ROMifying a Program for Uploading to ROMX

Converting a program to run under ROMX (or ROMifying) can be a fairly simple task. There are several different steps you need to take and they depend upon how the program is written. For this tutorial we will only concern ourselves with machine language programs (those you can BRUN from a disk or otherwise create as direct 6502 code that resides in a specific location in RAM). This document pertains only to the original ROMX for Apple II/II+; if you have a ROMXc, ROMXe, ROMXc+, or ROMX+ please see the document "*ROMifying a Program for Uploading to ROMXce+*" instead.

PART I – BASIC PROCESS

To work in ROMX, the code must fit into the 12K of space from $D000-$FFFF. Larger programs can be handled by splitting across multiple banks but we will not go into that here. If your program uses any F8 monitor ROM routines, then you need to either duplicate them inside your program or include some or all of the $F800-$FFFB space in your image (the RESET vector at $FFFC/D will definitely be changed).

There are two basic ways to ROMify code. The first involves rewriting the code so that it can run from ROM. This would include:

1. Translating and relocating all code that uses absolute addressing.
2. Separating the code and variable storage and finding a suitable location in RAM for the variables.
3. Removing any self-modifying code or references to any standard ROM addresses
4. Modifying the exit process when the program is terminated.

While this may seem a bit daunting, there is a second method that is far simpler and fortunately will work for the majority of programs. It works by relocating the entire program to ROM space - AS IS - with little if any modification. A small loader program is then added that moves the code back into its original RAM location at launch and then jumps to the starting address of the program. The code then runs just as if it had been loaded from disk (except that the standard ROM - and possibly DOS - might not be there).

For the rest of this tutorial we will describe this second method. We will also make the following assumption: **that the code is completely loaded into memory at one time and takes up less than $2700 bytes (roughly 10K or 40 sectors on disk)**. This will allow us to keep the entire F8 ROM code intact. This can also help with running and debugging your code since you'll have the full power of the monitor routines to inspect, break, and test your code. Another advantage of this approach is that you can even test your code in a Language Card or 16K RAM Card (LC) before committing to ROM. More on that later.

The steps to ROMify such code are then quite simple. We just need to combine the code plus the F8 ROM and the loader program into a single file. The F8 image is modified so that the reset vector points to our loader code. And the loader code is configured with the starting and ending addresses of the ROM data as well as the destination address in RAM. Finally, we tell the loader where to go after the code has been relocated into RAM.

The Image Tutorial.dsk file that accompanies this document has several files for converting a game to run under ROMX:

```
DISK VOLUME 254

 A 006 HELLO                   Standard Apple Language Card Loader
*B 050 INTBASIC                Integer Basic to Load into Language Card
*B 034 BREAKOUT                Original Breakout Game as downloaded from the INTERNET ARCHIVE
 B 050 BREAKOUT.IMG            ROMified image of Breakout
*B 026 CHIPOUT                 Original Chipout Game as downloaded from the INTERNET ARCHIVE
 B 050 CHIPOUT.IMG             ROMified image of Chipout
*B 050 IMAGE TEMPLATE 12K      Template Image for building 12K programs (no F8 Monitor ROM)
*B 050 IMAGE TEMPLATE WITH F8  Template Image for building 10K programs (with F8 Monitor ROM)

INTERNET ARCHIVE URL:
https://archive.org/search.php?query=Breakout_19xx____Breakout_19xx___Hires__Chipout_19xx_Twilight_Software
```

We will start with the IMAGE TEMPLATE WITH F8 file, which has the modified F8 code, loader program, and a blank area for the remainder of the 12K image file. You can BLOAD this file, move your program into the blank area, modify the loader parameters, and then BSAVE it back to disk. Or use the Template source code to add into your own Assembly programs.

Once you have a complete 12K image you can either Move it from RAM or BLOAD into a 16K LC for testing. Or you can Upload directly into a ROMX bank. We will now show some examples of how this is done. Since we will be testing on a LC first, you can even try this out using an Emulator such as Virtual II for the Mac. By the way, this is exactly how the ROMX firmware was developed. If you are working on real hardware and do not have a 16K RAM card, you can skip those sections and Upload directly into ROMX.

