(u16 Count){}                             //延时函数
void  LCD1602_Write(u8 cmdordata,u8 writetype){}     //写入液晶模组命令或数据函数
void  LCD1602_init(void){}                           //LCD1602 初始化函数
```

这个程序也不难，重点的语句还是在 Bandgap 参数的数据处理部分。因为取回的两字节数据并不是分立无关的(这一点有别于 9.7.1 节基础项目 A 实验中的 ID 数据)，这两字节是实际电压值的“高字节 XX＋低字节 YY”形式，所以必须要把头尾结合，重新组装为“XXYY”的形式，这个组装很简单，程序中通过如下两个语句实现。

```
BGV[0] = BGV[0]<< 8;                                 //将 BGV[0]数据内容移到高 8 位
RAM_BGV = BGV[0]|BGV[1];                             //将 BGV[0]与 BGV[1]内容拼合后给 RAM_BGV
```

首先把 BGV[0]中的数据通过按位左移运算“<<”向左移动 8 位，这时候的 BGV[0]数据就变成了“XX00”的形式，然后通过按位或运算“|”将 BGV[0]中的“XX00”数据及 BGV[1]的“YY”数据

进行按位或运算，得到“XXYY”数据并赋值给 RAM_BGV 变量即可。

接下来的数据转换就很简单了，就是对 RAM_BGV 变量进行取位运算，依次取出万位（即 RAM_BGV/10000）、千位（即 RAM_BGV%10000/1000）、百位（即 RAM_BGV%1000/100）、十位（即 RAM_BGV%1000%100/10）和个位（即 RAM_BGV%10）即可。需要说明的是，取出来的数据值域是 0～9，但是数字的 0～9 与字符形式的 0～9 是不一样的，所以需要在取位运算后面加上一个字符“0”，相当于加上了“0”的 ASCII 码，这样一来就可以把数字的 0～9 变成字符形式的 0～9 了。

将程序正确编译及下载后可以在 1602 液晶上看到如图 9.15 所示效果，小宇老师当场就吓蒙了，液晶第一行显示的“A：01.192V”是正常的，意思就是从 RAM 区域得到的 Bandgap 电压为 1.192V，但是液晶第二行显示的电压难道是 65.535V？这显然不对，要是 Bandgap 电压能上 60V，芯片早就“浓烟滚滚”了。

图 9.15 异常情况下的 Bandgap 电压实验效果

看到现象后，我进行了反思，为什么 RAM 区域取出的数据是对的，而在 ROM 区域中取出的数据出错了呢？假设程序的方法错了，那为什么之前的 ID 数据实验非常成功呢？在困惑的时候我首先想到了 STC8 的官方数据手册。果然，手册中给出了非常明确的 3 条说明。

第一，由于 RAM 中的“特殊”参数可能被人为或者误修改，所以 STC 公司不建议我们读取 RAM 区域中的参数，最好是去读 ROM 区域中的参数。特别是在使用 ID 数据进行程序加密的时候，ROM 区域中的 ID 数据较为稳定。

第二，由于某些 STC8 系列单片机型号中的 EEPROM 大小可以由用户调整（这部分的内容会在第 21 章中展开讲解），这样一来就有可能占用或者覆盖“特殊”参数原有的 ROM 空间，朋友们在这种情况下一定要考虑实际配置和内存划分。

第三，在默认情况下，ROM 区域中只有单片机全球唯一 ID 的数据（也就是我们做的单片机“身份证”数据内容），而 Bandgap 电压值、32kHz 掉电唤醒定时器的频率值以及 IRC 内部时钟频率值参数都是没有的，需要在程序下载时配置 STC-ISP 软件的相关选项才可以。小宇老师拍拍脑瓜，原来如此！那就按照如图 9.16 所示内容配置 STC-ISP 软件的硬件选项吧！我们需要勾选“在程序区的结束处添加重要测试参数”这一复选框。

勾选完毕之后，再次下载程序文件（不用对之前的程序代码进行任何改动），当我看到如图 9.17 所示的 1602 液晶屏的内容时，终于舒了一口气！示数正常了，RAM 区域和 ROM 区域读取出来的示数均是 1.192V。

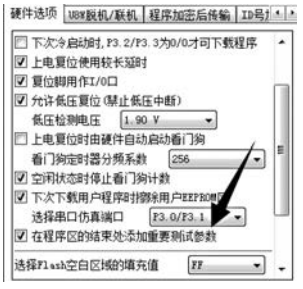


图 9.16 勾选硬件选项中的测试参数项

图 9.17 正常情况下的 Bandgap 电压实验效果

实验做到这里就算结束了，但是本章的两个实验的本质并不是读取 ID 数据或是 Bandgap 电压。我们应该深入理解 STC8 系列单片机内存划分、区域功能、寻址方式、功能特点、C51 语言扩展关键字、C51 语言存储类型、Keil C51 环境编译模式及存储器相关配置、STC-ISP 软件硬件选项配置等内容。希望朋友们基于本章内容再做深化与扩展，在以后的工作中遇到不熟悉的单片机产品时也能站在 STC8 系列单片机的“肩膀”上迅速拿下其他单片机。