第29卷 第8期 计 算 机 工 程 2003年5月
Vol.29 № 8 Computer Engineering May 2003
· 开发研究与设计技术 · 文章编号：1000—3428(2003)08—0183—03 文献标识码：A 中图分类号：TP311.11

MCS-51程序空间扩展原理及编译器优化

周敬利，卓 越，余胜生
(华中科技大学计算机学院，武汉 430074)

摘 要：讨论了MCS-51系列单片机程序空间扩展的原理，包括硬件与编译器两个方面，并提出一种编译器优化方案。该方案在Keil仿真器上检验并通过。

关键词：C51编译软件；MCS-51；Bank Switching

Mechanism of Expansion of Code Space in MCS-51 and Optimization of the Compiler

ZHOU Jingli, ZHUO Yue, YU Shengsheng
(College of Computer Science, Huazhong Univ. of Sci. & Tech.,Wuhan 430074)

【Abstract】 This paper discusses the mechanism of expansion of code space in MCS-51 microcontrollers in aspects of both hardware and the compiler, and puts up a method of optimization.The schme has been tested in the Keil simulator.

【Key words】 C51; MCS-51; Bank Switching

8051系列微处理器采用基于简化的嵌入式控制系统结构，被广泛应用于从军事到自动控制再到PC机的各种应用系统上。51系列单片机最大的优势在于低廉的价格。但随着各种控制及应用程序的发展，它的一个先天缺陷非常明显地暴露了出来。那就是它的寻址空间只有64kB，这是它的指令集决定的。作为世界上最先进的C51编译软件的提供商，德国Keil公司提供了一个解决方案，称作Bank Switching。通过一些附加硬件和程序，配合Keil的编译器，C51的寻址空间理论上可扩展至16MB，几乎可以满足所用控制需要。由于Keil只提供使用文档，而并未提供关于其内部实现机制的说明文档，文章中几乎所有的分析源自Keil提供的汇编程序源代码。本文先分析它的内部机制，然后提出一个改进方案。

1 Bank Switching的内部原理

Bank Switching 的思想是将大程序分块，每块称为一个Bank。下面主要以8个Bank，每个大小32kB为例解释其原理。需要注意的是程序空间的扩展是有代价的，有时候一些附加指令必须被执行。

