Read-Only Memory (ROM) on the IMSAI 8080

By Bruce E. Hall, W8BH

Turn on an old microcomputer without ROM. It will just sit there, with no instructions to follow. It won't give you any friendly prompt. It won't do anything. You can't type anything on your keyboard, and nothing will display on the screen. Such computers are hard to use!

The ROM, or read-only memory, contains the instructions your computer executes on startup. It lets you interact with the computer in an easier and more efficient way. In the early days, the ROM held a monitor program, which accepted basic commands and showed results on an attached terminal. From here you could write your own code. More advanced ROMs contained high level languages, such as BASIC, giving the user even greater power and ease of use.

The IMSAI computers of the late 1970s came with a monitor program on ROM.

On our IMSAI 8080 emulation, there is no ROM per se; only a ROM image - a file that we put on the SD card that acts like a ROM. Instead of inserting a physical ROM chip into our real IMSAI, we install a ROM image file onto our emulator's SD card.

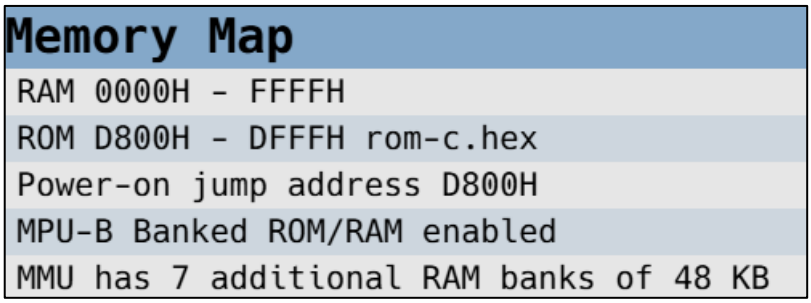
Our IMSAI comes with an assortment of ROMs. Here is a list of the current ROMs available on the Z80pack GitHub site. My kit came with 6 of these 7 ROMs preinstalled on the unit's SD card.

Filename	Use
Basic4K.hex	BASIC language – small instruction set
Basic8K.hex	BASIC language – larger instruction set
XYBasic.hex	Similar to Basic8K but easier to use.
Memon80.hex	Full-featured Monitor program
MPU-A-ROM.hex	IMSAI monitor program, version A
MPU-B-ROM.hex	IMSAI monitor program, version B
VIOFM.hex	VIO (CRT) firmware

How to change your ROM

The ROM files are stored on the SD card in the imsai/roms directory. You may study the SD card on your PC and transfer files to and from this folder. You may also put as many ROM files in this folder as you like. Once the SD card is returned to the IMSAI, you can assign ROMS to 7 different memory maps, with zero or more ROMs getting assigned to each map. This assignment is done through the "system.conf" text file, located on the SD card in the imsai/conf directory. Confused?

All of that is best illustrated with an example. Use the IMSAI web interface to see your current Memory Map. Click on the SYS icon. My system displays the information shown here.



There is 64K of RAM, and 8K of ROM. The ROM, named "rom-c.hex", is located in memory at D800-DFFF. On <Pwr On>, the system will jump to D800 to begin execution. Banked ROM means that this ROM can be disabled to expose the full 64K of RAM. Without ROM banking, the 8K of RAM at D800-DFFF is effectively "hidden" by the overlying ROM and cannot be used. Banked RAM means that additional RAM can be swapped in and out of the 64K address space. See my article on bank switching for more information.

Changing the ROM is a two-step process:

1. Assign the ROM(s) to a memory map.
2. Change your system configuration to use the memory map.

To see the list of memory maps, go to the SYS window again and click on the gear icon in the top right corner. Now click on "system.conf" and scroll down to the list of memory configurations. The first map, MEMORY 1, may look similar to this:

```
[MEMORY 1]
# Default memory configuration used in most situations with MPU-A ROM:
# 256 pages RAM, 8 pages ROM (replaces or overlays RAM)
ram         0,256
rom         0xd8,8,mpu-a-rom.hex
# Start address of the boot ROM
boot        0xd800
```

The numbers to the right of RAM and ROM refer to pages of memory, where a page is 256 bytes. The two numbers for RAM indicate the starting point (page 0, or 0000h) and its

length in pages (256 pages, or 64K). The entry for ROM, highlighted above, indicates that ROM file mpu-a-rom.hex is located on page D8 (D800h) and is 8 pages, or 2K in length.

