ROMifying a Program for Uploading to ROMX

PART I – BASIC PROCESS

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).

Additionally, 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, e.g. to identify machine or ROM versions.
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 always be replaced by a single JMP instruction to the same address, saving one byte. 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.

```
                  1    ** ROMXDOS **
                  2
                  3    *   Copyright 2000 Jeff Mazur
                  4
                  5    * V 0.990 INITIAL RELEASE
                  6    * V 0.991 Bypass countdown after warning.
                  7    * V 0.992 Added tertiary image.
                  8
                  9                   ;V 0.992 ROMXDOS  DOS 3.3 IMAGE
                  10
                  11   BASL      EQU   $78
                  12   BASH      EQU   BASL+1
                  13   RealBank  EQU   $02A6     ;Bank save for DOS call
                  14   RAM_LOC   EQU   $2000
                  15   BANK0     EQU   $CFE0
                  16   ROM2RAM   EQU   $C081     ;Enable ROM Read/RAM Write
                  17   LCWPROT   EQU   $C082     ;Write Protect LC
                  18   DOSSTART  EQU   $9D84     ;Assume 48K machine
                  19   KYBD      EQU   $C000     ;Keyboard address
                  20   KYBDSTB   EQU   $C010     ;Keyboard strobe reset
                  21   RMXInit   EQU   $1012     ;ROMX Init (Ver 0.992 and up)
                  22   RMXDoMenu EQU   $103C     ;ROMX DoMenu
                  23   RMXStrt   EQU   $DFD8     ;ROMX Start location
                  24
                  25             ORG   $FB00     ;Runs at $FB00
                  27
                  28
                  29   Launch
                  30   *------------------------------
                  31   *  DISPLAY WARNING
                  32   *------------------------------
                  33   CLRSCRN
FB00: A9 A0       34             LDA   #" "      Store spaces to entire screen
FB02: A2 04       35             LDX   #/$0400      ;TARGET
FB04: A0 00       36             LDY   #0
FB06: 84 00       37             STY   $00
FB08: 86 01       38   :1        STX   $01
                  39
FB0A: 91 00       40   :2        STA   ($00),Y
FB0C: C8          41             INY
FB0D: D0 FB       42             BNE   :2
FB0F: E8          43             INX
FB10: E0 08       44             CPX   #$08         ;END OF TARGET
FB12: D0 F4       45             BNE   :1
                  46
                  47   DispWarn
FB14: A2 04       48             LDX   #4           ;4 lines
FB16: 8A          49   :1        TXA
FB17: 20 2B FB    50             JSR   PrintOneLine
FB1A: CA          51             DEX
FB1B: 10 F9       52             BPL   :1
                  53
FB1D: AD 00 C0    54   :2        LDA   KYBD
FB20: 10 FB       55             BPL   :2
```

```
FB22: AD 10 C0  56            LDA   KYBDSTB
                57
FB25: 20 42 FF  58            JSR   RamCpy
FB28: 4C 50 20  59            JMP   Bk2Menu-$DF00
                60
                61   *-----------------------------------
                62   *  Print string on given line
                63   *  ON ENTRY, A = line# X = msg#
                64   *-----------------------------------
                65   PrintOneLine
FB2B: 20 81 FF  66            JSR   BASCALC          ;Get start of line
FB2E: 8A        67            TXA    ;Get msg#
FB2F: 0A        68            ASL    ;multiply by 2
FB30: A8        69            TAY
FB31: B9 EB FB  70            LDA   Msg_Add_Tbl,Y
FB34: 85 1E     71            STA   $1E
FB36: B9 EC FB  72            LDA   Msg_Add_Tbl+1,Y
FB39: 85 1F     73            STA   $1F
FB3B: A0 00     74            LDY   #0
FB3D: B1 1E     75   :1       LDA   ($1E),Y
FB3F: F0 05     76            BEQ   :2
FB41: 91 78     77            STA   (BASL),Y
FB43: C8        78            INY
FB44: D0 F7     79            BNE   :1
FB46: 60        80   :2       RTS
                81
                82
