



---

# RomWBW Architecture

---

RetroBrew Computing  
RomWBW Version 2.9.1

---

June 3, 2019

---

## Contents

Overview .....	2
Background .....	2
General Design Strategy.....	3
Runtime Memory Layout .....	4
System Boot Process.....	4
ROM Boot.....	5
Application Boot .....	6
Notes.....	6
Driver Model .....	6
Character / Emulation / Video Services .....	7
HBIOS Reference .....	8
Invocation .....	8
Function Overview .....	9
Character Input/Output (CIO).....	11
Disk Input/Output (DIO).....	13
Real Time Clock (RTC) .....	18
Video Display Adapter (VDA) .....	20
System (SYS).....	27

## Overview

RomWBW provides a complete firmware package for all of the Z80 and Z180 based systems that are available in the RetroBrew Computers Community (see <http://www.retrobrewcomputers.org>) as well as support for the RC2014 platform. Each of these systems provides for a fairly large ROM memory (typically, 512KB or more). RomWBW allows you to configure and build appropriate contents for such a ROM.

Typically, a computer will contain a small ROM that contains the BIOS (Basic Input/Output System) functions as well as code to start the system by booting an operating system from a disk. Since the RetroBrew Computers Projects provide a large ROM space, RomWBW provides a much more comprehensive software package. In fact, it is entirely possible to run a fully functioning RetroBrew Computers System with nothing but the ROM.

RomWBW firmware includes:

- System startup code (bootstrap)
- A basic system/debug monitor
- HBIOS (Hardware BIOS) providing support for the vast majority of RetroBrew Computers I/O components
- A complete operating system (either CP/M 2.2 or ZSDOS 1.1)
- A built-in CP/M filesystem containing the basic applications and utilities for the operating system and hardware being used

It is appropriate to note that much of the code and components that make up a complete RomWBW package are derived from pre-existing work. Most notably, the imbedded operating system is simply a ROM-based copy of generic CP/M or ZSDOS. Much of the hardware support code was originally produced by other members of the RetroBrew Computers Community.

The remainder of this document will focus on the HBIOS portion of the ROM. HBIOS contains the vast majority of the custom-developed code for the RetroBrew Computers hardware platforms. It provides a formal, structured interface that allows the operating system to be hosted with relative ease.

## Background

The Z80 CPU architecture has a limited, 64K address range. In general, this address space must accommodate a running application, disk operating system, and hardware support code.

All RetroBrew Computers Z80 CPU platforms provide a physical address space that is much larger than the CPU address space (typically 512K or 1MB physical RAM). This additional memory can be made available to the CPU using a technique called bank switching. To achieve this, the physical memory is divided up into chunks (banks) of 32K each. A designated area of the CPU's 64K address space is then

reserved to “map” any of the physical memory chunks. You can think of this as a window that can be adjusted to view portions of the physical memory in 32K blocks. In the case of RetroBrew Computers platforms, the lower 32K of the CPU address space is used for this purpose (the window). The upper 32K of CPU address space is assigned a fixed 32K area of physical memory that never changes. The lower 32K can be “mapped” on the fly to any of the 32K banks of physical memory at a time. The only constraint is that the CPU cannot be executing code in the lower 32K of CPU address space at the time that a bank switch is performed.

By cleverly utilizing the pages of physical RAM for specific purposes and swapping in the correct page when needed, it is possible to utilize substantially more than 64K of RAM. Because the RetroBrew Computers Project has now produced a very large variety of hardware, it has become extremely important to implement a bank switched solution to accommodate the maximum range of hardware devices and desired functionality.

## General Design Strategy

The design goal is to locate as much of the hardware dependent code as possible out of normal 64KB CP/M address space and into a bank switched area of memory. A very small code shim (proxy) is located in the top 512 bytes of CPU memory. This proxy is responsible for redirecting all hardware BIOS (HBIOS) calls by swapping the “driver code” bank of physical RAM into the lower 32K and completing the request. The operating system is unaware this has occurred. As control is returned to the operating system, the lower 32KB of memory is switched back to the original memory bank.

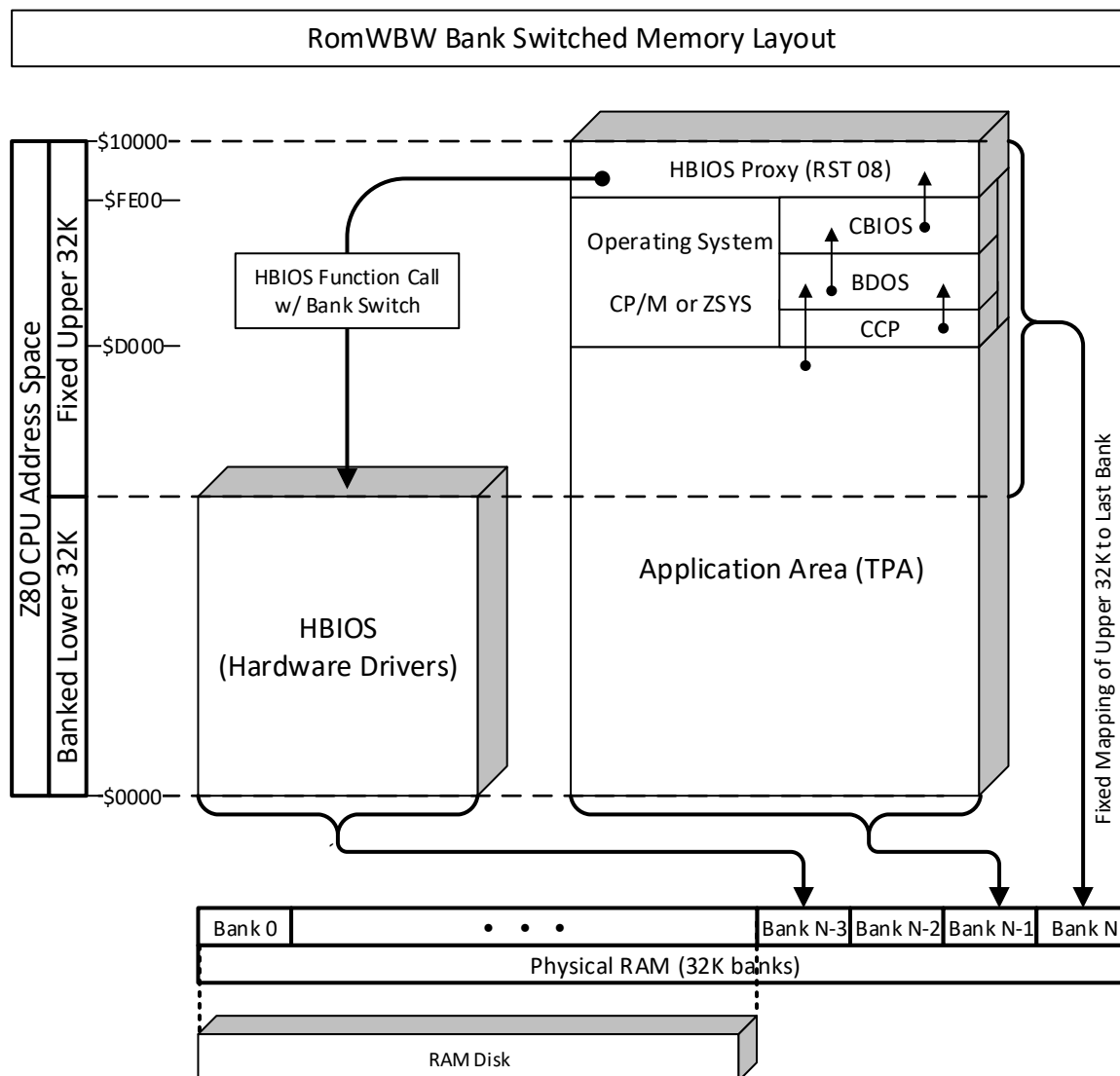
HBIOS is completely agnostic with respect to the operating system (it does not know or care what operating system is using it). The operating system makes simple calls to HBIOS to access any desired hardware functions. Since the HBIOS proxy occupies only 512 bytes at the top of memory, the vast majority of the CPU memory is available to the operating system and the running application. As far as the operating system is concerned, all of the hardware driver code has been magically implemented inside of a small 512 byte area at the top of the CPU address space.