Memory maps can be added, edited, or deleted. Be careful: an incorrect setting can render your system unresponsive. I recommend making a backup of this file (and the whole SD card, for that matter) before making changes. Often enough, the configuration you want is already present. My system came with 4 preconfigured memory maps.

Memory Map	Default Use
1	IMSAI monitor (booting CP/M)
2	IMSAI Monitor + VIO firmware
3	Memon/80 Monitor
4	XYBasic

After determining the memory map, you must tell the IMSAI to use it. This is done though the NVS Startup procedure, outlined here and here. Bits 8-10 of the address lines specify which memory configuration is used. For example, I often use Auto Run, Banked ROM, Memory Map #1, 4 MHz, Z80, and WiFi, which is **8958**:

Auto-Run	Res	Res	Res	Banked ROM	Map	Map	Map	Unlimited	4 Mhz	Undoc Instr.	Z80	WiFi STA	Log Level	Log Level	POST
1	0	0	0	**1**	0	0	**1**	0	**1**	0	**1**	**1**	0	0	0
8				**9**				**5**				**8**			

Notice that the '9' in '8958' specifies use of Memory Map #1 for CP/M. To switch to XYBASIC ROM on Memory Map #4, the NVS setting would be 8458, etc.

BASIC 4K ROM.

There aren't any predefined memory maps for the Basic4k and Basic8K ROMs. Where are they supposed to be located, and where do they boot? Look at the first and last lines of the Basic4K ROM hex file, shown here:

```
:0800000002100143EBAC34900BF
:200FD8004D008601005354455000544845 4E0020455252204020FE0010000002000000003
```

The addresses, 0000h and 0FD8h, are highlighted. This hex file will load starting at address 0000, and continue roughly through 0FFF, filling the expected 4K of space. The following configuration should do the trick:

```
[MEMORY 5]
# Memory configuration for running 4K Basic at 0000
# 4K ROM, 60K RAM
rom         0,16,basic4k.hex
ram         16,240
boot        0x0000
```

The ROM entry indicates a starting address of 0000 and length of 16 pages (4K). The RAM entry indicates a starting address of 1000h (16x256=1000h) and length of 240 pages (60K). An NVS setting of **0558** or **8558** will boot this ROM in memory map #5.

The object code includes no copyright information, but this appears to be IMSAI's version of 4K BASIC. Its user interface is minimal. There is no welcome message. There is no command prompt. Listed line numbers are 4 digits. Set CAPS LOCK on, because this BASIC does not understand lower case letters. It supports ABS, DATA, INT, RND, READ, REM, INPUT, LIS, NEW, RUN, GOTO, GOSUB, PRINT, RET, SGN, SQR, STEP, STOP, LET, TO, FOR, NEXT, plus basic math and relational operators. It does not support trig functions, strings ("A$"), or arrays "LET A(3)=5". Variable names can be a letter, or a letter followed by a one-digit number (A2 is OK; A22 and AC are not). The backspace key does not work (is ignored) while entering the current line.

Make sure to put your system in 8080 mode for this BASIC. Some math functions, including division and SQR, return erroneous results in Z80 mode.

This isn't ALTAIR Basic, since it is missing some functions such as DIM, TAB, and SIN.

The commented source code and listing are available online. Thank you, Udo Munk!

BASIC 8K ROM

Here is a memory map for running the 8K ROM:

```
[MEMORY 5]
# Memory configuration for running 8K Basic at 0000
# 8K ROM,56K RAM
rom         0,32,basic8k.hex
ram         32,224
boot        0x0000
```

The BASIC 8K ROM is Copyright 1977 by IMSAI. Its welcome message, "8K VER 1.4", and ':' prompt are a friendly update from the 4K version. Division and square root now work. BASIC 8K 1.4 adds the following functions: SIN, COS, TAN, ATN, LN, LOG, EXP, PI, POS, PEEK, POKE, INP, OUT, CHR$, STRING$, SPACE$, NUM$, LEFT$, RIGHT$, MID$, INSTR, DEF FN, CALL, LEN, POS, TAB, PI, LINE, TAPE, SAVE, CON, VAL, RESTORE, END, RANDOMIZE. Line editing is improved but imperfect. You still need to set CAPS LOCK for this BASIC.