FB47: A0 A0 A0  83   menu01   ASC   "          THIS IS A DOS IMAGE          ",00
FB70: A0 A0 A0  84   menu02   ASC   "         AND CANNOT BE LAUNCHED        ",00
FB99: A0 A0 A0  85   menu03   ASC   "                                       ",00
FBC2: A0 A0 A0  86   menu04   ASC   "     PRESS ANY KEY TO RETURN TO MENU   ",00
                87
                88
                89   Msg_Add_Tbl
FBEB: 99 FB 47  90            DW    menu03,menu01,menu02,menu03,menu04
                91
                92
                93            ERR   *-1/$FF00
FBF5: 00 00 00  94            DS    $FF00-*


                95
                96   *********   DOS LOADER  *********  Runs at $FF00
                97
                98   DOSLoad
FF00: 20 42 FF  99            JSR   RamCpy
FF03: 4C 06 20  100           JMP   MyCode-$DF00
                101
                102  *--------------------------------------
                103  *  START OF BOOT CODE
                104  *--------------------------------------
                105
                106  MyCode
FF06: A2 FA     107  PASS1    LDX   #/$FA00          ;SOURCE
FF08: A9 FB     108           LDA   #/$FB00          ;SOURCE END
```

```
FF0A: 8D 7A 20  109           STA   SrcEnd-$DF00+1
FF0D: A9 03     110           LDA   #/$0300          ;TARGET
FF0F: 20 63 20  111           JSR   CopyLp-$DF00
                112
FF12: A2 D0     113  PASS2    LDX   #/$D000          ;SOURCE
FF14: A9 F3     114           LDA   #/$F300          ;SOURCE END
FF16: 8D 7A 20  115           STA   SrcEnd-$DF00+1
FF19: A9 9D     116           LDA   #/$9D00          ;TARGET (assume 48K)
FF1B: 20 63 20  117           JSR   CopyLp-$DF00
                118
FF1E: 20 B1 20  119  BOOT     JSR   TripleChk-$DF00  ;see if there's more to do
FF21: 20 59 20  120           JSR   SelBank0-$DF00   ;Activate BANK 0
FF24: AE A6 02  121           LDX   RealBank         ;Activate real image
FF27: BD E0 CF  122           LDA   BANK0,X
                123
FF2A: D8        124           CLD
FF2B: A9 00     125           LDA   #$00             ;Applesoft ONERR flag
FF2D: 85 D8     126           STA   $D8
FF2F: A9 FF     127           LDA   #$FF             ;Integer RUNMODE flag
FF31: 85 D9     128           STA   $D9
FF33: 20 84 FE  129           JSR   $FE84            ;SETNORM
FF36: 20 2F FB  130           JSR   $FB2F            ;INIT
FF39: 20 93 FE  131           JSR   $FE93            ;SETVID
FF3C: 20 89 FE  132           JSR   $FE89            ;SETKBD
                133 ;       JSR $FB60            ;HOME & PRINT APPLE ][
                134 ;       JSR $FBDD            ;BELL
FF3F: 4C 84 9D  135           JMP   DOSSTART         ;DOS COLDSTART
                136
                137  RamCpy
FF42: A0 00     138           LDY   #0
FF44: B9 00 FF  139  CpyLp    LDA   DOSLoad,Y
FF47: 99 00 20  140           STA   RAM_LOC,Y
FF4A: C8        141           INY
FF4B: C0 ED     142           CPY   #MyCode_End
FF4D: D0 F5     143           BNE   CpyLp
FF4F: 60        144           RTS
                145
                146  Bk2Menu
FF50: 20 59 20  147           JSR   SelBank0-$DF00   ;Get to Bank 0
FF53: 20 12 10  148  RXInit   JSR   RMXInit          ;Init Firmware
FF56: 4C 3C 10  149  RXMenu   JMP   RMXDoMenu        ;Launch ROMX menu
                150
                151  SelBank0
FF59: 2C CA CA  152           BIT   $CACA            ;Select Bank 0 from another bank
FF5C: 2C CA CA  153           BIT   $CACA
FF5F: 2C FE CA  154           BIT   $CAFE
FF62: 60        155           RTS
                156
FF63: 85 03     157  CopyLp   STA   $03
FF65: A9 00     158           LDA   #$00
FF67: 85 00     159           STA   $00
FF69: 85 02     160           STA   $02
FF6B: 86 01     161  CpyLp2   STX   $01
FF6D: A0 00     162           LDY   #$00
                163
```

```
FF6F: B1 00     164  :2       LDA   ($00),Y          ;do 256 bytes
FF71: 91 02     165           STA   ($02),Y
FF73: C8        166           INY
FF74: D0 F9     167           BNE   :2
FF76: E6 03     168           INC   $03
FF78: E8        169           INX