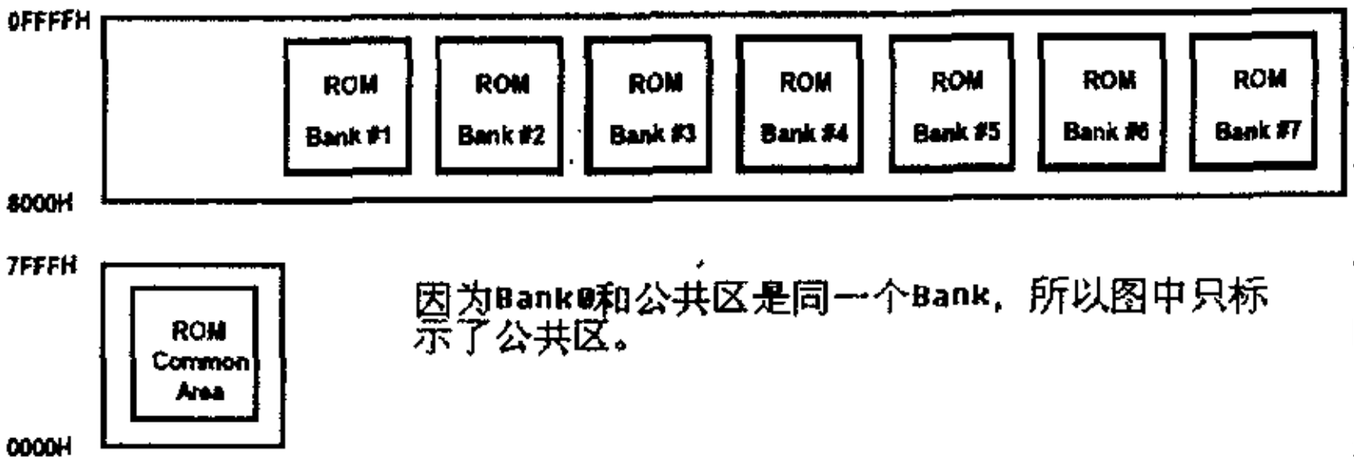

图1 Bank的地址映射

首先，程序员把程序分块后，编号从0到7，使用Keil编译成功后以32kB为单位将它们按顺序烧入256kB的EPROM，其中最低的32kB，也就是Bank0称为公共区，存放中断矢量、Bank Switching控制指令、最常用函数及主函数。它们之间的逻辑关系和地址范围如图1所示。注意：公共区与其他任何一个Bank合起来应该构成一个完整的64kB空间，程序中Bank1~Bank7均采用同样的地址空间。

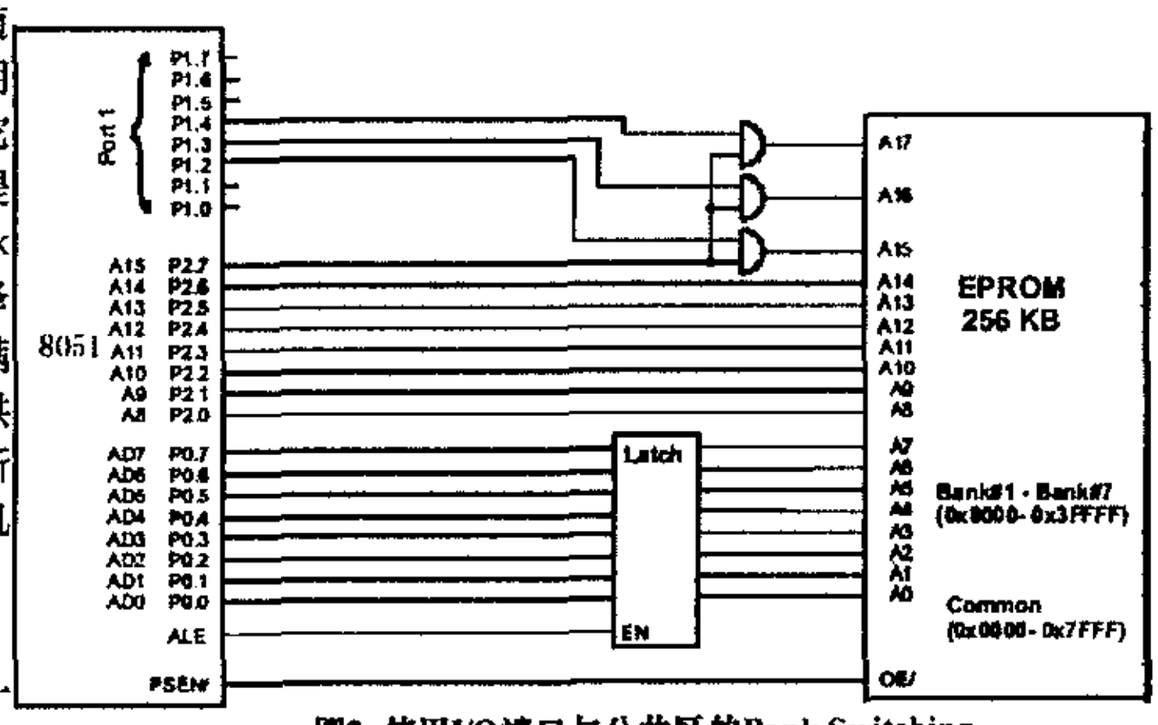

图2 使用I/O端口与公共区的Bank Switching

因为Bank0和公共区是同一个Bank，所以图中只标示了公共区。

另外，需要从I/O端口引出3根线与地址线最高位A_{15}进行“AND”操作后连接到EPROM的最高3位地址线输入(如图2,由于篇幅关系，省去了与RAM的连线)，它们将成为扩展地址线的最高3位。从图上可以看出：当A15=0时，P1.4，P1.3，P1.2 不起作用，EPROM中0~(32k-1)B的地址空间被选中；若A15=1，P1.4，P1.3，P1.2值为i(i在0到7之间)，那么EPROM中i*32k~[(i+1)*32k-1]B的地址空间被选中。