EXAMPLE #1

Let's say that we want to Upload into ROMX a Breakout program similar to the original one that came on cassette with the early Apple II machines. Several such programs can be found online and one is included here on the Image Tutorial.dsk as BREAKOUT. We can use a utility such as Copy][Plus to determine the load location and length of BREAKOUT; this turns out to be $800 and $2000 respectively. Since the total size of our program is only 8KB, we can also include the F8 ROM.

So we can now create a ROX image in three easy steps:

```
   1. BLOAD IMAGE TEMPLATE WITH F8          (Loads at $1000-$3FFF)
   2. BLOAD BREAKOUT, A$1000                (Loads into bottom of image)
   3. BSAVE BREAKOUT.IMG, A$1000, L$3000
```

Since our template defaults to a starting address of $800, we don't need to make any modifications to the Loader. This image could now be loaded into ROMX using the P)ick command or M)anually (after BLOADing at $3000). But let's test it out first in the LC (make sure INTEGER or Applesoft has already been loaded there):

```
CALL -151               (Go into Monitor)
C083 C083               (Enable writing to Language Card)
D000<1000.3FFFM         (Move our image into Language Card)
F700G                   (Execute our Launch code)
```

If all went well, the Launch code would have moved the Breakout program into RAM at $800 and it would have started running.

EXAMPLE #2

Let's see if we can do the same thing with the CHIPOUT program also found on the tutorial disk. This program loads at $07FD and is $1803 bytes long. So it will certainly fit into our ROM. But since the start address is not $800, we will have to make a couple of changes. After loading the Image Template, we need to BLOAD the program so that the start address is in the first page of the image. The Loader routine only copies full pages of memory (256 bytes) at a time (just think about the first two digits of a hex address). So we will end up loading $0700-$07FC or 253 bytes that we will copy back but not really use. In this case, these bytes are in the screen RAM area so when the Loader restores them, we are likely to see some random characters written to the screen just before the program launches. No harm, but watch for it when you launch the image either from the LC or ROM. So here are the steps to add the program to our template:

```
1. BLOAD IMAGE TEMPLATE WITH F8    (Loads at $1000-$3FFF)
2. BLOAD CHIPOUT, A$10FD           (Loads into bottom of image w/offset)
```

Before we save the image, we need to modify the Launch routine:

```
CALL -151               (Go into Monitor)
3706:07                 (Change TARGET hi-byte to $07)
370B:FD 07              (Change JMP address to $07FD)
```

Now we can save the image to disk:

```
BSAVE CHIPOUT.ROM, A$1000, L$3000
```

And to test out on the LC:

```
C083 C083                     (Enable writing to Language Card)
BLOAD CHIPOUT.ROM, A$D000     (Move our image into Language Card)
F700G                         (Execute our Launch code)
```

ADDITIONAL NOTES

When first trying to get a program to run from ROM, it is often helpful to set the RESET vector to $FF59 (the Monitor Entry point, assuming you are keeping the F8 ROM). When the image is loaded using ROMX, it will immediately enter the Monitor (* prompt). Then you can examine the ROM contents, execute the loader routine, and make changes or set breakpoints to the code in RAM.

Also note that the ROM images you create this way can be directly used with most emulators. First you need to extract the image from the Apple environment into a file on the host computer. CiderPress and AppleCommander are two great tools for doing this. Then you tell the Emulator to use this file as its boot ROM. For example in Virtual II on the Mac, you launch a new machine, click the Setup button, and select ROM memory under the Components section. You can then pick "Use specific ROM" and select your image file (make sure it ends with ".ROM"). When you hit Restart the virtual machine will start up with your image. With Applewin, you can use the -rom option on the command line when you launch the program.

