ROM Software Patching on the Motorola DSP56304

by

Tom Zudock

Motorola, Incorporated
Semiconductor Products Sector
6501 William Cannon Drive West
Austin, TX 78735-8598

Order this document by: APR33/D

Motorola reserves the right to make changes without further notice to any products herein to improve reliability, function, or design. Motorola does not assume any liability arising out of the application or use of any product or circuit described herein; neither does it convey any license under its patent rights nor the rights of others. Motorola products are not designed, intended, or authorized for use as components in systems intended for surgical implant into the body, or other application in which the failure of the Motorola product could create a situation where personal injury or death may occur. Should Buyer purchase or use Motorola products for any such unintended or unauthorized application, Buyer shall indemnify and hold Motorola and its officers, employees, subsidiaries, affiliates, and distributors harmless against all claims, costs, damages, and expenses, and reasonable attorney fees arising out of, directly or indirectly, any claim of personal injury or death associated with such unintended or unauthorized use, even if such claim alleges that Motorola was negligent regarding the design or manufacture of the part. Motorola and Ⓜ are registered trademarks of Motorola, Inc. Motorola, Inc. is an Equal Opportunity/Affirmative Action Employer.

TABLE OF CONTENTS

LIST OF EXAMPLES

SECTION 1

INTRODUCTION

1.1 INTRODUCTION

Although software is typically very robust before its introduction to a device's ROM, modifications must often be incorporated to software in ROM. On the Motorola DSP56304, incorporation is achieved by enabling the patch mechanism and using the cache to replace, skip, or insert portions of code in ROM.

1.2 SCOPE

In this application report, both simple and complex cases of patching ROM code are discussed. Additionally, methods are presented that optimize the patching procedure to minimize the impact on cache performance. Using the information presented within, a software developer will be able to use the patching mechanism as is most appropriate for the application at hand.

The following discussion outlines the straightforward procedure of patching ROM on the DSP56304, including many potential challenges encountered in implementing patches. Approaches are also laid out to optimizing the many aspects of implementing patches.

1.3 CACHE BASICS AND RELEVANT INSTRUCTIONS

The cache on the DSP56304 is located in internal Program RAM. It must be enabled by setting Bit 19, the CE bit, in the Status Register (SR). When disabled, the cache memory may be used as internal Program RAM. However, upon enabling the cache, any instructions that were resident are no longer available. The cache is subdivided into sectors, each of which is 128 program words in size. Hence, eight sectors are available for the 1 kword cache.

Although a handful of instructions are associated with cache operation, patching requires knowledge of only two instructions: PLOCK and PUNLOCK. For simple patches that are initiated at start-up and left active for the duration of program execution, only the PLOCK instruction will be used.

For patches that are dynamically engaged and disengaged, PUNLOCK will be used as well. The cache must be enabled for these cache instructions to be valid. If disabled, execution of any cache instructions will generate an illegal instruction interrupt.

PLOCK is used to lock a cache sector to a specified program memory address. More specifically, the seventeen Most Significant Bits (MSBs) of the argument specified with the PLOCK instruction are moved into the least recently used cache sector tag, and the sector is then locked. Locking the cache sector prevents the cache controller from modifying the cache sector tag. In other words, locking ensures that a cache sector is always assigned to a specified range of P memory words. This is extremely relevant when patching ROM code.

The PUNLOCK instruction is the inverse of the PLOCK instruction. It unlocks the specified cache sector. This returns control of that cache sector to the cache controller for allocation. Using PUNLOCK in ROM patching is necessary only if a cache sector is intended for reuse for multiple patches that do not fall within the addressing range covered by one cache sector. If all software resides in ROM and all patches fit within the available cache sectors, PUNLOCK will not be needed. However, if more than eight patches are required or if the user needs the improved performance the cache provides for external Digital Signal Processor (DSP) code, patches can be engaged and disengaged dynamically.

SECTION 2

PATCHING

2.1 OVERVIEW

Initiating a patch requires four steps.

1. Enable the cache by setting Bit 19, the CE bit, in the Status Register (SR).
2. Enable the patching mechanism by setting Bit 23, the PEP bit, in the Operation Mode Register (OMR).
3. Lock the cache sector to the ROM address to be patched (PLOCK).
4. Move the new opcode over the original opcode in ROM.

