ROMifying a Program for Uploading to ROMXce+

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 ROMXc, ROMXe, ROMXc+, and ROMX+; if you have an original ROMX for Apple II/II+, please see the document "*ROMifying a Program for Uploading to ROMX (original)"* instead. You may also wish to study that document as some of the techniques outlined there can also be applied to the later ROMX boards.

PART I – BASIC PROCESS

To work in ROMX, the code must fit into the (roughly) 16K of space from $C100-$FFFF or 32K space if you use both high and low sections of a single bank. 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 32,000 bytes (roughly 130 sectors on disk)**.

The steps to ROMify such code are then quite simple. We just need to break the code into two pieces (assuming it is over 16,000 bytes) and then combine with the loader program into a single file. The lower 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 Tutorial ROMXCE 32K.dsk file that accompanies this document has several files for converting programs to run under ROMX:

```
DISK VOLUME 254

 A 006 HELLO                    Standard Apple Language Card Loader
 B 131 TEMPLATE.32              Template Image for building 32K programs (no F8 Monitor ROM)
*B 128 CHOPLIFTER               Original CHOPLIFTER Game as downloaded from the INTERNET ARCHIVE
*B 101 RASTER BLASTER           Original RASTER BLASTER Game as downloaded from the INTERNET ARCHIVE
 A 026 INFO GEN 32              Program for adding rmx metadata to image

INTERNET ARCHIVE URL: https://archive.org
```

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 16K ROM images you create this way can be directly used with most (Apple IIe and later) 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. Although your image may also work on the ROMX+ in an Apple II/II+, it is not possible to use an Apple II/II+ emulator as the extra ROM space ($C100-$CFFF) will not be accessible.

PART II – PREPARING IMAGES FOR ROMX

All ROMX boards can use half-size (16K) or full-size (32K) image files. While you can create 16K images if that’s all you need, we’re going to assume you want as much space as possible for your application. Just be aware that there is a slight issue with creating 32K files under DOS 3.3. Normally, DOS will not let you save a file with a length of $8000 so you will need to either work in ProDOS or modify DOS to allow this. The Tutorial dsk file that accompanies this document has such a modified DOS. But if you ever find yourself trying to save a file and get the dreaded “RANGE ERROR,” don’t panic! You can instantly modify DOS to accept your command by typing either:

```
POKE 43364,255  (from the Basic),  or
A964:FF         (from the Monitor)
```

Now let’s go over the hardware implementations of the ROMX. From the *ROMXce+ API Reference* 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
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 ROMX, 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
$C100-$FEFF : Application code (Part1)

HIGH BANK
$C100-$FFFF : Application code (Part2)

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 other 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: 4FC7:90 24
9. BSAVE MY CHOPLIFTER, A$1000, L$8000 (delete CHOP1 or use another disk)

When Uploading to ROMX don't forget to add the &Sn command to the Description (where n is a valid System ROM, e.g. 1). 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 31-34). Then we JMP to our code at $BF8D to continue the Launch code from RAM.

Lines 37-49 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 41) so that we can select the upper sub-bank (line 42). Then we re-activate our DOS bank (lines 43-44) 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 51-52). Then we preset and activate that user bank (lines 53-55). Finally, lines 56-60 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  8/16/21
                    3
                    4    *           PART 1 - 0000-3FFF MAIN ROM
                    5    *           PART 2 - 4000-7FFF AUX ROM
                    6    *
                    7    *           Part 2 overwrites 1 page of part 1
                    8
                    9    * Ver 955s 8/10/21  Added ZipSlo to copy routine
                    10
                    11           DSK   TEMPIMAGE
                    12
                    13   OurBank     EQU   $0287     ;Our bank
                    14   RealBank    EQU   $02A6     ;Bank we need to run in
                    15   RAM_LOC     EQU   $0800     ;Change per program
                    16   START_LOC   EQU   $0800     ;Start of program
                    17   INTCXROMOFF EQU   $C006     ;Enable slot ROM C100-C1FF
                    18   INTCXROMON  EQU   $C007     ;Enable main ROM C100-C1FF
                    19   ZipSlo      EQU   $C0E0     ;ZIP CHIP slowdown
                    20   BANK0       EQU   $F800
                    21   TMP_LOWER   EQU   $F824
                    22   TMP_UPPER   EQU   $F825
                    23   TMP_BANK0   EQU   $F830     ;Latch temp bank#
                    24   SEL_MBANK   EQU   $F851     ;Select Main bank reg
                    25   INIT        EQU   $FB2F
                    26   SETVID      EQU   $FE93
                    27   SETKBD      EQU   $FE89
                    28
                    29           ORG   $BF80        ;Start of Image
                    30
BF80: 8D 07 C0      31   Launch   STA   INTCXROMON   ;Turn //e ROM ON
BF83: A2 FF         32            LDX   #/$FF80      ;Move our code
BF85: A9 BF         33            LDA   #$BF         ;to $BF80
BF87: 20 D6 FF      34            JSR   CopyLp+$4000 ;$FFD1
BF8A: 4C 8D BF      35            JMP   Launch2-Launch+$BF80    ;and execute it there
                    36
BF8D: A2 C1         37   Launch2  LDX   #/$C100      ;SOURCE hi-byte
BF8F: A9 08         38            LDA   #/RAM_LOC    ;TARGET hi-byte
BF91: 20 D6 BF      39            JSR   CopyLp
                    40
BF94: 20 C9 BF      41            JSR   SelBnk0      ;Enable firmware Bank 0
BF97: 2C 25 F8      42            BIT   TMP_UPPER    ;Switch sub-bank for Part 2
BF9A: AE 87 02      43            LDX   OurBank      ;Get back to our bank & copy
BF9D: BD 30 F8      44            LDA   TMP_BANK0,X
BFA0: A2 C1         45            LDX   #/$C100      ;SOURCE hi-byte
BFA2: A9 80         46            LDA   #$80         ;move 1/2 pages
BFA4: 8D D9 BF      47            STA   CopyLp+3
BFA7: A9 46         48            LDA   #/RAM_LOC+$3E00 ;TARGET hi-byte
BFA9: 20 D6 BF      49            JSR   CopyLp
                    50
BFAC: 20 C9 BF      51            JSR   SelBnk0
BFAF: AE A6 02      52            LDX   RealBank
BFB2: 29 0F         53            AND   #$0F
```