Here's another trick that might be helpful if your program just needs a little more space. If you examine the F8 monitor code, you will find that the first $70 bytes contain routines used to plot graphics. If your program does not use these, then the loader code could be moved there leaving the entire $D000-$F800 space available for your program. The IMAGE TEMPLATE 12K file goes even further giving you almost 12K of program space ($D000-$FEFF) at the expense of the entire Monitor ROM. So make sure your program does not attempt to make any calls there.

Finally, if you make a really cool image that you would like to share with others, please contribute to theRomExchange.com. Before sending however, please add some rmx metadata to describe your image:

```
1. BLOAD <your great image> ,A$3000         (Loads at $3000-$5FFF)
2. RUN INFO GEN 3                           (from the ROMX Utilities disk)
3. Enter Description and Additional Info    (Adds metadata to image)
4. BSAVE <image name>.RMX, A$3000, L$3100   (Saves complete rmx image)
```

TEMPLATE LISTINGS

IMAGE TEMPLATE WITH F8

```
                1       ;   ROMX IMAGE TEMPLATE CODE
                2       ;   FILE: IMAGE TEMPLATE WITH F8
                3
                4   RAM_LOC  EQU   $0800
                5   INIT     EQU   $FB2F
                6
                7            ORG   $1000           ;Start of Image
                8
1000: 00 00 00  9            DS    $3700-*         ;Room for our program
                10
3700: 20 28 F7  11  Launch   JSR   Setup+$C000
3703: A2 D0     12           LDX   #/$D000         ;SOURCE hi-byte
3705: A9 08     13           LDA   #/RAM_LOC       ;TARGET hi-byte
3707: 20 0D F7  14           JSR   CopyLp+$C000
                15
370A: 4C 00 08  16           JMP   RAM_LOC         ;Program entry point
                17
370D: 85 03     18  CopyLp   STA   $03
370F: A9 00     19           LDA   #$00
3711: 85 00     20           STA   $00
3713: 85 02     21           STA   $02
3715: 86 01     22  CpyLp2   STX   $01
3717: A0 00     23           LDY   #$00
                24
3719: B1 00     25  :2       LDA   ($00),Y         ;do 256 bytes
371B: 91 02     26           STA   ($02),Y
371D: C8        27           INY
371E: D0 F9     28           BNE   :2
3720: E6 03     29           INC   $03
3722: E8        30           INX
3723: E0 FF     31  SrcEnd   CPX   #/$FF00         ;END OF SOURCE  hi-byte
3725: D0 EE     32           BNE   CpyLp2          ;do xx pages
3727: 60        33           RTS
                34
3728: D8        35  Setup    CLD                   ;Do what we need to init computer
3729: 20 2F FB  36           JSR   INIT            ;Set up screen softswitches
372C: 60        37           RTS
                38
372D: 00 00 00  39           DS    $3FFC-*         ;F8 ROM moved into $3800-$3FFB
                40
3FFC: 00 F7     41           DA    Launch+$C000    ;RESET vector ($FF00)
3FFE: 40 FA     42           DA    $FA40           ;IRQ vector (not used)
                43
```

IMAGE TEMPLATE 12K

Change line 9 to:

```
1000: 00 00 00  9            DS    $3F00-*         ;Room for our program
```

PART II – ADDITIONAL OPTIONS

If your program has a Quit option or exits after performing its task, you may wish to return to the ROMX menu when the program is finished. From the ROMX API Guide, we know that we can enable the firmware with the following sequence:

```
  BIT $CACA
  BIT $CACA
  BIT $CAFE
```

At this point, with the ROMX firmware active we can either boot another image by executing:

```
  BIT $CFEn     (n=bank to launch)
  JMP (FFFC)    (Jump to RESET vector location in boot image)
```

Or we can re-launch the ROMX menu with:

```
  JMP $DFD0     (Jump to ROMX Menu loader)
```

Launching the ROMX menu this way will simulate a power-on condition, with the normal countdown to activate the default bank. We'll show you later how to manually initialize the firmware and drop into the menu without using the countdown feature.