MPU-A-ROM: what is it, and how does it differ from MPU-B-ROM?

Very interesting questions! IMSAI's first CPU board, called the MPU-A board, used the Intel 8080 microprocessor. The next version, which used the Intel 8085 processor, was called the MPU-B board. One might assume that MPU-A-ROM was the ROM file for the MPU-A board, and MPU-B-ROM was the ROM file for the MPU-B board. This is incorrect. There was, in fact, *no ROM* on the MPU-A board.

Original documentation for these two ROM files cannot be found. Curious, I spent a few weeks hand-disassembling and commenting the ROM images, which I have made available on GitHub. The results are interesting.

Feature	MPU-A-ROM	MPU-B-ROM
Code Size	1864 bytes	2045 bytes
Number of monitor commands	23	29
Contains orphaned code	Yes	No
Cassette commands: A, G, L, X	No*	Yes*
Memory protect commands: P, U	No*	Yes*
DIO (disk controller) support	Yes (E000 & F000)*	Yes (E000 only)*
Undocumented commands; :D, :I, :O	Yes	Yes
Memory commands: C,D,E,F,H,J,K,M,S,T,V	Yes	Yes
Port Commands: I, O	Yes*	Yes*
Disk Commands: B, N, R, W	Yes	Yes
Timer support/ Baud Rate/ Z command	Yes*	Yes*
MMU support	Yes	Yes
Parallel port (keyboard) input	Yes*	Yes*
VIO firmware support/ Q command	Yes	Yes
Boot to Disk OS (CP/M, etc)	Yes	Yes

(*) indicates that the feature does not work on an emulated IMSAI 8080.

Of the two, ROM "A" is smaller and contains less commands. It appears to have been created *after* ROM "B", since working code in B is orphaned in A. B's cassette functions are replaced with increased disk controller support in A.

Which one is better, A or B? They are *functionally equivalent* in our IMSAI 8080 system, since none of B's unique commands are supported in the emulated environment. Feel free to use either one.

What does MPU-A-ROM do?

We use this ROM every time we boot to CP/M or some other disk OS. From the table above it can do a lot, and yet we don't notice it. Why not?

MPU-A-ROM's first job is to read the first sector of Disk A into memory. If this is successful, it silently transfers control to the sector just read. On a system disk, the sector contains bootloader code to read in the rest of the operating system. CP/M is copied from disk and it starts. Booting to CP/M happens silently with no user notification.

However, without a disk in Drive A, you will notice something different if you press <Ext Clr> on the front panel and then press the space bar. The MPU-A-ROM will display "IMSAI IEEE MONITOR VERS 1.0" and a "?" monitor prompt.

Here is a complete list of commands found in MPU-A-ROM. I cannot find documentation of this list elsewhere. The list of commands in the VDP-80 VIO-D user guide is not a full list. All numerical values are in hexadecimal and all letters must be in uppercase. Any parameters must be separated by a single space. An address like 000D may be entered as D, 0D, 00D, or 000D.

IMSAI MPU-A-ROM MONITOR COMMANDS

Command	**Description**
Boot from A: drive	B
Call subroutine	C <addr> Call a subroutine at <addr>, returning control to the monitor on a return instruction
Display Memory	D <start addr> <end addr> Displays contents of memory block from <start> to <end>
Examine Memory	E <addr> Displays contents at address & allows entry of new data: • Press <space> to go to next location • Press <-> to display previous location • Press <enter> to quit • Enter two hex digits to modify memory
Fill Memory	F <start addr> <end addr> <fill byte> Fill memory block start..end with the fill byte
Read tape	H