Enabling the cache and patch mechanism are obvious steps, as it is this functionality that makes ROM patching possible. Locking the cache sector to the ROM address to be patched sets one cache sector's tag field to the seventeen MSBs of the ROM address. Moving the new opcode over the original opcode in ROM causes the new opcode to be stored into the locked cache sector. It also sets the valid bit for that location in the locked cache sector. Thereafter, whenever the core generates an address for the instruction fetch of the old ROM opcode, the cache controller will view this as a cache "hit" and will execute the instruction out of the cache instead of ROM. Because the desired new opcode is in the cache, the new opcode will be executed and the patch is complete.

2.2 SINGLE INSTRUCTION REPLACEMENT

The overview lists the steps one should take to install a ROM patch. Now, to focus on a specific code example, consider the ROM-resident filter shown in **Example 2-1**.

Example 2-1 ROM Filter Error

```
     org     p:$ff00d0                     ; start of ROM on 56304
     clr     a       x:(r0)+,x0 y:(r4)+,y0 ; clr accum, get data/coeff
     rep     #10                           ;
     mac     x0,y0,a x:(r0)+,x0 y:(r4)+,y0 ; filter, get data/coeff
     macr    x0,y1,a                       ; ERROR!!! y1 should actually be y0
```

The last line in the example of **Example 2-1** contains an error. Unlike the preceding line, one of the source operands is y1 as opposed to y0. The code in **Example 2-2** should be used to patch this piece of code.

Example 2-2 ROM Filter Patch

```
org    p:$400               ; start of ext P RAM
bset   #19,sr               ; enable the cache
bset   #23,omr              ; enable the rom patch mechanism
move   #$2000D3,x0          ; x0=ROM opcode to "macr x0,y0,a"
plock  $ff00d3              ; lock cache sectr with ROM patch address
movem  x0,p:$ff00d3         ; update ROM opcode to "macr x0,y0,a"
jmp    $ff00d0              ; jmp to start of ROM
```

Once the patch initialization instructions above are executed, the patch is in place indefinitely. The instructions parallel the steps outlined in **Section 2-1**. Executing a PUNLOCK on the ROM patch address will disable this patch. One detail requires attention in the above example. The new opcode put in place required only one program word. If the new instruction were to require two program words, two program words must be overwritten. However, the user must ensure in such a case that both words fall within the address range covered by the locked cache sector.

Recall that the PLOCK instruction moves the seventeen MSBs into the tag field of the least recently used cache sector and locks it. Hence, the cache sector will cover the program memory range from P:$FF0100 to P:$FF017F if the two-word instruction is at address P:$FF017F . Because the second word of the two instructions resides at P:$FF00180, overwriting that location in ROM will not introduce the complete opcode into the locked cache sector. In fact, the second word will not be moved into any cache sector at all. This is because the patch mechanism requires the destination of a patch opcode to be a locked cache sector. Two cache sectors must therefore be locked to implement a two word patch that crosses a cache sector's boundary. An alternative is to jump out before the boundary is reached (using a patched-in JMP instruction), execute the desired instructions in external RAM, and jump back into the code. This option will reduce the number of cache sectors consumed to one but is somewhat less efficient.

2.3 SKIPPING MULTIPLE INSTRUCTIONS

Skipping multiple consecutive instructions in ROM follows directly. The straightforward solution is to replace the first instruction in ROM to be skipped with a JMP or BRA instruction to the ROM address where execution should resume. The initialization procedure is the same as that described in the previous section. As long as the JMP or BRA is short enough to fit into one program word, there is no need to be concerned about the second word of the instruction lying outside the range covered by the locked cache sector. However, if the JMP or BRA is long and the resulting two-word instruction exceeds the range covered by one cache sector, a second sector must be locked. One solution is to exit ROM at an instruction before the cache sector boundary is reached, using a patched-in JMP or BRA instruction in the last location in the cache sector, jumping or branching to a RAM location, then executing a JMP or BRA back into the code at the desired reentry point. This method will reduce the number of cache sectors used to one but is less efficient.

Let's examine a specific example. Consider the following code in ROM:

Example 2-3 Multiple Instruction ROM Code

```
org    p:$ff00d0                     ; start of ROM on 56304
clr    a       x:(r0)+,x0 y:(r4)+,y0 ; clr accum, get data/coeff
rep    #10                           ;
mac    x0,y0,a x:(r0)+,x0 y:(r4)+,y0 ; filter, get data/coeff
macr   x0,y0,a                       ; round result
clr    b       x:(r0)+,x0 y:(r4)+,y0 ; clr accum, get data/coeff
rep    #10                           ;
mac    x0,y0,b x:(r0)+,x0 y:(r4)+,y0 ; filter, get data/coeff
macr   x0,y0,b                       ; round result
```

Now suppose it has been decided that the second filter that uses accumulator B is no longer needed. Using the technique just discussed, the user can install a BRA instruction in place of the CLR B instruction. The target of the BRA is just after the MACR instruction. The following code exemplifies this.