DEALING WITH LARGER PROGRAMS

When your program is larger than 12K, there are several techniques that can be tried to reduce the size. Obviously, if this is your own program you will already know what areas and possible shortcuts you can use to reduce its memory footprint. If you are trying to squeeze someone else's code into ROM then here are some things to look for.

Scan the code for cleartext ASCII strings, probably with high-bit set per Apple standards. Many of these will be warning or informational messages that might be ripe for shortening. For example, if the program displays a full screen intro page with credits, copyright info, etc. you might be able to remove this completely or at least change the strings to just a few characters.

The Apple II Memory Test V1.4RX program in the Image section of the theRomExchange.com was originally over 14K (14688 to be exact). And it used several routines in the F8 ROM. The F8 code plus the Loader routine added another 300 bytes, which made it about 2700 bytes too large. But the program had a very extensive help system that provided lengthy descriptions of every command. By removing or shortening these strings, I was able to easily save this amount and more. Quite a significant reduction to make it fit into our available space!

When looking to shorten or eliminate strings, you will need to examine where and how they are used. By looking at the starting address of a string you can usually find where in the program that string is being referenced. Hopefully it is used in only one place, but always check to see if that's the case. If the particular message is not essential, you can often just remove it along with the code that uses it. If you want to keep a shortened message instead, you will have to examine the code and/or the string to determine how the first and last characters are specified. Common methods are to add a length byte to the beginning of the string, use a zero (or other value) byte to terminate the string, or use the MSB of the last character as a flag. For example the string “HELLO” might be found in memory as:

```
   05 C8 C5 CC CC CF     (Leading length byte string)
   C8 C5 CC CC CF 00     (Zero terminated string)
   C8 C5 CC CC 4F        (MSB terminated string)
```

Once you discover the technique being used, it will most likely remain consistent throughout the program. Strings can be used extensively within a program and thus are a good candidate for code reduction. They can also point to areas of code that might be removed completely if not critical to the application. If you don’t need cassette read/write, game paddle controls, disk functions, etc. then finding where in the application such routines reside will make it much easier to eliminate.

If you do end up removing code, beware that you might be altering relative branching offsets or other absolute addresses. You can check the code around the deletion to see if that’s the case and manually make changes. But if you need to make large-scale changes then it will be easier to run the code through a disassembler and create source code that you can later use in your favorite assembler program. This will take care of many of the addressing issues even if you never fully discover and annotate all of the source.

When your code is down to just needing a few more small reductions you can look for JSR instructions followed immediately by an RTS. These four bytes can usually be replaced by a single JMP instruction to the same address, saving one byte (as long as the RTS in not used elsewhere in the code). You may be surprised how many bytes you can save this way! And when you fully understand the code, it is often possible to replace a JMP with a Branch instruction when a certain condition will always be met. One more byte saved. Early Apple programmers would have loved the new opcodes found in the 65C02 such as BRA (Branch Always) and more extensive Stack operations, but since these were not available at the time there is plenty of inefficient code that was needed to perform such functions.

SPLITTING PROGRAMS INTO MULTIPLE BANKS

When all of the above techniques are insufficient to compact your application into a single bank, the next step to consider is splitting the code into two or more banks. The first bank can copy up to 12K of code into RAM. Then it can copy and call a small routine in RAM that switches to another bank and continues with a second (or more) loader to move the rest of the code into RAM. Or a single loader routine could be moved into RAM that copies all of the code from each bank sequentially. After all of the code is copied it is even possible to switch to a standard Applesoft image before executing the application. That would allow you to count on the F8 (and Applesoft for that matter) routines being available to your program without taking up space in your images.