FF79: E0 F3     170  SrcEnd   CPX   #/$F300          ;END OF SOURCE
FF7B: D0 EE     171           BNE   CpyLp2           ;do xx pages
FF7D: 60        172           RTS
                173
                174
                175  BASCALC1
FF7E: 18        176           CLC    ;Compensate for top menu lines
FF7F: 69 01     177           ADC   #1               ;and fall into...
                178
                179  BASCALC
FF81: 48        180           PHA    ;CALC BASE ADR IN BASL,H
FF82: 4A        181           LSR    ;  FOR GIVEN LINE NO
FF83: 29 03     182           AND   #$03
FF85: 09 04     183           ORA   #$04
FF87: 85 79     184           STA   BASH
FF89: 68        185           PLA
FF8A: 29 18     186           AND   #$18
FF8C: 90 02     187           BCC   BSCLC2
FF8E: 69 7F     188           ADC   #$7F
FF90: 85 78     189  BSCLC2   STA   BASL
FF92: 0A        190           ASL
FF93: 0A        191           ASL
FF94: 05 78     192           ORA   BASL
FF96: 85 78     193           STA   BASL
FF98: 60        194           RTS
                195
FF99: C9 B1     196  DecBnk   CMP   #"1"             ;Valid numerical bank?
FF9B: 90 13     197           BLT   NotAlpNum
FF9D: C9 BA     198  DecHi    CMP   #":"
FF9F: B0 04     199           BGE   Alpha
FFA1: 38        200           SEC
FFA2: E9 B0     201           SBC   #"0"             ;Get hex value for bank
FFA4: 60        202           RTS    ;and return with 0x01-09
                203
                204  Alpha
                205                  ; AND #$DF            ;HANDLE LOWER CASE
FFA5: C9 C1     206           CMP   #"A"             ;Valid alpha bank?
FFA7: 90 07     207           BLT   NotAlpNum
FFA9: C9 C7     208           CMP   #"G"
FFAB: B0 03     209           BGE   NotAlpNum
FFAD: 38        210           SEC    ;Get hex value for bank
FFAE: E9 B7     211           SBC   #"7"             ;and return with 0x0A-0F
                212
                213  NotAlpNum
FFB0: 60        214           RTS    ;return with original key
                215
                216
                217  TripleChk
FFB1: AD A6 02  218           LDA   RealBank         ;recall original image
```

```
FFB4: 20 7E 20  219           JSR   BASCALC1-$DF00
                220
                221  DOS_Check
FFB7: A0 24     222           LDY   #36              ;position 37 for DOS BANK
FFB9: B1 78     223           LDA   (BASL),Y         ;Get DOS bank
FFBB: 20 99 20  224           JSR   DecBnk-$DF00
FFBE: 20 7E 20  225           JSR   BASCALC1-$DF00
FFC1: A0 22     226           LDY   #34              ;position 37 for 3rd BANK
FFC3: B1 78     227           LDA   (BASL),Y         ;Check for 3rd bank
FFC5: C9 A6     228           CMP   #"&"
FFC7: D0 23     229           BNE   Finish
FFC9: C8        230           INY
FFCA: C8        231           INY
FFCB: B1 78     232           LDA   (BASL),Y         ;Get 3rd bank
FFCD: 20 99 20  233           JSR   DecBnk-$DF00
FFD0: AA        234           TAX
FFD1: 20 59 20  235           JSR   SelBank0-$DF00   ;Get back to bank 0
FFD4: BD E0 CF  236           LDA   BANK0,X          ;Activate 3rd bank
FFD7: AD 81 C0  237           LDA   ROM2RAM          ;Enable LC Write
FFDA: AD 81 C0  238           LDA   ROM2RAM
                239
FFDD: A2 D0     240  PASS3    LDX   #/$D000          ;SOURCE
FFDF: A9 00     241           LDA   #/$0000          ;SOURCE END
FFE1: 8D 7A 20  242           STA   SrcEnd-$DF00+1
FFE4: A9 D0     243           LDA   #/$D000          ;TARGET
FFE6: 20 63 20  244           JSR   CopyLp-$DF00
                245
FFE9: 2C 82 C0  246           BIT   LCWPROT          ;Write Protect LC
                247
FFEC: 60        248  Finish   RTS
                249
                250
                251  MyCode_End
                252
                253
                254           ERR   *-1/$FFFC
FFED: 00 00 00  255           DS    $FFFC-*
                256
FFFC: 00 FB     257           DA    Launch            ;RESET Vector
FFFE: 00 FF     258           DA    DOSLoad           ;NMI Vector
                259
```