Example 2-4 ROM Code Alteration

```
org     p:$400                       ; start of ext P RAM
bset    #19,sr                       ; enable the cache
bset    #23,omr                      ; enable the ROM patch mechanism
move    #$050c04,x0                  ; x0=BRA opcode
PLOCK   $ff00d0                      ; lock cache sectr with ROM patch addr
movem   x0,p:$ff00d4                 ; overwrite ROM with opcode
jmp     $ff00d0                      ; jmp to start of ROM code
```

2.4 INSERTING INSTRUCTIONS

In some instances, the ROM code instructions do not need to be modified or replaced. Rather, one or more instructions must be inserted at a particular location. To meet this requirement, the patch begins with a JMP, BRA, JSR, BSR, etc., out of the ROM code at the instruction where execution should resume. The target is a location in RAM that contains the instructions to be inserted. After executing the inserted instructions, the original ROM instruction that was patched over with a JMP, BRA, JSR, or BSR instruction must be executed. Finally, a JMP, BRA, or RTS is used to return flow into the ROM after the code patch has been executed.

When inserting instructions, exiting the ROM will most likely require a two-word instruction because a long JMP, BRA, JSR, or BSR will be used. This, of course, is dependent upon where your RAM is mapped. In typical configurations, two program words will be needed. As mentioned earlier, caution should be observed to ensure that both program words reside within the address range covered by the locked cache sector.

Consider another specific example. First, assume we start with the ROM code listed below.

Example 2-5 ROM Code

```
org     p:$ff00d0                     ; start of ROM on 56304
clr     a       x:(r0)+,x0 y:(r4)+,y0 ; clr accum, get data/coeff
rep     #10                           ;
mac     x0,y0,a x:(r0)+,x0 y:(r4)+,y0 ; filter, get data/coeff
macr    x0,y0,a                       ; round result
;NEED TO INSERT CODE HERE             ;
add     a,b                           ; sum result of two filters
```

The functionality to be inserted is shown below. The code contains calculations needed to provide a filter output into accumulator “b” before summing with accumulator “a” above.

Example 2-6 ROM Code Alteration

```
clr     b       x:(r0)+,x0 y:(r4)+,y0 ; clr accum, get data/coeff
rep     #10                           ;
mac     x0,y0,b x:(r0)+,x0 y:(r4)+,y0 ; filter, get data/coeff
macr    x0,y0,b                       ; round result
```

To achieve this task, the ROM instructions “`macr x0,y0,a`” and “`add a,b`” will be replaced with a two word JSR to a subroutine patch. The MACR instruction that was replaced will execute at the entry to the patch, followed by the code to be inserted, followed by the replaced ADD instruction, and followed finally by an RTS that will return flow back into the ROM code.

The code below enables the cache and patch mechanism, initializes the JSR to exit the ROM code, and includes the patch code itself.

Example 2-7 Patch Code

```
     org    p:$400                        ; start of ext P RAM
     bset   #19,sr                        ; enable the cache
     bset   #23,omr                       ; enable the ROM patch mechanism
     move   #$0bf080,x0                   ; x0=first word of two word JSR opcode
     PLOCK  $ff00d0                         ; lock cache sector with ROM patch addr
     move   x0,p:$ff00d3                  ; update ROM with JSR opcode
     move   #$00040e,x0                     ; x0=second word of two work JSR opcode
     move   x0,p:$ff00d4                  ; update ROM with JSR destinatin addr
     jmp    $ff00d0                       ; jmp to start of ROM code
     ;START OF PATCH CODE                 ;
     macr   x0,y0,a                       ; round result
     clr    b       x:(r0)+,x0 y:(r4)+,y0 ; clr accum, get data/coeff
     rep    #10                           ;
     mac    x0,y0,b x:(r0)+,x0 y:(r4)+,y0 ; filter, get data/coeff
     macr   x0,y0,b                       ; round result
     add    a,b                           ; sum result of two filters
     rts                                  ; reenter ROM code
```

SECTION 3

OPTIMIZATION

3.1 INTRODUCTION

To yield a functioning application, it is essential in many circumstances to use the cache to modify code that exists in ROM. However, if that application makes use of external memory, the performance impact that results from reducing the number of available cache sectors must be considered. It may be worthwhile from an application development standpoint to simulate software with one or more cache sectors locked out of a usable range to see the effect on performance. It is of little benefit to find out after the fact that the number of sectors that must be locked to patch ROM code reduces performance to the point of software failure.