Unlike some other Z80 bank switching schemes, there is no attempt to build bank switching into the operating system itself. This is intentional so as to ensure that any operating system can easily be adapted without requiring invasive modifications to the operating system itself. This also keeps the complexity of memory management completely away from the operating system and applications.

There are some operating systems that have built-in support for bank switching (e.g., CP/M 3). These operating systems are allowed to make use of the bank switched memory and are compatible with HBIOS. However, it is necessary that the customization of these operating systems take into account the banks of memory used by HBIOS and not attempt to use those specific banks.

Note that all code and data are located in RAM memory during normal execution. While it is possible to use ROM memory to run code, it would require that more upper memory be reserved for data storage. It is simpler and more memory efficient to keep everything in RAM. At startup (boot) all required code is copied to RAM for subsequent execution.

## Runtime Memory Layout



## System Boot Process

A multi-phase boot strategy is employed. This is necessary because at cold start, the CPU is executing code from ROM in lower memory which is the same area that is bank switched.

Boot Phase 1 copies the phase 2 code to upper memory and jumps to it to continue the boot process. This is required because the CPU starts at address \$0000 in low memory. However, low memory is used as the area for switching ROM/RAM banks in and out. Therefore, it is necessary to relocate execution to high memory in order to initialize the RAM memory banks.

Boot Phase 2 manages the setup of the RAM page banks for HBIOS operation, performs hardware initialization, and then executes the boot loader.

Boot Phase 3 is the loading of the selecting operating system (or debug monitor) by the Boot Loader. The Boot Loader is responsible for prompting the user to select a target operating system to load, loading it into RAM, then transferring control to it. The Boot Loader is capable of loading a target operating system from a variety of locations including disk drives and ROM.

Note that the entire boot process is entirely operating system agnostic. It is unaware of the operating system being loaded. The Boot Loader prompts the user for the location of the binary image to load, but does not know anything about what is being loaded (the image is usually an operating system, but could be any executable code image). Once the Boot Loader has loaded the image at the selected location, it will transfer control to it. Assuming the typical situation where the image was an operating system, the loaded operating system will then perform it's own initialization and begin normal operation.

There are actually two ways to perform a system boot. The first, and most commonly used, method is a "ROM Boot". This refers to booting the system directly from the startup code contained on the physical ROM chip. A ROM Boot is always performed upon power up or when a hardware reset is performed.

Once the system is running (operating system loaded), it is possible to reboot the system from a system image contained on the file system. This is referred to as an "Application Boot". This mechanism allows a temporary copy of the system to be uploaded and stored on the file system of an already running system and then used to boot the system. This boot technique is useful to: 1) test a new build of a system image before programming it to the ROM; or 2) easily switch between system images on the fly.

A more detailed explanation of these two boot processes is presented below.

## ROM Boot

At power on (or hardware reset), ROM page 0 is automatically mapped to lower memory by hardware level system initialization. Page Zero (first 256 bytes of the CPU address space) is reserved to contain dispatching instructions for interrupt instructions. Address \$0000 performs a jump to the start of the phase 1 code so that this first page can be reserved.

The phase 1 code now copies the phase 2 code from lower memory to upper memory and jumps to it. The phase 2 code now initializes the HBIOS by copying the ROM resident HBIOS from ROM to RAM. It subsequently calls the HBIOS initialization routine. Finally, it starts the Boot Loader which prompts the user for the location of the target system image to execute.

Once the boot loader transfers control to the target system image, all of the Phase 1, Phase 2, and Boot Loader code is abandoned and the space it occupied is normally overwritten by the operating system.

## Application Boot

When a new system image is built, one of the output files produced is an actual CP/M application (an executable .COM program file). Once you have a running CP/M (or compatible) system, you can upload/copy this application file to the filesystem. By executing this file, you will initiate an Application Boot using the system image contained in the application file itself.

Upon execution, the Application Boot program is loaded into memory by the previously running operating system starting at \$0100. Note that program image contains a copy of the HBIOS to be installed and run. Once the Application Boot program is loaded by the previous operating system, control is passed to it and it performs a system initialization similar to the ROM Boot, but using the image loaded in RAM.

Specifically, the code at \$0100 (in low memory) copies phase 2 boot code to upper memory and transfers control to it. The phase 2 boot code copies the HBIOS image from application RAM to RAM, then calls the HBIOS initialization routine. At this point, the prior HBIOS code has been discarded and overwritten. Finally, the Boot Loader is invoked just like a ROM Boot.

## Notes

1. Size of ROM disk and RAM disk will be decreased as needed to accommodate RAM and ROM memory bank usage for the banked BIOS.
2. There is no support for interrupt driven drivers at this time. Such support should be possible in a variety of ways, but none are yet implemented.

## Driver Model

The framework code for bank switching also allows hardware drivers to be implemented mostly without concern for memory management. Drivers are coded to simply implement the HBIOS functions appropriate for the type of hardware being supported. When the driver code gets control, it has already been mapped to the CPU address space and simply performs the requested function based on parameters passed in registers. Upon return, the bank switching framework takes care of restoring the original memory layout expected by the operating system and application.