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.

PART III – THE ROMXe & ROMXc

With the release of the ROMXc and ROMXe (hereafter referred to as just ROMXc) the individual bank size has been increased to a whole 32K (minus two C0xx pages). This greatly opens up the possibilities for adding applications to the ROMX. Plus there are some new features of these boards which can make ROMified applications more powerful. You can also create 16K images if that's all you need but we're going to assume you want as much space as possible for your application. There is one slight issue however if you're going to create a 32K file. DOS 3.3 will not let you save a file with a length of $8000, which is required to build a 32K image for ROMXc. So you will need to either work in ProDOS (and constantly convert back and forth) or modify DOS to allow this. The Tutorial dsk file that accompanies this document has such a modified DOS.

Now let's go over the hardware implementations of the ROMXc). From the ROMX API Guide we have the following locations to use:

```
                ROMXc/e SOFTSWITCHES

BANK0-F         $F800-F80F ;Select Main System Bank 0-F
TXTBANK0-F      $F810-F81F ;Select Main Text Bank 0-F
MAIN_LO         $F820      ;Set System ROM Lo
MAIN_HI         $F821      ;Set System ROM Hi
TXT_LO          $F822      ;Set Text ROM Lo
TXT_HI          $F823      ;Set Text ROM Hi
TMP_LOWER       $F824      ;Set Temp Bank Lo
TMP_UPPER       $F825      ;Set Temp Bank Hi
TMP_BANK0-F     $F830-F83F ;Select Temp Bank 0-F
TMP_TXTBANK0-F  $F840-F84F ;Select Temp Text Bank 0-F
SEL_TBANK       $F850      ;Select Temp Banks
SEL_MBANK       $F851      ;Select Main Banks
```

With these addresses we can build an image that is activated by the ROMX menu and which can do almost anything. For the rest of this tutorial we will see how to take a binary program image, prepare it for ROM, and then craft a loader program that can be executed on launch to move the program into the appropriate RAM location.

ROMifiying CHOPLIFTER!

In attempting to make Choplifter run on ROMXc, I started with a cracked version of the program that occupies 128 sectors on disk (see CHOPLIFTER on the dsk image file that accompanies this document). That amounts to just under 32K so we know this will be a tight fit! Examining the file with Copy II Plus reveals that this file loads at $07FD and the actual length is $7D10 (ending at $850D). The actual starting address of the program is $2490.

We know the addresses $C000-$C0FF are unusable in each bank of ROM. Therefore after losing two pages, we will have exactly $7DFF bytes left for the image and loader combined. Further examination of the Choplifter binary file shows that the beginning of the file is just a JMP to the starting address of the program so the bytes from $07FD-$07FF are not needed. And after allocating

4 bytes for the RESET and BRK vectors at $FFFC that leaves us $7DFF-7D10-4-3=$E8 bytes to write our loader routine.

Furthermore, since we will need to switch banks while transferring the code, we have to account for the ROM contents changing from underneath us so to speak. This is accomplished by having some or all of the loader code run from RAM. Since we know that RAM above $850B is not part of the program we can safely put a copy of our loader routine there and run it without worrying about bank changes crashing our code. Thus I chose to copy and run the loader code at the top of RAM, $BF00 (at first glance anyway). So the memory map was shaping up this way:

LOW BANK
$FFFC-$FFFF : RESET and (unused) BRK vectors
$FF00-$FFFB : Loader routine including code to move itself into RAM

That would leave $C100-$FEFF in the low bank plus $C100-$FFFF in the high bank to hold the program. Unfortunately, that only adds up to $7D00 bytes and we actually need $7D0D. So close!

The standard RAM copy routine that I used in the previous examples copies one page (256 bytes) at a time and always on page boundaries. While this simplifies and shortens the code, it often copies more bytes than we actually need. And in this case we need to save every byte we can. So I made one more tweak to the loader code assuming that I wouldn't need all of the space from $FF00-$FFFB for the loader. This opened up any unused space there for those remaining few bytes. Specifically, I opted to start the loader code at $FF80, which would then place it at $BF80 after moving to RAM. That opened up $80 more bytes for program code, which was certainly enough.