3.2 MINIMIZING PATCHING IMPACT ON THE CACHE

In the event that the number of cache sectors dedicated to patching must be minimized or the number of patches required exceeds the number of cache sectors available, it becomes necessary to implement more than one patch per cache sector. Basically, this can be achieved in two ways. First, if two areas that need to be patched reside within the address range covered by one cache sector, two patches can easily be incorporated into one sector. Second, it is possible to daisy-chain multiple patches in a fashion that uses one cache sector.

3.3 MULTIPLE PATCHES COVERED BY ONE CACHE SECTOR ADDRESS RANGE

This case is simple in concept. If two sections of code reside within the address range covered by one cache sector, then both patches can easily be initialized to reside in one cache sector. The range covered by a locked cache sector was discussed earlier, but the topic will be revisited for clarification.

The PLOCK instruction initializes a cache sector’s tag to the seventeen MSBs of the argument specified. This is essentially the base address for that sector. The next 128 program word addresses are covered by that cache sector. In all preceding examples, the address of the start of the patch was used as the argument for PLOCK. Since the seventeen MSBs of that argument are used by PLOCK, it is important to realize that the range covered is not necessarily the address specified in the argument plus 128 program words (unless, of course, the start of the patch address happens to have the seven Least Significant Bits (LSBs) as 0). Rather, the base address for the cache sector, which is zero, is taken from the seventeen MSBs, extended, and used as the base address for that sector. The ROM address range covered by a cache is extended from that value.

Once it is determined that any two sections of code that need to be patched meet this criteria, putting two patches in one sector is trivial. Simply lock the sector using either address (both locations to be patched share the same seventeen MSBs) and install the appropriate type of patch, many of which have already been presented.

3.4 DAISY-CHAINING PATCHES TOGETHER

Daisy-chaining patches is a method whereby one cache sector is used to support multiple patches that do not reside within a range that could be covered by one cache sector. One incentive for using this approach is the need to maximize the effectiveness of the cache by not consuming too many cache sectors for patching. A second demand for this approach stems from software that requires more patches than there are available cache sectors (of which there are eight total). It is important to realize, however, that such a method comes at a performance cost for those patches that are daisy-chained. More than likely, a mixture of daisy-chained patches and patches that use dedicated cache sectors will provide the balance between cache and patch performance best suited for a given application.

Chaining patches together requires dynamically enabling and disabling patches. At the onset of application execution, the first patch is initialized using a cache sector. The patch code that is executed must perform the purpose of the patch (i.e., correct a coding error), disable the current patch, and enable the next patch. This procedure continues through a series of patches chained together eventually reenabling the first patch and fulfilling the cycle. This behavior continues for the life of the application.

To consider a specific example, one should begin with some code that is already in ROM, as shown in **Example 3-1**.

Example 3-1 ROM Code

```
        page    255,60
        org     p:$ff00d0       ;start of rom on 56304
rom_main                        ;call all rom subroutines
        jsr     sub1_rom
        jsr     sub2_rom
        jsr     sub3_rom
        jmp     rom_main

        org     p:$ff1000
sub1_rom
        move    #-1,m1          ;m1=linear
        move    (r1)+           ;increment r1,n1 for testing
        move    r1,n1
        rts

        org     p:$ff2000
sub2_rom
        move    #-1,m2          ;m2=linear
        move    (r2)+           ;increment r2,n2 for testing
        move    r2,n2
        rts

        org     p:$ff3000
sub3_rom
        move    #-1,m3          ;m3=linear
        move    (r3)+           ;increment r3,n3 for testing
        move    r3,n3
        rts
```

The ROM software shown in **Example 3-1** is simply an executive that calls three subroutines, each of which increments an address register. Notice that they are spread out in memory in such a way that one cache sector would not be able to cover them all. The goal is to use one cache sector to add a patch to each of the subroutines. The software listed in **Example 3-2** provides that functionality.