假如采用更一般的划分方法，比如说公共区是48kB，其他每个Bank是16kB。那么控制线就不只是A15，而应该是A15和A14的“与”。连接方式也要复杂一些，因为Bank1~Bank7在EPROM中的基地址不再是48kB的整数倍，而是[48+(i-1)*16]kB。

作者简介：周敬利(1946－)，女，教授、博导，主要研究多媒体技术；卓 越，硕士生；余胜生，教授、博导

收稿日期：2002-06-06

但Bank Switching的最重要部分并不在硬件，而是编译软件的支持，这是因为C51的指令集是16位的，不能生成超过64kB的跳转指令，这就需要额外指令。分析图1与图2可以发现：

(1)一旦P1口的输出确定以后，EPROM中就有一个32kB的空间被选中，而在这32kB空间中，跳转指令是可以正常工作的。这就是说在同一个Bank内部，函数之间的调用及跳转不需要额外指令。

(2)从各个Bank中调用公共区的函数，这时编译器生成的LCALL addr16指令中代表16位跳转地址的addr16的最高位A15一定是0，因为公共区只有32kB。这样一来，根据刚才的讨论,EPROM自动选择最低的32kB，也不需要额外的指令。

(3)剩下的两种情况：无论是Bank1~Bank7之间的调用或者从公共区调用其余Bank中的函数，均需要额外的流程，我们把这种情况称为跨Bank调用，而把Bank内部的函数调用及从Bank调用公共区的函数调用统称为Bank内部调用。下面将分析跨Bank调用。

整体上看，它牵涉到3个部分：跳转表、?BANK?SELECT段和?BANK?SWITCH段，它们均位于公共区，其中后面两个段是Keil提供的汇编源文件L51_BANK.A51宏展开后的主要部分。?BANK?SELECT段作一些返回时用到地址的保存以及目标地址的压栈工作，而?BANK?SWITCH段向P1口输出电信号，实现实际的跳转。

假设一个最复杂的情况，从Bank1中调用Bank2中的函数，那总体的流程如图3所示。首先，主调函数的返回地址压栈；其次，跳至跳转表；然后，进入?BANK?SELEC段的标志?B_BANK2处(之所以为2是因为到达的Bank是Bank2)，将返回时所需的?BANK?SWITCH段的标志?B_SWITCH1(之所以为1是因为要返回的Bank是Bank1)以及目标函数地址压栈；接着，程序跳至?BANK?SWITCH段的标志?B_SWITCH2处，这时Port1口产生正确的输出；之后，程序进入目标Bank；最后，函数返回时再次进入?BANK?SWITCH段的标志?B_SWITCH1处，P1口输出选中Bank1的地址信号，程序返回原来的Bank。