CREATING CHOPLIFTER FROM THE TEMPLATE

To re-create a CHOPLIFTER image file from scratch using the TEMPLATE.32, perform the following steps:

1. BLOAD the CHOPLIFTER file at $0FFD (to eliminate the initial JMP)
2. BSAVE CHOP1, A$1000, L$3E80
3. BSAVE CHOP2, A$4E80, L$3F00
4. BLOAD TEMPLATE.32
5. BLOAD CHOP1, A$1100
6. BLOAD CHOP2, A$5100
7. CALL -151 to enter Monitor
8. Modify the start address to begin execution: 4FC5:90 24
9. BSAVE MY CHOPLIFTER, A$1000, L$8000 (delete CHOP1 or use another disk)

When Uploading to ROMXc don't forget to add the &Pn command to the Description. It will not run correctly without it.

Refer to the Listing below as we show line-by-line how the loader works. Starting at location $FF80 in ROM, the Launch code begins by copying itself from the $FF page in lower ROM to location $BF80 in RAM (lines 30-33). Then we JMP to our code at $BF8D to continue the Launch code from RAM.

Lines 35-47 now copy the program to location $800. First we copy the lower bank from $C100-FFFF into $800-$46FF. The last byte of program code is now at $467F with a bogus copy of the loader in $4680-$46FF but we will overwrite that in the next step.

Now we switch back to the firmware bank 0 (line 39) so that we can select the upper sub-bank (line 40). Then we re-activate our DOS bank (lines 41-42) and continue to copy the rest of the program from the upper bank. But our copy destination now needs to start right after the last location from the first pass. That would be location $4680. So we modify our CopyLp routine to move pages to locations starting at xx80. Thus we copy $C100-$FFFF into $4680-$857F. All the program code is now in RAM at the correct location.

Finishing up, we get back to bank 0 and retrieve the initial System ROM bank passed to us by the firmware at $02A6 (line 49-50). Then we preset and activate that user bank (lines 51-52). Finally, lines 53-57 perform some initialization and then jump to the starting location of the program, $2490 (after manually patching).