This is exactly how our DOS image banks work. We have even included a short warning dialog that is called if you attempt to launch one of the secondary banks by itself. See the listing below.

```
                ** ROMXDOS **

                *   Copyright 2000 Jeff Mazur

                * V 0.990 INITIAL RELEASE
                * V 0.991 Bypass countdown after warning.
                * V 0.992 Added tertiary image.

                                ;V 0.992 ROMXDOS  DOS 3.3 IMAGE

                BASL      EQU   $78
                BASH      EQU   BASL+1
                RealBank  EQU   $02A6     ;Bank save for DOS call
                RAM_LOC   EQU   $2000
                BANK0     EQU   $CFE0
                ROM2RAM   EQU   $C081     ;Enable ROM Read/RAM Write
                LCWPROT   EQU   $C082     ;Write Protect LC
                DOSSTART  EQU   $9D84     ;Assume 48K machine
                KYBD      EQU   $C000     ;Keyboard address
                KYBDSTB   EQU   $C010     ;Keyboard strobe reset
                RMXInit   EQU   $1012     ;ROMX Init (Ver 0.992 and up)
                RMXDoMenu EQU   $103C     ;ROMX DoMenu
                RMXStrt   EQU   $DFD8     ;ROMX Start location

                          ORG   $FB00     ;Runs at $FB00


                Launch
                *------------------------------
                *  DISPLAY WARNING
                *------------------------------
                CLRSCRN
FB00: A9 A0              LDA   #" "      Store spaces to entire screen
FB02: A2 04              LDX   #/$0400      ;TARGET
FB04: A0 00              LDY   #0
FB06: 84 00              STY   $00
FB08: 86 01     :1       STX   $01

FB0A: 91 00     :2       STA   ($00),Y
FB0C: C8                 INY
FB0D: D0 FB              BNE   :2
FB0F: E8                 INX
FB10: E0 08              CPX   #$08         ;END OF TARGET
FB12: D0 F4              BNE   :1

                DispWarn
FB14: A2 04              LDX   #4           ;4 lines
FB16: 8A        :1       TXA
FB17: 20 2B FB           JSR   PrintOneLine
FB1A: CA                 DEX
FB1B: 10 F9              BPL   :1

FB1D: AD 00 C0  :2       LDA   KYBD
```

```
FB20: 10 FB              BPL   :2
FB22: AD 10 C0           LDA   KYBDSTB

FB25: 20 42 FF           JSR   RamCpy
FB28: 4C 50 20           JMP   Bk2Menu-$DF00

                *-----------------------------------
                *  Print string on given line
                *  ON ENTRY, A = line# X = msg#
                *-----------------------------------
                PrintOneLine
FB2B: 20 81 FF           JSR   BASCALC          ;Get start of line
FB2E: 8A                 TXA    ;Get msg#
FB2F: 0A                 ASL    ;multiply by 2
FB30: A8                 TAY
FB31: B9 EB FB           LDA   Msg_Add_Tbl,Y
FB34: 85 1E              STA   $1E
FB36: B9 EC FB           LDA   Msg_Add_Tbl+1,Y
FB39: 85 1F              STA   $1F
FB3B: A0 00              LDY   #0
FB3D: B1 1E     :1       LDA   ($1E),Y
FB3F: F0 05              BEQ   :2
FB41: 91 78              STA   (BASL),Y
FB43: C8                 INY
FB44: D0 F7              BNE   :1
FB46: 60        :2       RTS


FB47: A0 A0 A0  menu01   ASC   "          THIS IS A DOS IMAGE           ",00
FB70: A0 A0 A0  menu02   ASC   "         AND CANNOT BE LAUNCHED         ",00