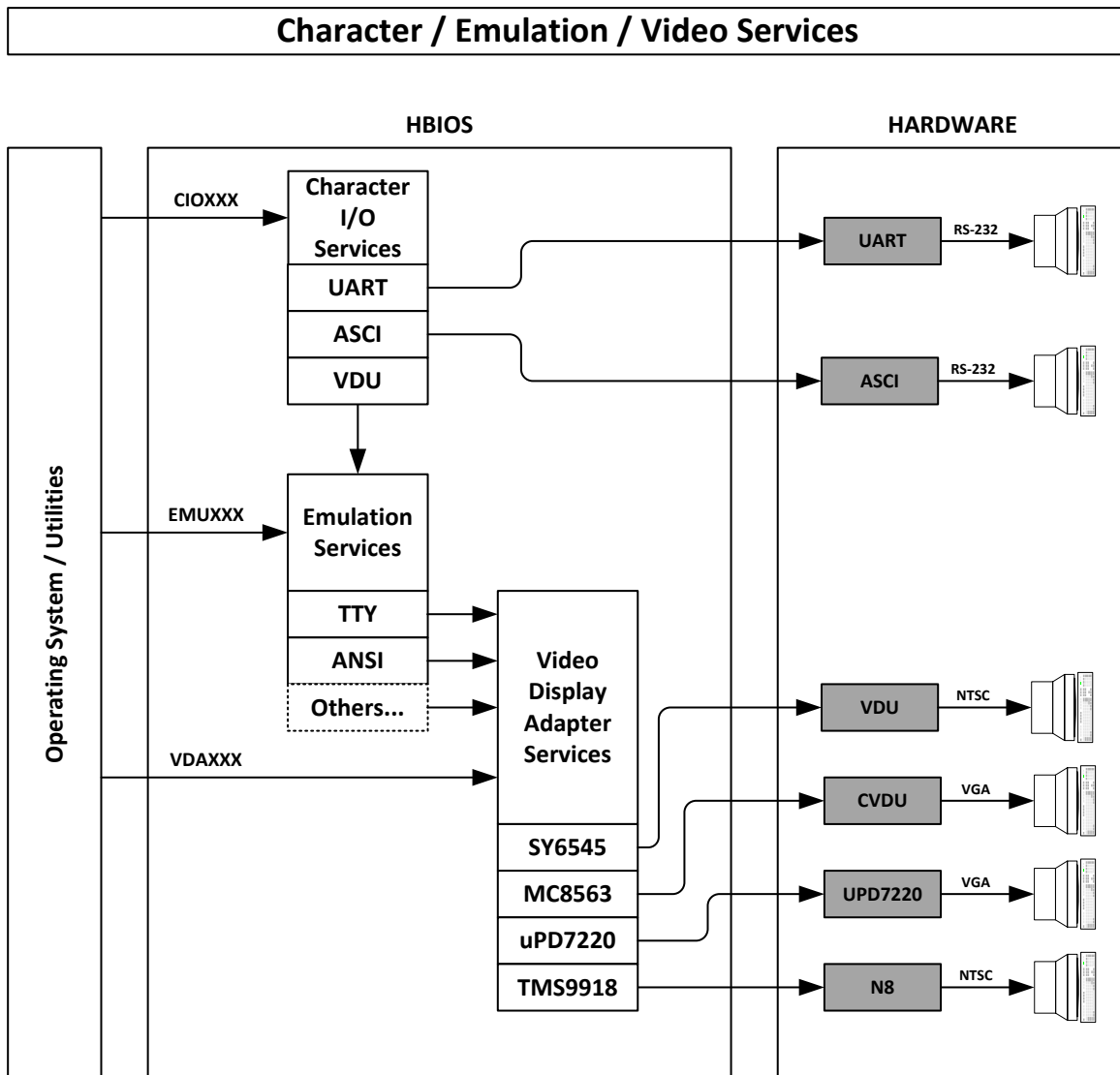
However, the one constraint of hardware drivers is that any data buffers that are to be returned to the operating system or applications must be allocated in high memory. Buffers inside of the driver's memory bank will be swapped out of the CPU address space when control is returned to the operating system.

If the driver code must make calls to other code, drivers, or utilities in the driver bank, it must make those calls directly (it must not use RST 08). This is to avoid a nested bank switch which is not supported at this time.

## Character / Emulation / Video Services

In addition to a generic set of routines to handle typical character input/output, HBIOS also includes functionality for managing built-in video display adapters. To start with there is a basic set of character input/output functions, the CIOXXX functions, which allow for simple character data streams. These functions fully encompass routing byte stream data to/from serial ports. Note that there is a special character pseudo-device called "CRT". When characters are read/written to/from the CRT character device, the data is actually passed to a built-in terminal emulator which, in turn, utilizes a set of VDA (Video Display Adapter) functions (such as cursor positioning, scrolling, etc.).

The following diagram depicts the relationship between these components of HBIOS video processing:



Normally, the operating system will simply utilize the CIOXXX functions to send and receive character data. The Character I/O Services will route I/O requests to the specified physical device which is most frequently a serial port (such as UART or ASCI). As shown above, if the CRT device is targeted by a



CIOXXX function, it will actually be routed to the Emulation Services which implement TTY, ANSI, etc. escape sequences. The Emulation Services subsequently rely on the Video Display Adapter Services as an additional layer of abstraction. This allows the emulation code to be completely unaware of the actual physical device (device independent). Video Display Adapter (VDA) Services contains drivers as needed to handle the available physical video adapters.

Note that the Emulation and VDA Services API functions are available to be called directly. Doing so must be done carefully so as to not corrupt the “state” of the emulation logic.

Before invoking CIOXXX functions targeting the CRT device, it is necessary that the underlying layers (Emulation and VDA) be properly initialized. The Emulation Services must be initialized to specify the desired emulation and specific physical VDA device to target. Likewise, the VDA Services may need to be initialized to put the specific video hardware into the proper mode, etc.

## HBIOS Reference

### Invocation

HBIOS functions are invoked by placing the required parameters in CPU registers and executing an RST 08 instruction. Note that HBIOS does not preserve register values that are unused. However, it must not modify the Z80 alternate registers or IX/IY (these registers can be used within HBIOS as long as they are saved and restored internally).

Normally, applications will not call HBIOS functions directly. It is intended that the operating system makes all HBIOS function calls. Applications that are considered system utilities may use HBIOS, but must be careful not to modify the operating environment in any way that the operating system does not expect.

In general, the desired function is placed in the B register. Register C is frequently used to specify a subfunction or a target device number. Additional registers are used as defined by the specific function. Register A should be used to return function result information. A=0 should indicate success, other values are function specific.

Some functions utilize pointers to memory buffers. Such memory buffers are required to be located in the upper 32K for CPU RAM address space. This requirement significantly simplifies the HBIOS proxy and improves performance by avoiding “double copies” of buffers.