```
BFB4: BD 00 F8  54            LDA   BANK0,X      ;Prepare to launch
BFB7: 2C 51 F8  55            BIT   SEL_MBANK
BFBA: 20 2F FB  56            JSR   INIT
BFBD: 20 93 FE  57            JSR   SETVID
BFC0: 20 89 FE  58            JSR   SETKBD
BFC3: 8D 06 C0  59            STA   INTCXROMOFF  ;Turn //e ROM OFF
BFC6: 4C 00 08  60            JMP   START_LOC
                61
                62    SelBnk0
BFC9: 2C E0 C0  63            BIT   ZipSlo       ;Slow down for accelerators
BFCC: 2C CA FA  64            BIT   $FACA        ;So we don't use cached data
BFCF: 2C CA FA  65            BIT   $FACA
BFD2: 2C FE FA  66            BIT   $FAFE
BFD5: 60        67            RTS
                68
BFD6: 85 03     69    CopyLp  STA   $03
BFD8: A9 00     70            LDA   #$00
BFDA: 85 02     71            STA   $02
BFDC: A9 00     72            LDA   #$00
BFDE: 85 00     73            STA   $00
BFE0: 86 01     74    CpyLp2  STX   $01
BFE2: A0 00     75            LDY   #$00
                76
BFE4: B1 00     77    :2      LDA   ($00),Y      ;do 256 bytes
BFE6: 91 02     78            STA   ($02),Y
BFE8: C8        79            INY
BFE9: D0 F9     80            BNE   :2
BFEB: E6 03     81            INC   $03
BFED: E8        82            INX
BFEE: 2C E0 C0  83            BIT   ZipSlo       ;Keep accelerators off
BFF1: E0 00     84    SrcEnd  CPX   #/$0000      ;END OF SOURCE  hi-byte
BFF3: D0 EB     85            BNE   CpyLp2       ;do xx pages
BFF5: 60        86            RTS
                87
BFF6: 00 00 00  88            DS    $BFFC-*
BFF9: 00 00 00
                89            ERR   *-1/$BFFC    ;Make sure we haven't overrun
                90
BFFC: 00 00     91            DA    $0000        ;RESET vector (not used)
BFFE: 80 FF     92            DA    Launch+$4000 ;BRK vector (our launch)
```

CREATING AN IMAGE FROM THE TEMPLATE

In general, you can use the included TEMPLATE.32 file to easily create your own image for ROMX. 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: 4FC7: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)
```

CREATING PRODOS IMAGES FROM SYSTEM FILES

Creating images for ProDOS System applications can often be done with just three easy steps:

In ProDOS:

- BLOAD PRODOS.2.4.2 (from ROMX Apps2.po)
- BLOAD (app.SYSTEM file) , TSYS, A$1100 (Must be less than $3E00 in length)
- BSAVE (your image) , A$1000, L$8100 (keeps original RMX metadata)

This may not work for all .SYSTEM files, but it’s always worth a try!

PART III – 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 *ROMXce+ API Reference*, we know that we can enable the firmware with the following sequence:

```
BIT $FACA
BIT $FACA
BIT $FAFE
```

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

```
BIT $F80n     (n=bank to launch)
BIT $F851     (activate bank)
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. If you want to manually initialize the firmware and drop into the menu without using the countdown feature, use this:

```
JSR $DFD9     (moves ROMX code to RAM)
JSR $1012     (init ROMX code in RAM)
JMP $103C     (launch ROMX Menu 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. Most of the work is done for you by the firmware, which will slow down the CPU speed to 1 MHz when the ROMX menu is first displayed. When a bank is selected, in most cases the accelerator will be restored to its previous speed or explicitly set by the &An command. When your image code takes over there are two important things to consider. One is that the CPU may be executing at a faster (or even slower) speed than normal. And the other 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 FACA FACA FAFE sequence. While the accelerator is doing its job trying to speed things up it may see no need to actually access the "memory" location $FACA 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. This will cause the accelerator to slow down typically for about 50 mS. You can see this technique used in line 63 above. If you need more time (e.g. to move large chunks of code), you can also add this to your copy loop as shown in line 83 above.

If you really must disable and re-enable an accelerator, there are routines in the firmware that can help with this. They will be documented at a later time when the API is fixed.

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
```