FB99: A0 A0 A0  menu03   ASC   "                                        ",00
FBC2: A0 A0 A0  menu04   ASC   "     PRESS ANY KEY TO RETURN TO MENU    ",00


                Msg_Add_Tbl
FBEB: 99 FB 47           DW    menu03,menu01,menu02,menu03,menu04


                         ERR   *-1/$FF00
FBF5: 00 00 00           DS    $FF00-*



                *********   DOS LOADER  *********  Runs at $FF00

                DOSLoad
FF00: 20 42 FF           JSR   RamCpy
FF03: 4C 06 20           JMP   MyCode-$DF00

                *--------------------------------------
                *  START OF BOOT CODE
                *--------------------------------------

                MyCode
FF06: A2 FA     PASS1    LDX   #/$FA00          ;SOURCE
```

```
FF08: A9 FB              LDA   #/$FB00          ;SOURCE END
FF0A: 8D 7A 20           STA   SrcEnd-$DF00+1
FF0D: A9 03              LDA   #/$0300          ;TARGET
FF0F: 20 63 20           JSR   CopyLp-$DF00

FF12: A2 D0     PASS2    LDX   #/$D000          ;SOURCE
FF14: A9 F3              LDA   #/$F300          ;SOURCE END
FF16: 8D 7A 20           STA   SrcEnd-$DF00+1
FF19: A9 9D              LDA   #/$9D00          ;TARGET (assume 48K)
FF1B: 20 63 20           JSR   CopyLp-$DF00

FF1E: 20 B1 20  BOOT     JSR   TripleChk-$DF00  ;see if there's more to do
FF21: 20 59 20           JSR   SelBank0-$DF00   ;Activate BANK 0
FF24: AE A6 02           LDX   RealBank         ;Activate real image
FF27: BD E0 CF           LDA   BANK0,X

FF2A: D8                 CLD
FF2B: A9 00              LDA   #$00             ;Applesoft ONERR flag
FF2D: 85 D8              STA   $D8
FF2F: A9 FF              LDA   #$FF             ;Integer RUNMODE flag
FF31: 85 D9              STA   $D9
FF33: 20 84 FE           JSR   $FE84            ;SETNORM
FF36: 20 2F FB           JSR   $FB2F            ;INIT
FF39: 20 93 FE           JSR   $FE93            ;SETVID
FF3C: 20 89 FE           JSR   $FE89            ;SETKBD
               ;       JSR $FB60            ;HOME & PRINT APPLE ][
               ;       JSR $FBDD            ;BELL
FF3F: 4C 84 9D           JMP   DOSSTART         ;DOS COLDSTART

                RamCpy
FF42: A0 00              LDY   #0
FF44: B9 00 FF  CpyLp    LDA   DOSLoad,Y
FF47: 99 00 20           STA   RAM_LOC,Y
FF4A: C8                 INY
FF4B: C0 ED              CPY   #MyCode_End
FF4D: D0 F5              BNE   CpyLp
FF4F: 60                 RTS

                Bk2Menu
FF50: 20 59 20           JSR   SelBank0-$DF00   ;Get to Bank 0
FF53: 20 12 10  RXInit   JSR   RMXInit          ;Init Firmware
FF56: 4C 3C 10  RXMenu   JMP   RMXDoMenu        ;Launch ROMX menu

                SelBank0
FF59: 2C CA CA           BIT   $CACA            ;Select Bank 0 from another bank
FF5C: 2C CA CA           BIT   $CACA
FF5F: 2C FE CA           BIT   $CAFE
FF62: 60                 RTS

FF63: 85 03     CopyLp   STA   $03
FF65: A9 00              LDA   #$00
FF67: 85 00              STA   $00
FF69: 85 02              STA   $02
FF6B: 86 01     CpyLp2   STX   $01
FF6D: A0 00              LDY   #$00
```

```

FF6F: B1 00   :2       LDA   ($00),Y          ;do 256 bytes
FF71: 91 02            STA   ($02),Y
FF73: C8               INY
FF74: D0 F9            BNE   :2
FF76: E6 03            INC   $03
FF78: E8               INX
FF79: E0 F3   SrcEnd   CPX   #/$F300          ;END OF SOURCE