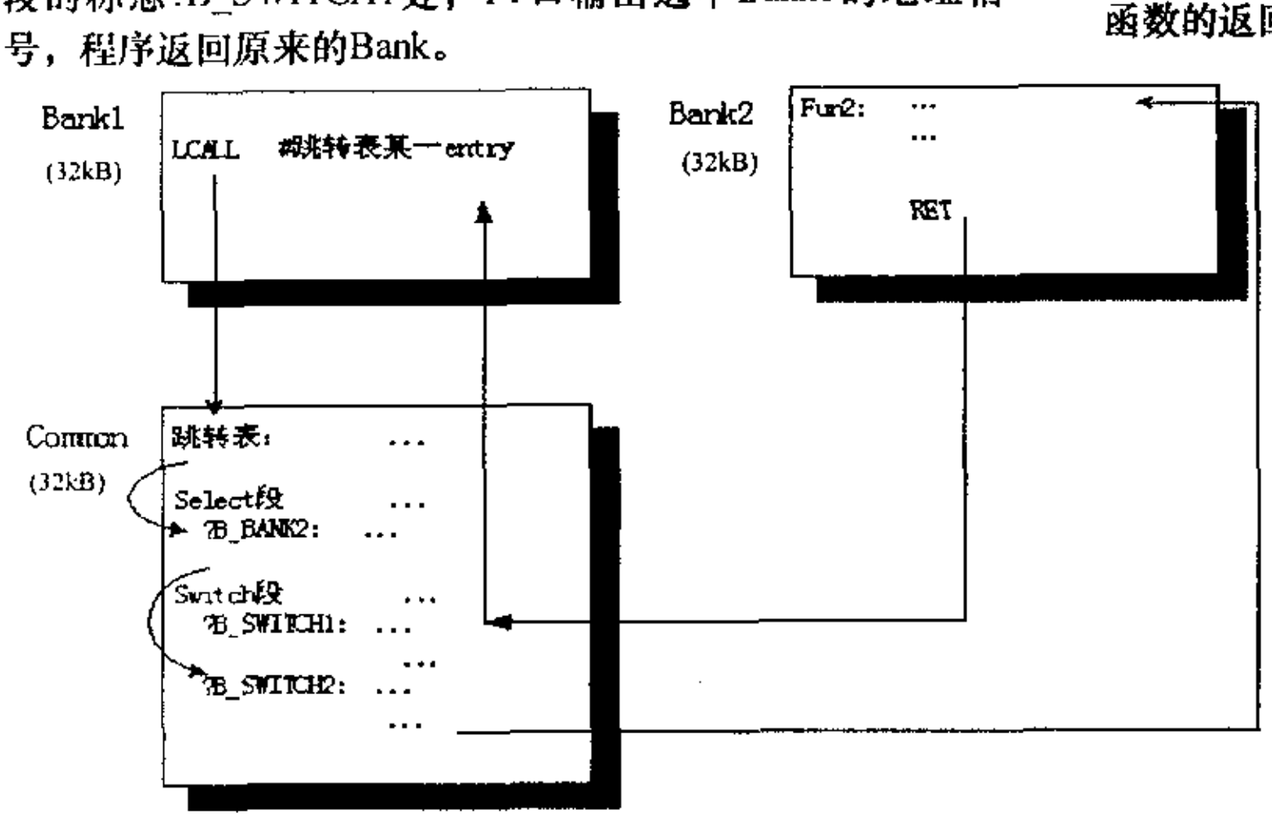

图3 跨Bank函数调用的流程

1.1 跳转表

为了实现跨Bank调用，Keil首先会扫描源程序以发现被跨Bank调用的函数，并为每一个这样的函数写了一个跳转入口(entry)，合起来成为一个跳转表，放置在公共区。每当在程序的任何一个地方需要跨Bank调用函数的时候，它就先跳转到这个地方。跳转表很简单，每个函数占用两行：第一行是把这个函数的地址(在各自Bank中的相对地址0x8000~0xffff)写入DPTR(16位寄存器)，将来要压入堆栈。第二行是跳转到?BANK?SELECT段的对应的标志?BANK&N处(N为该函数所在的Bank号)。

1.2 ?BANK?SELECT段

?BANK?SELECT段先将转向当前Bank所需的?B_SWITCH&N的地址压栈(用在函数返回时回到正确的Bank)，然后将目标函数入口地址压栈。这时栈最满，它的情况如图4。

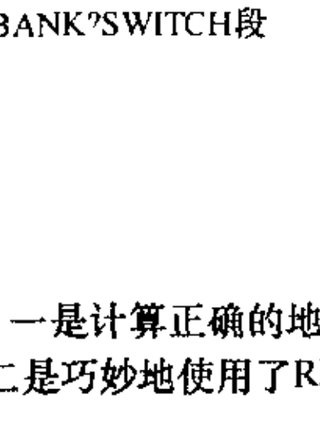

图4 ?BANK?SELECT段

宏展开并化简后，?BANK?SELECT段的大致结构如下：

```
?B_BANK0: MOV   A,?B_CURRENTBANK
          ANL   A,#?B_MASK
          RL    A
          PUSH  ACC
          MOV   A,#HIGH ?BANK?SWITCH
          PUSH  ACC ;将转到当前Bank所需的?BANK?SWITCH段的?B_SWITCH&N标记的地址压栈，为返回作准备。
          PUSH  DPL     ;目标函数地址压栈。
          PUSH  DPH     ; 目标函数地址压栈
          LJMP  ?B_SWITCH0  ; 转入?BANK?SWITCH段
?B_BANK1. ……
          LJMP  ?B_SWITCH1
?B_BANK2:       ……
……
```