## Function Overview

Character Input/Output (CIO)	Character Input – CIOIN Character Output – CIOOUT Character Input Status – CIOIST Character Output Status – CIOOST Character I/O Initialization – CIOINIT Character I/O Query – CIOQUERY Character I/O Device – CIODEVICE
Disk Input/Output (DIO)	Disk Status – DIOSTATUS Disk Reset – DIORESET Disk Seek – DIOSEEK Disk Read – DIORD Disk Write – DIOWR Disk Verify – DIOVERIFY Disk Format – DIOFORMAT Disk Device – DIODEVICE Disk Media – DIOMEDIA Disk Define Media – DIODEFMED Disk Capacity – DIOCAP Disk Geometry -- DIOGEOM
Real Time Clock (RTC)	RTC Get Time – RTCGETTIM RTC Set Time – RTCSETTIM RTC Get NVRAM Byte – RTCGETBYT RTC Set NVRAM Byte – RTCSETBYT RTC Get NVRAM Block – RTCGETBLK RTC Set NVRAM Block – RTCSETBLK
Video Display Adapter (VDA)	VDA Initialize – VDAINI VDA Query – VDAQRY VDA Reset – VDARES VDA Set Cursor Style – VDASCS VDA Set Cursor Position – VDASCP VDA Set Character Attribute – VDASAT VDA Set Character Color – VDASCO VDA Write Character – VDAWRC VDA Fill – VDAFIL VDA Copy – VDACPY VDA Scroll – VDASCR VDA Keyboard Status – VDAKST VDA Keyboard Flush – VDAKFL VDA Keyboard Read – VDAKRD

System (SYS)	System Reset – SYSRESET System Version – SYSVER System Set Bank – SYSSETBNK System Get Bank – SYSGETBNK System Set Copy – SYSSETCPY System Bank Copy – SYSBNKCPY System Alloc – SYSALLOC System Free – SYSFREE System Get – SYSGET System Set – SYSSET System Peek – SYSPEEK System Poke – SYSPOKE System Int – SYSINT
--------------	--

## Character Input/Output (CIO)

Character input/output functions require that a character unit be specified in the C register. This is the logical device number assigned during the boot process that identifies all character i/o devices uniquely. Each character device is handled by an appropriate driver (UART, ASCI, etc.) which is identified by a device type id from the table below.

Device Type	
0x00	UART
0x10	ASCI
0x20	PropIO VGA
0x30	ParPortProp VGA

Character devices can usually be configured with line characteristics such as speed, framing, etc. A word value (16 bit) is used to describe the line characteristics as indicated below:

		RTS	Baud Rate (encoded)						DTR	XON	Parity			Stop	Data 8/7/6	
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	

The 5-bit baud rate value (V) is encoded as  $V = 75 * 2^X * 3^Y$ . The bits are defined as YXXXX.

### Character Input - CIOIN (\$00)

<u>Input</u> B=\$00 (function) C=Unit	<u>Output</u> A=Status (0=OK, 1=Error) E=Character input
Wait for a single character to be available at the specified device and return the character in E. Function will wait indefinitely for a character to be available.	

### Character Output - CIOOUT (\$01)

<u>Input</u> B=\$01 (function) C= Unit E=Character to output	<u>Output</u> A=Status (0=OK, 1=Error)
Wait for unit to be ready to send a character, then send the character specified in E.	

### Character Input Status - CIOIST (\$02)

<u>Input</u> B=\$02 (function) C=Unit	<u>Output</u> A=Status: # characters in input buffer
Return the number of characters available to read in the input buffer of the unit specified. If the device has no input buffer, it is acceptable to return simply 0 or 1 where 0 means there is no character available to read and 1 means there is at least one character available to read.	

### *Character Output Status – CIOOST (\$03)*

<u>Input</u> B=\$03 (function) C=Unit	<u>Output</u> A=Status: output buffer space available
<p>Return the space available in the output buffer expressed as a character count. If a 16 byte output buffer contained 6 characters waiting to be sent, this function would return 10, the number of positions available in the output buffer. If the port has no output buffer, it is acceptable to return simply 0 or 1 where 0 means the port is busy and 1 means the port is ready to output a character.</p>	

### *Character IO Initialization – CIOINIT (\$04)*

<u>Input</u> B=\$04 (function) C=Unit DE=Line Characteristics	<u>Output</u> A=Status: 0=Success, otherwise failure
<p>Setup line characteristics (baudrate, framing, etc.) of the specified unit. Register pair DE specifies line characteristics. If DE contains -1 (0xFFFF), then the device will be reinitialized with the last line characteristics used.</p>	

### *Character IO Query – CIOQUERY (\$05)*

<u>Input</u> B=\$05 (function) C=Unit	<u>Output</u> A=Status: 0=Success, otherwise failure DE=Line Characteristics
<p>Reports the line characteristics (baudrate, framing, etc.) of the specified unit. Register pair DE contains the line characteristics upon return.</p>	

### *Character IO Device – CIODEVICE (\$06)*

<u>Input</u> B=\$06 (function) C=Unit	<u>Output</u> A=Status: 0=Success, otherwise failure C=Device Attributes D=Device Type E=Device Number
<p>Reports information about the character device unit specified. Register C indicates the device attributes: 0=RS-232 and 1=Terminal. Register D indicates the device type (driver) and register E indicates the physical device number assigned by the driver.</p>	

## Disk Input/Output (DIO)

Character input/output functions require that a character unit be specified in the C register. This is the logical disk unit number assigned during the boot process that identifies all disk i/o devices uniquely.

Each disk device is handled by an appropriate driver (IDE, SD, etc.) which is identified by a device type id from the table below.

Disk Device Type	
0x00	Memory Disk
0x10	Floppy Disk
0x20	RAM Floppy
0x30	IDE Disk
0x40	ATAPI Disk (not implemented)
0x50	PPIDE Disk
0x60	SD Card
0x70	PropIO SD Card
0x80	ParPortProp SD Card
0x90	SIMH HDSK Disk

The currently defined media types are:

Media ID	Value	Format
MID_NONE	0	No media installed
MID_MDROM	1	ROM Drive
MID_MDRAM	2	RAM Drive
MID_RF	3	RAM Floppy (LBA)
MID_HD	4	Hard Disk (LBA)
MID_FD720	5	3.5" 720K Floppy
MID_FD144	6	3.5" 1.44M Floppy
MID_FD360	7	5.25" 360K Floppy
MID_FD120	8	5.25" 1.2M Floppy
MID_FD111	9	8" 1.11M Floppy

## Disk Status – DIOSTATUS (\$10)

<u>Input</u> B=\$10 (function) C=Unit	<u>Output</u> A=Status (0=OK, 1=Error)
Returns the current status (result code) of the specified disk unit. This function does not clear an error status.	