FF7B: D0 EE            BNE   CpyLp2           ;do xx pages
FF7D: 60               RTS


              BASCALC1
FF7E: 18               CLC    ;Compensate for top menu lines
FF7F: 69 01            ADC   #1               ;and fall into...

              BASCALC
FF81: 48               PHA    ;CALC BASE ADR IN BASL,H
FF82: 4A               LSR    ;  FOR GIVEN LINE NO
FF83: 29 03            AND   #$03
FF85: 09 04            ORA   #$04
FF87: 85 79            STA   BASH
FF89: 68               PLA
FF8A: 29 18            AND   #$18
FF8C: 90 02            BCC   BSCLC2
FF8E: 69 7F            ADC   #$7F
FF90: 85 78   BSCLC2   STA   BASL
FF92: 0A               ASL
FF93: 0A               ASL
FF94: 05 78            ORA   BASL
FF96: 85 78            STA   BASL
FF98: 60               RTS

FF99: C9 B1   DecBnk   CMP   #"1"             ;Valid numerical bank?
FF9B: 90 13            BLT   NotAlpNum
FF9D: C9 BA   DecHi    CMP   #":"
FF9F: B0 04            BGE   Alpha
FFA1: 38               SEC
FFA2: E9 B0            SBC   #"0"             ;Get hex value for bank
FFA4: 60               RTS    ;and return with 0x01-09

              Alpha
                              ; AND #$DF            ;HANDLE LOWER CASE
FFA5: C9 C1            CMP   #"A"             ;Valid alpha bank?
FFA7: 90 07            BLT   NotAlpNum
FFA9: C9 C7            CMP   #"G"
FFAB: B0 03            BGE   NotAlpNum
FFAD: 38               SEC    ;Get hex value for bank
FFAE: E9 B7            SBC   #"7"             ;and return with 0x0A-0F

              NotAlpNum
FFB0: 60               RTS    ;return with original key


              TripleChk
```

```
FFB1: AD A6 02             LDA   RealBank          ;recall original image
FFB4: 20 7E 20             JSR   BASCALC1-$DF00

                DOS_Check
FFB7: A0 24                LDY   #36               ;position 37 for DOS BANK
FFB9: B1 78                LDA   (BASL),Y          ;Get DOS bank
FFBB: 20 99 20             JSR   DecBnk-$DF00
FFBE: 20 7E 20             JSR   BASCALC1-$DF00
FFC1: A0 22                LDY   #34               ;position 37 for 3rd BANK
FFC3: B1 78                LDA   (BASL),Y          ;Check for 3rd bank
FFC5: C9 A6                CMP   #"&"
FFC7: D0 23                BNE   Finish
FFC9: C8                   INY
FFCA: C8                   INY
FFCB: B1 78                LDA   (BASL),Y          ;Get 3rd bank
FFCD: 20 99 20             JSR   DecBnk-$DF00
FFD0: AA                   TAX
FFD1: 20 59 20             JSR   SelBank0-$DF00    ;Get back to bank 0
FFD4: BD E0 CF             LDA   BANK0,X           ;Activate 3rd bank
FFD7: AD 81 C0             LDA   ROM2RAM           ;Enable LC Write
FFDA: AD 81 C0             LDA   ROM2RAM

FFDD: A2 D0     PASS3      LDX   #/$D000           ;SOURCE
FFDF: A9 00                LDA   #/$0000           ;SOURCE END
FFE1: 8D 7A 20             STA   SrcEnd-$DF00+1
FFE4: A9 D0                LDA   #/$D000           ;TARGET
FFE6: 20 63 20             JSR   CopyLp-$DF00

FFE9: 2C 82 C0             BIT   LCWPROT           ;Write Protect LC

FFEC: 60        Finish     RTS


                MyCode_End


                           ERR   *-1/$FFFC
FFED: 00 00 00             DS    $FFFC-*

FFFC: 00 FB                DA    Launch             ;RESET Vector
FFFE: 00 FF                DA    DOSLoad            ;NMI Vector