	Reads characters console in Intel HEX format. To use it, copy a hex file to the clipboard, then paste as text (Ctrl-Shift-V) to TTY.
Input Port	I <port #> Executes an input port command and returns the result. This command does not work properly on the current system.
Jump to Address	J <addr> Transfers control to the specified address. The ROM remains in place at D800-DFFF. See command K below.
Kill ROM & Jump	K <addr> Removes ROM from the address space at D800-DFFF, then jumps to the specified location
Move Memory	M <start addr> <end addr> <dest addr> Copy memory block start..end to the destination address
New Descriptor	N <addr> This undocumented command changes the address of the disk descriptor.
Output Port	O <port> <value> Executes an OUT <port> command with the specified value. This command does not work properly on the current system.
Quit to Video	Q Switches off the monitor ROM and transfers execution to the VIO firmware. No action if the VIO firmware is not present.
Read from Disk	R <tt> <ss> <addr> <unit> Example: "R 00 01 0100 2" will read disk #2, track 0, sector 1 to memory address 0100h
Search Memory	S <start> <end> <2 byte pattern> Search Memory from start to end for a 2-byte pattern. For example, "S 0 200 02DB will search 0000-0200h for the two bytes 02 DB.
Test Memory	T <start addr> <end addr> Tests a memory block by writing and reading all 255 bit patterns. Memory contents are NOT destroyed. A silent return to the monitor prompt = no errors found. If T is entered without start or end address, it will search all memory until a bad location (or ROM) is encountered. The first bad memory (or ROM) encountered will be displayed as: <addr> <byte found> <byte expected>
Verify Memory	V <start addr> <end addr> <dest addr>

	Verifies that two blocks of memory are the same. It compares one block of memory (start..end) against another block at (destF) and prints the address/values of any discrepancies.
Write to Disk	W <tt> <ss> <addr> <unit> Example: "W 00 01 0100 2" will write to disk #2, track 0, sector 1 from memory address 0100h. Dangerous command!!
Change Direct I/O address	Y <addr> Moves the location where the INPUT/OUTPUT port commands are executed.
Change Baud Rate	Z <baud> Changes the serial port baud rate. This command does not work on our emulated system (wrong port, no timer support).
Hidden "programmer" commands	:D = change the Floppy Controller Type :I = change the Input Device list :O = change the Output Device list

As mentioned above, MPU-B-ROM contains additional commands, but none of these are supported on the emulated system.

Making a custom ROM

The rest of this article will discuss how to create your own custom ROM. Stop here if that is not your cup of tea. Continue reading if you want to personalize what your system does on startup. Do you want a friendly greeting? Or maybe even a password challenge? It's up to you.

The idea first occurred to me after watching a great video from the ShadowTron Blog. He creates a ROM containing small programs that can be executed from the front panel. He even uses my assembler of choice. Nerdy and cool. It made me wish for a more user-friendly ROM that can communicate via keyboard and screen.

Writing ROM code is different than writing code for an operating system like CP/M. You must create bare-metal routines which interact directly with the hardware (or emulated hardware, in our case). Finally, there is no place to "return" to and no operating system to fall back on. We assume only that there is a serial port connected to a console.

Let's start small, with a simple ROM "hello, word" program. The code to initialize and write to the serial port UART is borrowed from UART.ASM in my introductory article on the IMSAI 8080.

The ROM code, on my GitHub page as rom0.asm, is a single routine:

```
.ORG    0D800H

start:      LD      A,0CAh      ; 7 BITS, NO PARITY, 2 STOP
            OUT     (3),A       ; UART A - SET MODE
            LD      A,027H      ; ENABLE TX & RX
            OUT     (3),A       ; UART A - SET CONTROL
            LD      HL,str

loop:       LD      A,(HL)      ; load next char in string
            OR      A           ; is it end-of-string NULL?
            JP      Z,stop      ; if so, we are done
            OUT     (2),A       ; send char to serial port
            INC     HL          ; point to next character in string
            JP      loop        ; and loop until done

stop:       JP      stop        ; loop here until reset

str:        .db "IMSAI 8080 Custom ROM",0
```

Note the D800h origin address. This was chosen for historical reasons, as the ROM in a real MPU-B board sits at this address. D800h is the starting address of both IMSAI ROM images, A and B. The board – and our emulator – allow us to remove the ROM from the address space under software control.