1.3 ?BANK?SWITCH段

?BANK?SWITCH段有两个功能：一是计算正确的地址值并输出到P1口以选择所需的Bank。二是巧妙地使用了RET指令，这个指令又有两个功能：

(1)当处于调用过程中时，从栈里弹出的目标函数的地址，这其实是对RET功能的非正常使用；

(2)当处于函数返回的过程中时，从栈里弹出的是主调函数的返回地址。

经过化简后，?BANK?SWITCH段的大致机构如下：

```
?B_SWITCH0 :  ANL  ?B_CURRENTBANK,#(BANK0 OR NOT ?B_MASK)；产生进入Bank0的地址线输出。
              RET；使用?BANK?SELECT段压入堆栈的目标函数地址进入目标函数。
                    ORG   1*8
?B_SWITCH1:   ORL   ?B_CURRENTBANK,#?B_MASK
              ANL   ?B_CURRENTBANK,#(BANK1 OR NOT ?B_MASK)
                    RET
                    ORG   2*8
              ……
```

1.4 其他需要注意的细节

目标函数段(本例子中是Bank2里的Func2)与正常的子程序段没有任何区别。唯一需要注意的是，当执行它的最后一条指令RET时，它不是不可能直接返回主调函数。这时栈顶处的两个字节是?B_SWITCH&N的地址值，下面的两个字节才是主调函数的返回地址。

之所以将?BANK?SELECT、?BANK?SWITCH这两个段与跳转表分开，原因在于?BANK?SELECT、?BANK?SWITCH这两个段是L51_bank.a51直接汇编后的结果，不需要牵涉到其他的模块；而跳转表的生成需要扫描各个模块以

发现被跨Bank调用的函数。

而之所以将 ?BANK?SELECT、?BANK?SWITCH 分开，是因为?BANK?SWITCH 要被使用两次。

2 对Bank Switching的改进及结果分析

经过前面的分析，我们知道跨Bank调用的其中一种情况是从公共区调用其余Bank中的函数。这种情况是最经常发生的，几乎占到所有跨Bank调用的70%，假如能够提高这种情况的执行效率，将明显地提高使用Bank Switching的程序的效率。

仔细分析这种情况，可以清楚地看出它有一个独特的地方，那就是它虽然在调用目标函数的过程中需要输出地址选择信号，但它在返回时是不需要的。这是因为主调函数的返回地址的最高位是0，无论P1.4,P1.3,P1.2值为多少，EPROM中始终是最低的32kB空间被选中。因此，当处在调用过程中，进入?BANK?SELECT段时就不需要保存?B_SWITCH0的地址，而返回时也不再经过?BANK?SWITCH 段，直接返回主调函数。

修改的方法也并不复杂，L51_BANK.A51所有的内容都不需要更改，变动发生在编译器生成转跳表时。这时候：(1)需要检查一下是否主调函数在公共区而目标函数在Bank区(这一点是很容易做到的，因为Keil编译器总是会生成函数调用的关系树)，如果是这样，就为该函数增加一个entry，该entry中原先的第二条语句不再跳到?BANK?SELECT 段，而是直接进入?BANK?SWITCH 段。另外，在该entry的第一条语句之后加入两条指令：PUSH DPL 和 PUSH DPH，用在将来调用目标函数；(2)主调函数编译后生成的LCALL指令不再指向该函数原先的一个entry，而指向新增加的这个entry。

需要注意的是,所有这一切发生在编译时间，在程序实际运行时没有增加任何的判断语句。另外一点是：虽然有可能Bank中的一个函数在跳转表中有两个entry，但跨Bank执行时并不会引起混乱。因为从公共区调用该函数与从别的Bank调用该函数会进入跳转表中不同的entry。而且，很多情况下这种函数只会有一个entry，因为它们经常只被公共区的函数调用。