### ***Disk Reset – DIORESET (\$11)***

<u>Input</u>	<u>Output</u>
B=\$11 (function) C=Unit	A=Status (0=OK, 1=Error)
<p>Reset the physical interface associated with the specified unit. Flag all units associated with the interface for unit initialization at next I/O call. Clear media identified unless locked. Reset result code of all associated units of the physical interface.</p>	

### ***Disk Seek – DIOSEEK (\$12)***

<u>Input</u>	<u>Output</u>
B=\$12 (function) C=Unit D:7=Address Type (0=CHS, 1=LBA) CHS: D:6-0=Head, E=Sector, HL=Track LBA: DE:HL is 32 bit block address	A=Status (0=OK, 1=Error)
<p>Update target CHS or LBA for next I/O request on designated unit. Physical seek is typically deferred until subsequent I/O operation.</p> <p>Bit 7 of D indicates whether the disk seek address is specified as cylinder/head/sector (CHS) or Logical Block Address (LBA). If D:7=1, then the remaining bits of the 32 bit register set DE:HL specify a linear, zero offset, block number. If D:7=0, then the remaining bits of D specify the head, E specifies sector, and HL specifies track.</p> <p>Note that not all devices will accept both types of addresses. Specifically, floppy disk devices must have CHS addresses. All other devices will accept either CHS or LBA. The DIOGEOM function can be used to determine if the device supports LBA addressing.</p>	

### ***Disk Read – DIOREAD (\$13)***

<u>Input</u>	<u>Output</u>
B=\$13 (function) C= Unit HL=Buffer Address E=Block Count	A=Status (0=OK, 1=Error) E=Blocks Read
<p>Read Block Count sectors to buffer address starting at current target sector. Current sector must be established by prior seek function; however, multiple read/write/verify function calls can be made after a seek function. Current sector is incremented after each sector successfully read. On error, current sector is sector where error occurred. Blocks read indicates number of sectors successfully read. Caller must ensure buffer address is large enough to contain data for all sectors requested.</p>	

### *Disk Write – DIOWRITE (\$14)*

<u>Input</u>	<u>Output</u>
B=\$14 (function) C= Unit HL=Buffer Address E=Block Count	A=Status (0=OK, 1=Error) E=Blocks Written
<p>Write Block Count sectors to buffer address starting at current target sector. Current sector must be established by prior seek function; however, multiple read/write/verify function calls can be made after a seek function. Current sector is incremented after each sector successfully written. On error, current sector is sector where error occurred. Blocks written indicates number of sectors successfully written. Caller must ensure buffer address is large enough to contain data for all sectors being written.</p>	

### *Disk Verify – DIOVERIFY (\$15)*

<u>Input</u>	<u>Output</u>
B=\$15 (function) C= Unit HL=Buffer Address E=Block Count (not implemented)	A=Status (0=OK, 1=Error) E=Blocks Verified
<p>***Not Implemented***</p>	

### *Disk Format – DIOFORMAT (\$16)*

<u>Input</u>	<u>Output</u>
B=\$15 (function) C= Unit D=Head E=Fill Byte HL=Cylinder	A=Status (0=OK, 1=Error)
<p>***Not Implemented***</p>	



### ***Disk Device - DIDEVICE (\$17)***

<u>Input</u>	<u>Output</u>
B=\$17 (function) C=Unit	A=Status (0=OK, 1=Error) C=Attributes D=Device Type E=Device Number
<p>Reports information about the character device unit specified. Register D indicates the device type (driver) and register E indicates the physical device number assigned by the driver.</p> <p>Register C reports the following device attributes:</p> <p>Bit 7: 1=Floppy, 0=Hard Disk (or similar, e.g. CF, SD, RAM)</p> <p>If Floppy:</p> <p>Bits 6-5: Form Factor (0=8", 1=5.25", 2=3.5", 3=Other) Bit 4: Sides (0=SS, 1=DS) Bits 3-2: Density (0=SD, 1=DD, 2=HD, 3=ED) Bits 1-0: Reserved</p> <p>If Hard Disk:</p> <p>Bit 6: Removable Bits 5-3: Type (0=Hard, 1=CF, 2=SD, 3=USB, 4=ROM, 5=RAM, 6=RAMF, 7=Reserved) Bits 2-0: Reserved</p>	

### ***Disk Media - DIOMED (\$18)***

<u>Input</u>	<u>Output</u>
B=\$13 (function) C=Unit E:0 Enable Media Discovery	A=Status (0=OK, 1=Error) E=Media ID
<p>Report the media definition for media in specified unit. If bit 0 of E is set, then perform media discovery or verification. If no media in device, return no media error.</p>	

### ***Disk Define Media - DIODEFMED (\$19)***

<u>Input</u>	<u>Output</u>
B=\$19 (function) C=Unit E=Media ID	A=Status (0=OK, 1=Error)
<p>*** Not implemented ***</p>	

### ***Disk Capacity – DIOCAPACITY (\$1A)***

<u>Input</u>	<u>Output</u>
B=\$1A (function) C=Unit HL=Buffer Address	A=Status (0=OK, 1=Error) DE:HL=Blocks on Device BC=Block Size
Report current media capacity information. ED:HL is a 32 bit number representing the total number of blocks on the device. BC contains the block size. If media is unknown, an error will be returned.	

### ***Disk Geometry – DIOGEOMETRY (\$1B)***

<u>Input</u>	<u>Output</u>
B=\$1B (function) C=Unit	A=Status (0=OK, 1=Error) HL=Cylinders D:6-0=Heads D:7=LBA Capability BC=Block Size
Report current media geometry information. If media is unknown, return error (no media).	

## Real Time Clock (RTC)

The Real Time Clock functions provide read/write access to the clock and related Non-Volatile RAM.

The time functions (RTCGTM and RTCSTM) require a 6 byte date/time buffer of the following format. Each byte is BCD encoded.

Offset	Contents
0	Year (00-99)
1	Month (01-12)
2	Date (01-31)
3	Hours (00-24)
4	Minutes (00-59)
5	Seconds (00-59)

### *RTC Get Time – RTCGETTIM(\$20)*

<u>Input</u> B=\$20 (function) HL=Time Buffer Address	<u>Output</u> A=Status: 0=Success, otherwise failure
Read the current value of the clock and store the date/time in the buffer pointed to by HL.	

### *RTC Set Time – RTCSETTIM(\$21)*

<u>Input</u> B=\$21 (function)	<u>Output</u> A=Status: 0=Success, otherwise failure
Set the current value of the clock based on the date/time in the buffer pointed to by HL.	

### *RTC Get NVRAM Byte – RTCGETBYT(\$22)*

<u>Input</u> B=\$22 (function) C=Index	<u>Output</u> A=Status: 0=Success, otherwise failure E=Value
Read a single byte value from the Non-Volatile RAM at the index specified by C. The value is returned in register E.	

### *RTC Set NVRAM Byte – RTCSETBYT(\$23)*

<u>Input</u> B=\$23 (function) C=Index	<u>Output</u> A=Status: 0=Success, otherwise failure E=Value
Write a single byte value into the Non-Volatile RAM at the index specified by C. The value to be written is specified in E.	