The first section of code initializes the serial port UART. The second section, at loop, sends our message to serial port 2, one character at a time, until the entire message has been transmitted. Finally, the jump statement at *stop* is an infinite loop. The program will stay here until we stop the processor via the front panel. An infinite loop is preferable to a HLT instruction, which would shut down the entire emulation environment.

Compile the code on the TASM assembler "TASM -80 rom0.asm" to produce an Intel hex file:. The -80 option specifies the use of Z80 mnemonics. Change the name of the object file from rom0.obj to rom0.hex. This file will be installed on the IMSAI below.

Next, we must create a memory map for this rom image. The first four or five maps are predefined in the system, so we will create map #6. Add the following lines to system.conf on your SD card. You can edit the file via the web user interface.

```
[MEMORY 6]
# Memory configuration for running 2K custom ROM at D800
# 256 pages RAM, 8 pages ROM (replaces or overlays RAM)
ram         0,256
rom         0xd8,8,rom0.hex
boot        0xd800
```

Change your NVS startup configuration to **0658** in order to activate this memory map.

Finally, install the rom0.hex file onto your IMSAI 8080:

- Remove power.
- Remove the system's micro-SD card
- Insert the card into your PC.
- Put rom0.hex into the /roms folder on the card
- Reinstall the micro-SD card on the IMSAI.
- Reapply power to the IMSAI.

Once powered up, flip the front panel <Pwr On> switch. The WAIT LED will light, and the address bus should show the boot address of D800h. If you are using the web GUI, click on the TTY: icon to open its window. Now toggle the front panel <run> switch to see your custom ROM message.

Micro-SD card

Swapping the small, fragile micro SD card in and out of your system can be fiddly and tedious. Consider using an extender cable for easier access.

About those old floppy drive controllers...

A basic monitor function checks for a floppy disk and, if found, starts the disk operating system. We must first know about the IMSAI's floppy drive controllers. Several different types were available by the late 1970s.

The first floppy disk controller, used by IMSAI in 1976, consisted of two printed circuit boards: the FIB (floppy interface board) and the IFM (interface master board). Together they were sold as the "FIF Disk System". The controller was paired with a CalComp floppy drive. It used 8" IBM-formatted single-sided, single-density floppy (SSSD) disks.

There was a follow-up controller in 1977, the DIO/PDS controller, that worked with 8-inch standard and 5.25-inch "mini" disks, in both single and double density. Although MPU-A-ROM and MPU-B-ROM work with both controller types, only the FIF controller is supported in our emulation environment.

To use a disk under CP/M, we call on handy BDOS routines to do the heavy hitting. But without CP/M we must write code directly to the floppy drive controller.

The FIF controller is accessed in software through I/O port F3. The FIF supports at least two "byte commands" sent to this port:

COMMAND	DESCRIPTION
00h	Execute a 7-byte command string, called the “disk descriptor”
10h	Specify the disk descriptor location, with the next two bytes indicating the LSB and MSB of the descriptor address.

The disk descriptor contains all the information needed to execute a disk operation: a disk operation (read or write), the disk location (track, sector), and the memory address where data is to be read from/written to. The controller takes control of system memory from the CPU through direct memory access (DMA). Confusingly, the address where this action takes place is also referred to as the DMA. Here is the structure of the 7-byte disk descriptor:

Byte Offset	Function
0	Command in upper 4 bits, Drive in lower 4 bits: 10h = Write, 20h = Read; Drive A = 01, Drive B = 02, Drive C = 04.
1	Controller result code, 01 = command successful
2	Track# MSB, set to 0.
3	Track #
4	Sector #
5	Direct Memory LSB, set to 00h
6	Direct Memory MSB, set to 00h

The first byte in the disk descriptor contains both the command and the drive for which the command is meant. The drives are enumerated 0 through 3, which we typically refer to as Drives A through D. Set bit 0 in the command/unit byte to operate on Drive A, set bit 1 for Drive B, and so on. The two basic commands are 10h for Write and 20h for Read. Add the command and drive together. Writing to Drive A = command 10h + drive A (01h) = 11h. Reading from Drive A = command 20h + Drive A (01h) = 21h:

(unused)	(unused)	Command	Command	Drive D	Drive C	Drive B	Drive A
0	0	**1**	0	0	0	0	**1**
2				**1**			

The track and sector numbers go in bytes 3 and 4, respectively. The address which is read from/written to is given in bytes 5 and 6. One sector (128 bytes) is written at a time.