Example 3-2 Patch Code Lines 1–44

```
1:          page    255,60
2:  ;****************************************************************************
3:  ;       equates
4:  ;****************************************************************************
5:  sub1_rom_patch_addr     equu    $FF1002     ;equate for first patch address
6:  sub2_rom_patch_addr     equu    $FF2002     ;equate for second patch address
7:  sub3_rom_patch_addr     equu    $FF3002     ;equate for third patch address
8:  rom_main                equu    $FF00D0     ;equate for entry to rom_main
9:
10: ;****************************************************************************
11: ;       data for daisy chain patching
12: ;****************************************************************************
13:         org     x:$0
14: patch_data_table                           ;data struct for daisy chain patching
15:                 dc      sub1_patch         ;address of patch routine
16:                 dc      sub1_rom_patch_addr;first address to patch
17:                 dc      $205900            ;first rom word replaced by patch jsr
18:                 dc      $223900            ;second rom word replaced by patch jsr
19:                 dc      sub2_patch         ;etc...
20:                 dc      sub2_rom_patch_addr
21:                 dc      $205A00
22:                 dc      $225A00
23:                 dc      sub3_patch
24:                 dc      sub3_rom_patch_addr
25:                 dc      $205B00
26:                 dc      $227B00
27: patch_data_table_end
28:                 dc      0                  ;FREE, here for patch_data_table_end
29: patch_ptr       dc      patch_data_table+1 ;ptr into patch_table data struct
30: patch_r1        ds      1                  ;register save memory locations
31: patch_m0        ds      1
32: patch_m2        ds      1
33: patch_r2        ds      1
34: patch_r0        ds      1
35:
36: ;****************************************************************************
37: ;        main program
38: ;****************************************************************************
39:         org     p:$400
40:         bset    #19,sr                     ;enable cache
41:         bset    #23,omr                    ;enable patch mode for 56304
42:         nop
43:         pflush                             ;clear and unlock all cache sectors
44:                                            ;enable first patch
```

Example 3-2 (Cont.) Patch Code Lines 45–88

```
45:          plock   sub1_rom_patch_addr       ;lock cache sector for first patch
46:          move    #$0bf080,x0               ;x0=opcode for jsr
47:          movem   x0,p:sub1_rom_patch_addr  ;install jsr in cache (write over rom)
48:          move    #sub1_patch,x0            ;x0=address of patch routine
49:          move    x0,p:sub1_rom_patch_addr+1;install addr in cache-write over rom
50:          jmp     rom_main                  ;enter rom_code executive
51:
52:  ;*******************************************************************************
53:  ;        patch routines
54:  ;*******************************************************************************
55:  sub1_patch
56:          move    #-1,m5                    ;patch code added to sub1_rom
57:          move    (r5)+
58:          jmp     daisy_chain_patch         ;setup next patch
59:
60:  sub2_patch
61:          move    #-1,m6                    ;patch code added to sub2_rom
62:          move    (r6)+
63:          jmp     daisy_chain_patch         ;setup next patch
64:
65:  sub3_patch
66:          move    #-1,m7                    ;patch code added to sub3_rom
67:          move    (r7)+
68:          jmp     daisy_chain_patch         ;setup next patch
69:
70:  ;*******************************************************************************
71:  ;        patch daisy chain routine
72:  ;*******************************************************************************
73:  daisy_chain_patch
74:          move    r0,x:patch_r0             ;save r0
75:          move    m0,x:patch_m0             ;save m0
76:          move    x:patch_ptr,r0            ;r0=ptr in patch_data_table
77:                                            ;m0=mod patch_data_table
78:          move    #patch_data_table_end-patch_data_table-1,m0
79:          move    m2,x:patch_m2             ;save m2
80:          move    r1,x:patch_r1             ;save r1
81:          move    r2,x:patch_r2             ;save r2
82:          move    x:(r0)+,r1                ;r1=current rom adress patched
83:          move    x:(r0)+,r2                ;r2=rom word replaced by patch jsr
84:          move    #-1,m2                    ;m2=linear addressing
85:          punlock (r1)                      ;disable current patch, unlock sector
86:          move    x:(r0)+,r1                ;r1=rom word replaced by patch jsr addr
87:          movem   r2,p:orig_instr           ;install first word of instr
88:          movem   r1,p:orig_instr+1         ;install second word of instr
```