### ***RTC Get NVRAM Block – RTCGETBLK(\$24)***

<u>Input</u> B=\$24 (function) HL=Buffer	<u>Output</u> A=Status: 0=Success, otherwise failure
Read the entire contents of the Non-Volatile RAM into the buffer pointed to by HL. HL must point to a location in the top 32K of CPU address space.	

### ***RTC Set NVRAM Block – RTCSETBLK(\$25)***

<u>Input</u> B=\$25 (function) HL=Buffer	<u>Output</u> A=Status: 0=Success, otherwise failure
Write the entire contents of the Non-Volatile RAM from the buffer pointed to by HL. HL must point to a location in the top 32K of CPU address space.	

## Video Display Adapter (VDA)

The VDA functions are provided as a common interface to Video Display Adapters. Not all VDAs will include keyboard hardware. In this case, the keyboard functions should return a failure status.

The VDA functions require that a VDA device/unit be specified in the C register. The upper nibble (upper 4 bits) specifies the device. The lower nibble specifies the unit (not currently used).

The currently defined video devices are:









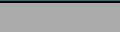







VDA ID	Value	Device
VDA_NONE	0	No VDA
VDA_VDU	1	ECB VDU board
VDA_CVDU	2	ECB Color VDU board
VDA_7220	3	ECB uPD7220 video display board
VDA_N8	4	TMS9918 video display built-in to N8

Depending on the capabilities of the hardware, the use of colors and attributes may or may not be supported. If the hardware does not support these capabilities, they will be ignored.

Color byte values are constructed using typical RGBI (Red/Green/Blue/Intensity) bits. The high four bits of the value determine the background color and the low four bits determine the foreground color. This results in 16 unique color values for both foreground and background. The following table illustrates the color byte value construction:

	Bit	Color
Background	7	Intensity
	6	Blue
	5	Green
	4	Red
Foreground	3	Intensity
	2	Blue
	1	Green
	0	Red

The following table illustrates the resultant color for each of the possible 16 values for foreground or background:

Foreground		Background		Color	Sample
_0	___0000	0_	0000___	Black	
_1	___0001	1_	0001___	Red	
_2	___0010	2_	0010___	Green	
_3	___0011	3_	0011___	Brown	
_4	___0100	4_	0100___	Blue	
_5	___0101	5_	0101___	Magenta	
_6	___0110	6_	0110___	Cyan	
_7	___0111	7_	0111___	White	
_8	___1000	8_	1000___	Gray	
_9	___1001	9_	1001___	Light Red	
_A	___1010	A_	1010___	Light Green	
_B	___1011	B_	1011___	Yellow	
_C	___1100	C_	1100___	Light Blue	
_D	___1101	D_	1101___	Light Magenta	
_E	___1110	E_	1110___	Light Cyan	
_F	___1111	F_	1111___	Bright White	

Attribute byte values are constructed using the following bit encoding:

Bit	Effect
7	n/a (0)
6	n/a (0)
5	n/a (0)
4	n/a (0)
3	n/a (0)
2	Reverse
1	Underline
0	Blink

The following codes are returned by a keyboard read to signify non-ASCII keystrokes:

Value	Keystroke	Value	Keystroke
E0	F1	F0	Insert
E1	F2	F1	Delete
E2	F3	F2	Home
E3	F4	F3	End
E4	F5	F4	PageUp
E5	F6	F5	PadeDown
E6	F7	F6	UpArrow
E7	F8	F7	DownArrow
E8	F9	F8	LeftArrow
E9	F10	F9	RightArrow
EA	F11	FA	Power
EB	F12	FB	Sleep
EC	SysReq	FC	Wake
ED	PrintScreen	FD	Break
EE	Pause	FE	
EF	App	FF	

### *Video Display Adapter Initialize –VDINI (\$40)*

<u>Input</u>	<u>Output</u>
B=\$40 (function) C=Device/Unit E=Video Mode (device specific) HL=Character Bitmap (optional)	A=Status: 0=Success, otherwise failure
<p>Performs a full (re)initialization of the specified video device. The screen is cleared and the keyboard buffer is flushed. If the specified VDA supports multiple video modes, the requested mode can be specified in E (set to 0 for default/not specified). Mode values are specific to each VDA.</p> <p>HL may point to a location in memory with the character bitmap to be loaded into the VDA video processor. The location MUST be in the top 32K of the CPU memory space. HL must be set to zero if no character bitmap is specified (the VDA video processor will utilize a default character bitmap).</p>	

### *Video Display Adapter Query –VDAQRY (\$41)*

<u>Input</u>	<u>Output</u>
B=\$41 (function) C=Device/Unit HL=Character Bitmap Data (optional)	A=Status: 0=Success, otherwise failure C=Video Mode D=Row Count E=Column Count HL=Character Bitmap Data (zero if none)
<p>Return information about the specified video device. C will be set to the current video mode. DE will return the dimensions of the video display as measured in rows and columns. Note that this is the <b>count</b> of rows and columns, not the <b>last</b> row/column number.</p> <p>If HL is not zero, it must point to a suitably sized memory buffer in the upper 32K of CPU address space that will be filled with the current character bitmap data. It is critical that HL be set to zero if it does not point to a proper buffer area or memory corruption will result. The video device driver may not have the ability to provide character bitmap data. In this case, on return, HL will be set to zero.</p>	

### *Video Display Adapter Reset –VDARES (\$42)*

<u>Input</u>	<u>Output</u>
B=\$42 (function) C=Device/Unit	A=Status: 0=Success, otherwise failure
<p>Performs a soft reset of the Video Display Adapter. Should clear the screen, home the cursor, restore active attribute and color to defaults. Keyboard should be flushed.</p>	

### *Video Display Adapter Set Cursor Style –VDASCS (\$43)*

<u>Input</u>	<u>Output</u>
B=\$43 (function) C=Device/Unit D=Start/End pixel E=Style	A=Status: 0=Success, otherwise failure
<p>If supported by the video hardware, adjust the format of the cursor such that the cursor starts at the pixel specified in the top nibble of D and end at the pixel specified in the bottom nibble of D. So, if D=\$08, a block cursor would be used that starts at the top pixel of the character cell and ends at the ninth pixel of the character cell.</p> <p>Register E is reserved to control the style of the cursor (blink, visibility, etc.), but is not yet implemented.</p> <p>Adjustments to the cursor style may or may not be possible for any given video hardware.</p>	



#### *Video Display Adapter Set Cursor Position –VDASCP (\$44)*

<u>Input</u> B=\$44 (function) C=Device/Unit D=Row E=Column	<u>Output</u> A=Status: 0=Success, otherwise failure
Reposition the cursor to the specified row and column. Specifying a row/column that exceeds the boundaries of the display results in undefined behavior. Cursor coordinates are 0 based (0,0 is the upper left corner of the display).	