What is the command string to read Track 5, Sector 1 from Drive A to address 0000? Answer: 21 00 00 05 01 00 00. Byte 21h is the command “Read from Drive A”. The controller will return the result of this operation in the second byte, with 01 = success.

What is the command string to write to Track 6, Sector 2 of Drive B with data at address 2000h? Answer: 12 00 00 06 02 00 20.

Reading a Floppy Disk

This code is available on GitHub as rom1.asm:

```
DDA         .EQU    0080h       ; disk descriptor address
FDC         .EQU    0FDh        ; Floppy disk controller (FIF) port
LED         .EQU    0FFh        ; front panel LED port

            .ORG    0D800H

readFD:     LD      HL,cmdStr   ; source = command string
            LD      DE,DDA      ; destination = 0080h
            LD      BC,7        ; 7 bytes to copy
            LDIR                ; do the copy, Z80 style

            ; the disk descriptor is now initialized with a command
            ; to read Drive 0/Track 0/Sector 1 to Address 0000.
            ; Now inform the controller where the command is located.

            LD      A,10H       ; cmd 10h = set disk descriptor
            OUT     (FDC),A     ; sent it
            LD      HL,DDA
            LD      A,L
            OUT     (FDC),A     ; send LSB of disk descriptor
            LD      A,H
            OUT     (FDC),A     ; send MSB of disk descriptor

            ; Tell the FDC to execute the command string

            XOR     A           ; cmd 0 = execute the command string
            OUT     (FDC),A     ; send it

            ; Poll the result byte & return when response is received

            LD      HL,DDA+1    ; point to result code
loop:       LD      A,(HL)      ; get result
            OR      A           ; still zero = no response?
            JP      Z,loop      ; wait until response received

            ; Show controller response code on the front panel LEDs

            CPL                 ; complement A for LED output
            OUT     (LED),A     ; output code to front panel LEDs
stop:       JP      stop

cmdStr:     .db 21h,00,00,00,01,00,00h
```

Briefly, there are 4 steps to read the disk, each highlighted above: (1) set up a command string, (2) tell the controller where to find the command string, (3) execute the command, and get the result code.

Run the code without a disk in Drive A. Go to address D800 (you remember how to do that, don't you?) and toggle the <run> switch. The output LEDs should show the controller result code A1, which means "No disk in Drive A". Now, put a CP/M system disk in Drive A, return to address D800 and run it again. The LEDs should now display 01, which means "No errors reading disk in Drive A"

Code 01 also means that Track 0, Sector 1 was transferred to system memory at 0000. We should be able to see it. From the front panel, examine 0000 and <examine next> through the first few bytes. You should see the following bytes:

C3 0A 00 21 00 00 00 02 00 BC 31 FF

Those bytes aren't random; they are the CP/M bootloader which the ROM just retrieved from disk. Rerun your ROM with a non-system disk, such as "Zork", and notice the difference.

Booting a System Disk

Once the CP/M bootloader is loaded into system memory, executing it (and therefore loading and executing CP/M) is very easy! A single jump statement to 0000h is all it takes. The bootloader code takes over from there. Check out rom2.asm for a full listing.

ROM Banking

The IMSAI MPU-B board contains a 2K ROM at address D800h. This ROM is enabled at startup but can be "removed" from the address space by writing to a specific I/O port. This memory control functionality is an option in our emulated environment, controlled by the "NVS_BANK_ROM" bit in the system configuration. See my bank switching article for more details.

If you set the NVS_BANK_ROM bit to 1, by changing the system configuration from 0658 to 0E58, rom2.hex will fail. The 0E58 setting causes ROM to get mapped to D800 *and* 0000. Our code uses the first memory page at 0000 to receive data from the disk. Anything written there is obscured by the overlying ROM. Bottom line: if ROM banking is active, we need to turn off ROM mapping at 0000 to use the RAM underneath.