Example 3-2 (Cont.) Patch Code Lines 89–104

```
89:          move    x:(r0)+,r1           ;r1=next patch routine address
90:          move    x:(r0),r2            ;r2=next rom address to patch
91:          move    r0,x:patch_ptr       ;update patch_ptr
92:          move    #$0bf080,r0          ;r1=jsr opcode
93:          plock   (r2)+                ;lock sector using next patch address
94:          movem   r1,p:(r2)-           ;install jsr patch addr (overwrite rom)
95:          movem   r0,p:(r2)            ;install jsr (overwrite rom)
96:          move    x:patch_m0,m0        ;restore m0
97:          move    x:patch_m2,m2        ;restore m2
98:          move    x:patch_r1,r1        ;restore r1
99:          move    x:patch_r2,r2        ;restore r2
100:         move    x:patch_r0,r0        ;restore r0
101: orig_instr
102:         nop                          ;space for original instruction
103:         nop                          ;space for original instruction
104:         rts                          ;return back into rom code
```

The next paragraphs discuss the patch code shown in **Example 3-2**. Much of the code is interleaved to maximize the efficiency of the DSP56304 processor, minimizing pipeline stalls but making it more confusing to read the code. Therefore, line numbers are used to clarify references to the code.

- The code begins with a list of equates in lines 5–8. The variables that are defined indicate the address where patches are to be inserted in ROM and the start of the main program in ROM.
- Following in lines 14–26 is a table (patch_data_table) containing all information required by the daisy-chaining routine to perform its purpose. The table contains four elements for each of the three patches that are implemented: the start address of the patch routine, the address where the ROM code is exited, and two instruction opcodes that are located where the ROM code is exited. The patch daisy-chain code increments through this table using the values to dynamically enable and disable patches. Additionally, storage for a pointer into the table and for saving and restoring registers used by the daisy_chain_patch routine is allocated.
- Execution of the patch start-up code begins with the main program on line 40. The first four lines enable the cache, enable the patching mechanism, and flush the cache contents.
- Next, the first patch is enabled by locking a cache sector to the appropriate ROM address and overwriting that address with a JSR to the patch routine. Initialization is finalized by jumping to the start of the main ROM routine. The ROM code is listed in **Example 3-1** on page 3-5.

- The main ROM executive is a collection of subroutine calls. The first patch affects the ROM subroutine sub1_rom. When the JSR that has been introduced into that subroutine is encountered, program flow will be directed to the sub1_patch routine on line 55. This routine executes the patch and then jumps to the daisy_chain_patch routine.
- At the start and end of the daisy_chain_patch routine, all used registers are saved and restored. This ensures the code is transparent in operation to the ROM code. The R0 register is initialized with the pointer into patch_data_table, which is pointing to the second element in the table (the current ROM address being patched). This value is loaded into R1 on line 82 and used to unlock the cache sector on line 85.
- Since the pointer into patch_data_table has been advanced, it is now pointing to the two opcodes that were replaced in ROM by the JSR instruction used to exit the ROM code. These two opcodes are retrieved from patch_data_table and written into Program RAM at the end of the patch routine in lines 83, 86 and 87–88 (they replace the NOPs). Hence, the daisy_chain_patch routine will execute them just before exiting. At this point, the current patch has been disabled and the instructions lost exiting the ROM code have been restored for execution. The code now turns its focus to setting up the next patch.
- The next two elements in patch_data_table are the address of the next patch routine and the address to exit from ROM. These values are retrieved in lines 89–90 and the pointer into the patch_data_table is saved in memory since it is no longer needed. The ROM address is used to lock the cache sector, which prepares it for the installation of the next patch.
- The pointer into ROM is then used as the destination for a JSR opcode and the starting address of the next patch routine on lines 94–95. Since a sector has already been locked for this region of ROM, the action of writing these opcodes into ROM installs the opcodes in the cache and enables the next patch.
- At this point, the next patch has been installed and enabled. After restoring all registers used, the daisy_chain_patch routine executes the two instructions that were replaced by the JSR in ROM and executes an RTS, which returns execution to the point in ROM just past the exit point.
- Eventually, the next ROM patch address will be encountered and the daisy_chain_patch routine will repeat.

The code of **Example 3-2** on page 3-6 inserted instructions into various locations into ROM. However, skipping instructions is also easy to implement. The programmer simply uses a JMP or BRA to exit the ROM code with the destination being the daisy_chain_patch routine. In the data structure where the “lost” opcodes for that patch are stored, place opcodes for a JMP or BRA back into the desired reentry point into the ROM. This prevents the RTS at the end of the daisy_chain_patch routine from being executed and allows choice of any destination for ROM reentry.

Many other patching variations depend upon the needs of a given situation and can be supported by some careful thought and modification to the code of **Example 3-2**.