#### *Video Display Adapter Set Character Attribute –VDASAT (\$45)*

<u>Input</u> B=\$45 (function) C=Device/Unit E=Character Attribute Code	<u>Output</u> A=Status: 0=Success, otherwise failure
Assign the specified character attribute code to be used for all subsequent character writes/fills. This attribute is used to fill new lines generated by scroll operations. Refer to the character attribute for a list of the available attribute codes. Note that a given video display may or may not support any/all attributes.	

#### *Video Display Adapter Set Character Color –VDASCO (\$46)*

<u>Input</u> B=\$46 (function) C=Device/Unit E=Color Code	<u>Output</u> A=Status: 0=Success, otherwise failure
Assign the specified color code to be used for all subsequent character writes/fills. This color is also used to fill new lines generated by scroll operations. Refer to color code table for a list of the available color codes. Note that a given video display may or may not support any/all colors.	

#### *Video Display Adapter Write Character –VDAWRC (\$47)*

<u>Input</u> B=\$47 (function) C=Device/Unit E=Character	<u>Output</u> A=Status: 0=Success, otherwise failure
Write the character specified in E. The character is written starting at the current cursor position and the cursor is advanced. If the end of the line is encountered, the cursor will be advanced to the start of the next line. The display will <b>not</b> scroll if the end of the screen is exceeded.	

### *Video Display Adapter Fill –VDAFIL (\$48)*

<u>Input</u> B=\$48 (function) C=Device/Unit E=Character HL=Count	<u>Output</u> A=Status: 0=Success, otherwise failure
<p>Write the character specified in E to the display the number of times specified in HL. Characters are written starting at the current cursor position and the cursor is advanced by the number of characters written. If the end of the line is encountered, the characters will continue to be written starting at the next line as needed. The display will <b>not</b> scroll if the end of the screen is exceeded.</p>	

### *Video Display Adapter Copy –VDACPY (\$49)*

<u>Input</u> B=\$48 (function) C=Device/Unit D=Source Row E=Source Column L=Count (max 255)	<u>Output</u> A=Status: 0=Success, otherwise failure
<p>Copy count (L) bytes from the source row/column (DE) to current cursor position. The cursor position is not updated. The maximum count is 255. Copying to/from overlapping areas is not supported and will have an undefined behavior. The display will <b>not</b> scroll if the end of the screen is exceeded. Copying beyond the active screen buffer area is not supported and results in undefined behavior.</p>	

### *Video Display Adapter Scroll –VDASCR (\$4A)*

<u>Input</u> B=\$49 (function) C=Device/Unit E=Scroll distance (# lines)	<u>Output</u> A=Status: 0=Success, otherwise failure
<p>Scroll the video display by the number of lines specified in E. If E contains a negative number, then reverse scroll should be performed.</p>	

### *Video Display Adapter Keyboard Status –VDAKST (\$4B)*

<u>Input</u> B=\$4A (function) C=Device/Unit	<u>Output</u> A=Status: # key codes in keyboard buffer
<p>Return a count of the number of key codes in the keyboard buffer. If it is not possible to determine the actual number in the buffer, it is acceptable to return 1 to indicate there are key codes available to read and 0 if there are none available.</p>	

### *Video Display Adapter Keyboard Flush –VDAKFL (\$4C)*

<u>Input</u>	<u>Output</u>
B=\$4B (function) C=Device/Unit	A=Status: 0=Success, otherwise failure
If a keyboard buffer is in use, it should be purged and all contents discarded.	

### *Video Display Adapter Keyboard Read –VDAKRD (\$4D)*

<u>Input</u>	<u>Output</u>
B=\$4C (function) C=Device/Unit	A=Status: 0=Success, otherwise failure C=Scancode D=Keystate E=Keycode
<p>Read next key code from keyboard. If a keyboard buffer is used, return the next key code in the buffer. If no key codes are available, wait for a keypress and return the keycode.</p> <p>The scancode value is the raw scancode from the keyboard for the keypress. Scancodes are from scancode set 2 standard.</p> <p>The keystate is a bitmap representing the value of all modifier keys and shift states as they existed at the time of the keystroke. The bitmap is defined as:</p> <ul style="list-style-type: none"><li>Bit 7: Set to indicate key pressed was from the num pad</li><li>Bit 6: Set to indicate Caps Lock was active</li><li>Bit 5: Set to indicate Num Lock was active</li><li>Bit 4: Set to indicate Scroll Lock was active</li><li>Bit 3: Set to indicate Windows key was held down</li><li>Bit 2: Set to indicate Alt key was held down</li><li>Bit 1: Set to indicate control key was held down</li><li>Bit 0: Set to indicate Shift key was held down</li></ul> <p>Keycodes are generally returned as appropriate ASCII values, if possible. Special keys, like function keys, are returned as reserved codes as described at the start of this section.</p>	

## System (SYS)

### System Reset – SYSRESET (\$F0)

Input	Output
B=\$F0 (function)	A=Previously active Bank ID
Perform a soft reset of HBIOS. Releases all HBIOS memory allocated by current OS. Does not reinitialize physical devices.	

### System Version – SYSVER (\$F1)

Input	Output
B=\$F1 (function) C=Reserved (set to 0)	A=Previously active Bank ID DE=Version (Maj/Min/Upd/Pat) L=Platform Id
<p>This function will return the HBIOS version number. The version number is returned in DE. High nibble of D is the major version, low nibble of D is the minor version, high nibble of E is the patch number, and low nibble of E is the build number.</p> <p>The hardware platform is identified in L:</p> <ul style="list-style-type: none"><li>1: SBC V1 or V2</li><li>2: ZETA</li><li>3: ZETA 2</li><li>4: N8</li><li>5: MK4</li><li>6: UNA</li><li>7: RC2014 w/ Z80</li><li>8: RC2014 w/ Z180</li><li>9: Easy Z80</li></ul>	

### System Set Bank – SYSSETBNK (\$F2)

Input	Output
B=\$F2 (function) C=Bank ID	A=Status: 0=Success, otherwise failure C=Previously active Bank ID
Activates the Bank ID specified in C and returns the previously active Bank ID in C. The caller <b>MUST</b> be invoked from code located in the upper 32K and the stack <b>must</b> be in the upper 32K.	

### System Get Bank – SYSGETBNK (\$F3)

Input	Output
B=\$F2 (function)	A=Status: 0=Success, otherwise failure C=Active Bank ID
Returns the currently active Bank ID in C.	