ROM mapping at 0000 is turned off by storing the value 40h to port F3h. Add this code and the ROM will work - regardless of the NVS_BANK_ROM bit setting. As a bonus, if you store value C0 to port F3h at the end of the code, ROM mapping to D800 is also eliminated, freeing up the 2K of RAM underneath. Take care not to turn off the ROM until the ROM code

is finished. If you do, your code will suddenly disappear from the address space, likely crashing the system. You have just enough time to execute a jump instruction:

```
LD   A,40h            ; 40h: 0000 RAM, D800 ROM
OUT  (0F3h),A         ; change memory configuration
;
; put your ROM code HERE
;
LD   A,0C0h           ; C0h: 0000 RAM, D800 RAM
OUT  (0F3h),A         ; change memory configuration
JP   0000             ; ROM at D800 is vanishing, jump away!
```

See rom3.asm for implementation details.

One last detail: rom3.asm works when manually started at address D800h, but it fails on Auto-Run. Why? Because ROM execution begins at 0000 *(see Appendix for more details)*. Once the code turns off ROM in lower memory, the code disappears! The CPU attempts to run whatever random bytes are in RAM. Not good. A simple fix is to add a jump statement at the beginning of the code:

```
start:  LD      A,40H
        OUT     (MCP),A         ; turn off ROM at 0000-07FF
        JP      readFD          ; allow auto-run start @ 0000.
readFD: LD      HL,cmdStr       ; ROM code continues…
```

This jump statement does nothing if the program is running at D800 – it simply advances the program counter one byte from D806 to D807. But if the program began at 0000, the jump bumps the program counter from 0006 to D807, where the code remains intact. Neat! Auto-run now works. See rom4.asm (no error message) and rom5.asm (error message if no disk) for full listing. Either file can be used as an MPU-A-ROM substitute.

Creating your own ROM monitor

Before rolling your own, check out the excellent, feature-packed “Memon/80”, which is available on ROM. It was preinstalled as Memory Map 3 on my system. Memon/80 likely has all the monitor functions you want. I took up the challenge of making my own monitor because I wanted something simpler than Memon/80 and friendlier than MPU-A-ROM.

Here is a simple monitor algorithm:

1. Read the CP/M bootloader (Track 0, Sector 1) from Drive A.
2. If successful, jump to 0000 to run it.
3. Otherwise, wait for user commands & execute them.
4. If the entered command is “Boot to Disk”, go back to step 1.

Items 1 and 2 are already done. What remains is accepting & executing user commands. Those commands are entirely up to you.

My custom ROM is at rom-c.asm. I hope you are inspired to create your own custom ROM.

Happy Hacking!

Bruce.

Last updated: 16 Aug 2025

Other articles in this series: IMSAI 8080 articles

APPENDIX: Effects of Banked ROM and Auto-Run settings on ROM startup address

I did an experiment to see how the Banked ROM and Auto-Run settings affect the ROM startup address. To do this I wrote a program (rom6.asm) that does 3 things:

1. Shows the program counter just after startup, called Addr1
2. Executes a local jump statement
3. Shows the program counter right after the jump, called Addr2

The program was installed as the system ROM with boot address of D800 and was run under 4 different configurations:

NVS setting	Auto-Run?	Banking?	Boot Address:	Addr1:	Addr2:
0658	No	No	D800	D80F	D822
0E58	No	Yes	0000	000F	D822
8658	Yes	No	D800, not visible	D80F	D822
8E58	Yes	Yes	0000, not visible	000F	D822

The first two configurations require the <run> switch to be toggled before they execute. I recorded their boot address as shown by the Address front panel lights. The boot address for the auto-run configurations cannot be determined in the same way.

Notice that, in all cases, ROM ran in the D800 page after the jump statement. However, the ROM starting address varied between configurations. Those without banked ROM started at D800, but those with banked ROM started at 0000 (highlighted above).

Conclusion: **ROM Banking causes ROM execution to begin at 0000, regardless of the "boot address" setting in the memory map.** Auto-Run has no direct effect on startup address. If Auto-Run is off, the user has the option to set the starting address manually to D800 and all is well. But when Auto-Run and ROM-banking are on, the 0000 starting address cannot be manually overridden. Turning off ROM at 0000-07FF will then cause the ROM code to "disappear" during execution. A single jump statement solves the problem, since it always transfers execution to the D800 page.