下面用8个Bank的例子具体分析一下改进的效果：

(1)附加指令的比较

· 未改进前，每次跨Bank函数调用需要18条额外指令。

· 改进后，当跨Bank函数调用从公共区发出时需要7条额外指令；跨Bank函数调用的其他情况仍然需要18条额外指令。

我们采用一个比较保守的估计，即从公共区发出的跨Bank函数调用只占所有跨Bank函数调用的50%，那么改进后的跨Bank函数调用需要的平均额外指令是：7*50%+18*50% = 12.5，大大减少了Bank切换的开销。

(2)堆栈占用情况的比较

对于8051系列微处理器来说，堆栈是非常重要的一个资源，不仅它的数量极其有限(经典的8051结构只有最多80个字节可供内部堆栈使用，假如被某些变量占用，实际数目还要减少)；而且所有的函数返回地址必须压入内部堆栈。

堆栈中最需要考虑的问题是在一个函数调用发生以后，尚未返回之前，有多少个字节被压入堆栈，因为这直接影响到函数调用的深度。改进前，若发生跨Bank函数调用，这个数值是4个字节；改进后，若跨Bank函数调用从公共区发出，这个数值将减少到2个字节。

这种改进方法不只适用于标准的32kB划分方法，对其他划分方式也适用。举个例子，如果公共区为48kB，那么此时的控制线是A15和A14，由于公共区的地址空间是0~48kB，A15和A14不可能同时为1，选中的还是EPROM的最低48kB。

以上所有的改进都在Keil仿真器上检验并通过。考虑到Keil数以万计的用户，这一改进的效果是难以估量的。

参考文献

1 Keil Elektronik GmbH and Keil Software Inc..GS51.PDF——Getting Started with μ Vision 2.2001-02:67-68

2 Keil Elektronik GmbH and Keil Software,Inc..A51.PDF——Macro Assembler and Utilities for 8051 and Variants.2000-07:279-280

3 Keil Elektronik GmbH and Keil Software Inc..L51_BANK.A51 Version 2.13,2000

☆☆

(上接第182页)

2.2 在HLR中完成HLR/VLR的错误恢复

(1)HLR从数据错误中恢复

当HLR从数据错误中恢复后，遍历VLR号码索引表，向所有VLR发送UNRELDIR消息，同时删除该表项管理的所有IMSI数据表中指向该VLR的位置指针。最后修改ptr_first_imsi=ptr_last_imsi=NULL。为提高用户重新登记时系统处理的效率，同样可以不删除以VLR号码为索引的双向链表中的各表项。

(2)HLR收到VLR发送的BULKDEREG消息

当HLR收到VLR_k发送的BULKDEREG消息后，从“发送方标识符”参数中提取VLR号码，以该VLR号码为索引，查找VLR索引表，找到该VLR号码表项对应的ptr_first_imsi,并删除以ptr_first_imsi为首地址的双向IMSI链表中的所有指向该VLR的位置指针，调整ptr_first_imsi=ptr_last_imsi=NULL，在此可以删除对应的VLR号码索引表项。

3 小结

从上面的分析可看出，在HLR/VLR数据库中建立VLR号码/HLR号码的索引结构，不仅很容易实现HLR和VLR的错误恢复，而且也实现了移动漫游用户在HLR/VLR数据库中的有序管理，例如，用户在规定时间(通常为24小时)内没有活动的去激活管理和用户鉴权参数的更新管理，大大提高了HLR/VLR对用户管理的效率。

此外，在HLR和VLR中分别把用户的位置信息和归属HLR信息以及对应的系统数据放于号码索引表表项中统一管理，在一定程度上节省了系统的存储开销。

参考文献

1 中华人民共和国信息产业部.800MHz CDMA数字蜂窝移动通信网移动应用部分技术要求.2000

2 TIA/EIA/IS-41:Cellular Radio Telecommunications Intersystem Operations.1997-12