### *System Set Copy – SYSSETCPY (\$F4)*

<u>Input</u>	<u>Output</u>
B=\$F4 (function) D=Destination Bank Id E=Source Bank Id HL=Count of Bytes to Copy	A=Status: 0=Success, otherwise failure
<p>Prepare for a subsequent interbank memory copy (SYSBNKCPY) function by setting the source bank, destination bank, and byte count for the copy. The bank id's are not range checked and must be valid for the system in use.</p> <p>No bytes are copied by this function. The SYSBNKCPY must be called to actually perform the copy. The values setup by this function will remain unchanged until another call is made to this function. So, after calling SYSSETCPY, you may make multiple calls to SYSBNKCPY as long as you want to continue to copy between the already established Source/Destination Banks and the same size copy if being performed.</p>	

### *System Bank Copy – SYSBNKCPY (\$F5)*

<u>Input</u>	<u>Output</u>
B=\$F5 (function) DE=Destination address HL=Source address	A=Status: 0=Success, otherwise failure
<p>Copy memory between banks. The source bank, destination bank, and byte count to copy MUST be established with a prior call to SYSSETCPY. However, it is not necessary to call SYSSETCPY prior to subsequent calls to SYSBNKCPY if the source/destination banks and copy length do not change.</p> <p>WARNINGS:</p> <ul style="list-style-type: none"><li>• This function is inherently dangerous and does not prevent you from corrupting critical areas of memory. Use with EXTREME caution.</li><li>• Overlapping source and destination memory ranges are not supported and will result in undetermined behavior.</li><li>• Copying of byte ranges that cross bank boundaries is undefined.</li></ul>	

### *System Alloc – SYSALLOC (\$F6)*

<u>Input</u>	<u>Output</u>
B=\$F6 (function) HL=Size in bytes	A=Status: 0=Success, otherwise failure HL=Address of allocated memory block
<p>This function will attempt to allocate a block of memory of HL bytes from the internal HBIOS heap. The HBIOS heap resides in the HBIOS bank in the area of memory left unused by HBIOS. If the allocation is successful, the address of the allocated memory block is returned in HL. You will typically want to use the SYSBNKCPY function to read/write the allocated memory.</p>	

### *System Free – SYSFREE (\$F7)*

<u>Input</u> B=\$F7 (function) HL=Address of memory block to free	<u>Output</u> A=Status: 0=Success, otherwise failure
*** This function is not yet implemented ***	

### *System Get – SYSGET (\$F8)*

<u>Input</u> B=\$F8 (function) C=Subfunction	<u>Output</u> A=Status: 0=Success, otherwise failure
This function will report various system information based on the sub-function value. Additional input and output registers may be used as defined by the sub-function.	
CIOCNT (\$00)	Return count of serial units in E
DIOCNT (\$10)	Return count of disk units in E
VDACNT (\$40)	Return count of video display units in E
TIMER (\$D0)	Return current timer tick count value in DE:HL
BOOTINFO (\$E0)	Return boot bank id in L, disk unit in D, and disk slice in E
CPUINFO (\$F0)	Return Z80 variant in H, CPU Speed in MHz in L, and CPU Speed in KHz in DE
MEMINFO (\$F1)	Return count of 32K ROM banks in D and count of RAM banks in E
BNKINFO (\$F2)	Return BIOS bank id in D and User bank id in E

### *System Set – SYSSET (\$F9)*

<u>Input</u> B=\$F9 (function) C=Subfunction	<u>Output</u> A=Status: 0=Success, otherwise failure
This function will set various system parameters based on the sub-function value. Additional input and output registers may be used as defined by the sub-function.	
TIMER (\$D0)	Set timer tick count value from DE:HL
BOOTINFO (\$E0)	Set boot bank id in L, disk unit in D, and disk slice in E

### *System Peek – SYSPEEK (\$FA)*

<u>Input</u> B=\$FA (function) D=Bank HL=Address	<u>Output</u> A=Status: 0=Success, otherwise failure E=Byte Value
This function gets a single byte value from the specified bank/address. The bank specified is not range checked.	

### *System Poke – SYSPoke (\$FB)*

<u>Input</u> B=\$FB (function) D=Bank E=Value HL=Address	<u>Output</u> A=Status: 0=Success, otherwise failure
This function sets a single byte value in the specified bank/address. The bank specified is not range checked.	

### *System Int – SYSINT (\$FC)*

<u>Input</u>	<u>Output</u>
B=\$FC (function) C=Subfunction	A=Status: 0=Success, otherwise failure
<p>This function allows the caller to query information about the interrupt configuration of the running system and allows adding or hooking interrupt handlers dynamically. Register C is used to specify a subfunction. Additional input and output registers may be used as defined by the sub-function.</p> <p>Note that during interrupt processing, the lower 32K of CPU address space will contain the RomWBW HBIOS code bank, not the lower 32K of application TPA. As such, a dynamically installed interrupt handler does not have access to the lower 32K of TPA and must be careful to avoid modifying the contents of the lower 32K of memory. Invoking RomWBW HBIOS functions within an interrupt handler is not supported.</p> <p>Interrupt handlers are different for IM1 or IM2.</p> <p>For IM1:</p> <p style="padding-left: 40px;">The new interrupt handler is responsible for chaining (JP) to the previous vector if the interrupt is not handled. If the interrupt is handled, the new handler may simply return (RET). When chaining to the previous interrupt handler, ZF must be set if interrupt is handled and ZF cleared if not handled. The interrupt management framework takes care of saving and restoring AF, BC, DE, HL, and IY. Any other registers modified must be saved and restored by the interrupt handler.</p> <p>For IM2:</p> <p style="padding-left: 40px;">The new interrupt handler may either replace or hook the previous interrupt handler. To replace the previous interrupt handler, the new handler just returns (RET) when done. To hook the previous handler, the new handler can chain (JP) to the previous vector. Note that initially all IM2 interrupt vectors are set to be handled as “BAD” meaning that the interrupt is unexpected. In most cases, you do not want to chain to the previous vector because it will cause the interrupt to display a “BAD INT” system panic message.</p> <p style="padding-left: 40px;">The interrupt framework will take care of issuing an EI and RETI instruction. Do not put these instructions in your new handler. Additionally, interrupt management framework takes care of saving and restoring AF, BC, DE, HL, and IY. Any other registers modified must be saved and restored by the interrupt handler.</p> <p>If the caller is transient, then the caller must remove the new interrupt handler and restore the original one prior to termination. This is accomplished by calling this function with the Interrupt Vector set to the Previous Vector returned in the original call.</p> <p>The caller is responsible for disabling interrupts prior to making an INTSET call and enabling them afterwards. The caller is responsible for ensuring that a valid interrupt handler is installed prior to enabling any hardware interrupts associated with the handler. Also, if the handler is transient, the caller must disable the hardware interrupt(s) associated with the handler prior to uninstalling it.</p>	



INTINF (\$00)	Return interrupt mode in D and size of interrupt vector table in E. For IM1, the size of the table is the number of vectors chained together. For IM2, the size of the table is the number of slots in the vector table.
INTGET (\$10)	On entry, register E must contain an index into the interrupt vector table. On return, HL will contain the address of the current interrupt vector at the specified index.
INTSET (\$20)	On entry, register E must contain an index into the interrupt vector table and register HL must contain the address of the new interrupt vector to be inserted in the table at the index. On return, HL will contain the previous address in the table at the index.