```

When a bank is selected in ROMX that has the “&Dn” command in its description, it will automatically store the bank selected in address $02A6. Then it will enable bank “n” and launch its code via the vector at location $FFFE/F (normally used for the BRK vector). If bank “n” is mistakenly launched directly, it will begin execution at the normal Reset vector $FFFC/D. This can be used to display a warning message and return to the ROMX menu.

Here is a detailed breakdown of the code:

Lines 11-23 set up the equates used by the remaining code. Note the RealBank location at $02A6 and the firmware entry points RMXInit and RMXDoMenu, which initialize and launch the ROMX firmware respectively.

When the image is called via the &Dn command, it will begin execution at the address pointed to by FFFE, or from line 258, $FF00 (DOSLoad). This routine executes at line 98 and starts by copying the loader code itself from FFF0-FFEC into RAM at $2000. Then at line 100 it JMPs to its RAM copy starting at $2006 (MyCode). This then copies DOS and the page 3 vectors in two passes. Next it checks to see if there is a link to load a third image into the Language Card (TripleChk). If so, it activates that bank, enables writing to the Language Card, and copies the image into the card. After that the card is write protected.

The boot code continues on line 120 by activating bank 0 so that it can select the original RealBank image. Then, assuming that image contains the Autostart F8 Monitor, we perform the basic system initialization, optionally print APPLE][followed by a beep, and then drop into the DOS coldstart entry to continue booting and linking to DOS.

On the other hand, if the image is called directly from the ROMX menu, it will begin execution at the address pointed to by FFFC, or from line 257, $FB00 (Launch). This code begins with lines 34-45 which store a blank or space character at each location from $400-$7FF, effectively clearing the screen. Then at lines 48-52 we display 4 lines of text, which puts up the warning message. Lines 54-56 wait for a keypress and then line 58 copies the launch code into RAM just so we can jump back to the ROMX menu.

Lines 61-80 comprise a subroutine (lifted from the ROMX firmware) to print a single line of text on the screen. This is used with the strings and address table in lines 83-90. The Bk2Menu routine starting at line 146 handles the return to the ROMX menu. It does this by activating the ROMX firmware in bank 0, initializing the code, and then jumping to the entry point where it draws the menu. Note that we would normally need to move the ROMX firmware from ROM to RAM as well. But since we know that we got here from that code in the first place, it is safe to assume it is still in RAM.

CREATING IMAGES THAT WORK WITH ACCELERATORS

If you want your image to work correctly when running with an accelerator, there are a few extra steps to take. The most important is that accesses to memory may be bypassed by the accelerator's built-in RAM or cache. This can cause issues with the ROMX softswitches and previously loaded data from the ROM space from other banks, including the firmware.

For example, whenever you switch back from one bank to the firmware in bank 0, you need to execute the CACA CACA CAFE sequence. While the accelerator is doing its job trying to speed things up it may see no need to actually access the "memory" location $CACA twice in a row. So your code may not perform as expected in this case. There are several ways to remedy this situation:

1. Temporarily tell the accelerator to slow down and resume normal access
2. Explicitly disable the accelerator until your code is complete; then re-enable it.
3. Flush the accelerator's cache or RAM to cause it to re-read the memory address.

In most cases option 1 will suffice. All accelerators need a way to resume normal operation when accessing a Disk II drive since its code is driven by software timing loops. Because of this we can always pick a slot (usually 6) where a disk controller would sit and access its ROM or I/O space briefly. A BIT $C0E0 instruction will usually do the trick. This will cause the accelerator to slow down typically for about 50 mS.

Finally, with respect to the ZIP CHIPs, it is also possible that the internal cache will hold data from other banks that were switched in during the launch process. This will cause unexpected results when run with the accelerator on but will work fine when it is disabled. The solution for this problem is to completely flush the 8K of cache on the ZIP CHIP. The code below is a straightforward way to do this. You would probably execute this near the start of your image loader routine to ensure that you copy the desired data from the ROM space into RAM.

```
CACHEBUSTER  LDA #$00    ; Starting base address
             LDX #$20    ; Starting page and count
             STA $00     ; Set ($00) to base address
             STX $01     ;
             LDY #$00    ; Initialize offset

:1           LDA ($00),Y ; Forcibly cache memory
             INY         ;
             BNE :1      ;
             INC $01     ; Next memory page
             DEX         ;
             BNE :1      ; Until $20 pages (8K) are done
```