TEMPLATE.32 LISTING

```
                    1    *           ROMXce 32K IMAGE TEMPLATE
                    2    *           J. Mazur  3/29/21
                    3
                    4    *           PART 1 - 0000-3FFF MAIN ROM
                    5    *           PART 2 - 4000-7FFF AUX ROM
                    6    *
                    7    *           Part 2 overwrites $80 bytes of part 1
                    8
                    9
                    10               DSK   TEMPIMAGE
                    11
                    12   OurBank     EQU  $0287      ;Our bank
                    13   RealBank    EQU  $02A6      ;Bank we need to run in
                    14   RAM_LOC     EQU  $0800      ;Must be less than $4200
                    15   START_LOC   EQU  $0800      ;Start of program
                    16   INTCXROMOFF EQU  $C006      ;Enable slot ROM C100-C1FF
                    17   INTCXROMON  EQU  $C007      ;Enable main ROM C100-C1FF
                    18   BANK0       EQU  $F800
                    19   TMP_LOWER   EQU  $F824
                    20   TMP_UPPER   EQU  $F825
                    21   TMP_BANK0   EQU  $F830      ;Latch temp bank#
                    22   SEL_MBANK   EQU  $F851      ;Select Main bank reg
                    23   INIT        EQU  $FB2F
                    24   SETVID      EQU  $FE93
                    25   SETKBD      EQU  $FE89
                    26
                    27            ORG   $BF80        ;Start of Image
                    28
BF80: 8D 07 C0      29   Launch   STA   INTCXROMON   ;Turn //e ROM ON
BF83: A2 FF         30            LDX   #/$FF80      ;Move our code
BF85: A9 BF         31            LDA   #$BF         ;to $BF80
BF87: 20 D1 FF      32            JSR   CopyLp+$4000
BF8A: 4C 8D BF      33            JMP   Launch2-Launch+$BF80  ;and execute it there
                    34
BF8D: A2 C1         35   Launch2  LDX   #/$C100      ;SOURCE hi-byte
BF8F: A9 08         36            LDA   #/RAM_LOC    ;TARGET hi-byte
BF91: 20 D1 BF      37            JSR   CopyLp
                    38
BF94: 20 C7 BF      39            JSR   SelBnk0      ;Enable firmware Bank 0
BF97: 2C 25 F8      40            BIT   TMP_UPPER    ;Switch sub-bank for Part 2
BF9A: AE 87 02      41            LDX   OurBank      ;Get back to our bank & copy
BF9D: BD 30 F8      42            LDA   TMP_BANK0,X
BFA0: A2 C1         43            LDX   #/$C100      ;SOURCE hi-byte
BFA2: A9 80         44            LDA   #$80         ;move 1/2 pages
BFA4: 8D D4 BF      45            STA   CopyLp+3
BFA7: A9 46         46            LDA   #/RAM_LOC+$3E00 ;TARGET hi-byte
BFA9: 20 D1 BF      47            JSR   CopyLp
                    48
BFAC: 20 C7 BF      49            JSR   SelBnk0
BFAF: AE A6 02      50            LDX   RealBank
BFB2: BD 00 F8      51            LDA   BANK0,X      ;Prepare to launch
BFB5: 2C 51 F8      52            BIT   SEL_MBANK
BFB8: 20 2F FB      53            JSR   INIT
```

```
BFBB: 20 93 FE  54            JSR   SETVID
BFBE: 20 89 FE  55            JSR   SETKBD
BFC1: 8D 06 C0  56            STA   INTCXROMOFF  ;Turn //e ROM OFF
BFC4: 4C 00 08  57            JMP   START_LOC
                58
                59   SelBnk0
BFC7: 2C CA FA  60            BIT   $FACA
BFCA: 2C CA FA  61            BIT   $FACA
BFCD: 2C FE FA  62            BIT   $FAFE
BFD0: 60        63            RTS
                64
BFD1: 85 03     65   CopyLp   STA   $03
BFD3: A9 00     66            LDA   #$00
BFD5: 85 02     67            STA   $02
BFD7: A9 00     68            LDA   #$00
BFD9: 85 00     69            STA   $00
BFDB: 86 01     70   CpyLp2   STX   $01
BFDD: A0 00     71            LDY   #$00
                72
BFDF: B1 00     73   :2       LDA   ($00),Y             ;do 256 bytes
BFE1: 91 02     74            STA   ($02),Y
BFE3: C8        75            INY
BFE4: D0 F9     76            BNE   :2
BFE6: E6 03     77            INC   $03
BFE8: E8        78            INX
BFE9: E0 00     79   SrcEnd   CPX   #/$0000             ;END OF SOURCE  hi-byte
BFEB: D0 EE     80            BNE   CpyLp2              ;do xx pages
BFED: 60        81            RTS
                82
BFEE: 00 00 00  83            DS    $BFFC-*
                84            ERR   *-1/$BFFC           ;Make sure we haven't overrun
                85
BFFC: 00 00     86            DA    $0000               ;RESET vector (not used)
BFFE: 80 FF     87            DA    Launch+$4000        ;BRK vector (our launch)
                88
```

CREATING AN IMAGE FROM THE TEMPLATE

In general, you can use the included TEMPLATE.32 file to easily create your own image for ROMXc. First make sure that your program consists of a single binary file that is less than $7D80 bytes (larger programs or those that load multiple files from disk can be ROMified but that is beyond the scope of this document). Find the starting address of where it loads and then if necessary pad the file to start on an even page boundary and re-save. Then do the following:

1. BLOAD the program file at $1000
2. BSAVE PROG1, A$1000, L$3E80
3. BSAVE PROG2, A$4E80, L$3F00
4. BLOAD TEMPLATE.32
5. BLOAD PROG1, A$1100
6. BLOAD PROG2, A$5100
7. If the program loads anywhere other than $800, do a CALL-151
8. Modify the location where part1 will be located: 4F90:xx (where xx00 is start address)
9. Modify the location where part2 will be located: 4FA8:xx (ramloc + $3E)
10. Modify the start address to begin execution: 4FC5:lo hi (execute at $hilo)
11. BSAVE <your great image>, A$1000, L$8000

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$1000         (Loads at $1000-$8FFF)
2. RUN INFO GEN 32                          (from the ROMX Utilities disk)
3. Enter Description and Additional Info    (Adds metadata to image)
4. BSAVE <image name>.RMX, A$1000, L$8100   (Saves complete rmx image)
```