|  |
| --- |
| RomWBW User Guide |
| RetroBrew Computing RomWBW Version 2.8 |
| June 8, 2016 |



Contents

[Overview 2](#_Toc453182560)

[System Requirements 2](#_Toc453182561)

[Acknowledgements 4](#_Toc453182562)

[RomWBW Distribution Package 5](#_Toc453182563)

[Getting Started 6](#_Toc453182564)

[Background 9](#_Toc453182565)

[General Design Strategy 10](#_Toc453182566)

[Runtime Memory Layout 10](#_Toc453182567)

[System Boot Process 11](#_Toc453182568)

[ROM Boot 14](#_Toc453182569)

[Application Boot 14](#_Toc453182570)

[Notes 14](#_Toc453182571)

[Driver Model 15](#_Toc453182572)

[Character / Emulation / Video Services 15](#_Toc453182573)

[HBIOS Reference 17](#_Toc453182574)

[Invocation 17](#_Toc453182575)

[Function Overview 17](#_Toc453182576)

[Character Input/Output (CIO) 19](#_Toc453182577)

[Disk Input/Output (DIO) 21](#_Toc453182578)

[Real Time Clock (RTC) 24](#_Toc453182579)

[Emulation (EMU) 26](#_Toc453182580)

[Video Display Adapter (VDA) 27](#_Toc453182581)

[System (SYS) 35](#_Toc453182582)

[Memory Layout Detail 38](#_Toc453182583)

# Overview

RomWBW is system software that support the Z80/Z180 based computing platforms produced by the RetroBrew Computing Group (see <http://www.retrobrewcomputers.org>). The goal of the project is to provide all firmware and software needed to make a fully functional computing platform. This includes both firmware (ROM) and software (disk images). The source code is provided and is licensed under GPL v3. A GitHub repository is used to maintain all source code and can be found at <http://www.github.com/RomWBW>.

Essentially all Z80/Z180 based hardware produced by RetroBrew Computing is fully supported by RomWBW. Much of the software was adapted from software produced by others in the community (see Acknowledgements) and is packaged within RomWBW to provide an integrated solution. A companion document (RomWBW System Guide) provides substantial detail on the architecture and internal operation of this software.

It is worth noting that this software is a perpetual work-in-progress. While it has become fairly stable and robust over time, it is undergoing constant updates to support new and revised hardware produced by the community. Backward compatibility between releases should not be assumed.

In order to provide a complete solution, the RomWBW package incorporates a hardware BIOS (hardware drivers) and selected operating systems and application software. All software is derived from the CP/M era of 8-bit computing. The operating systems included have been adapted to run under the RomWBW architecture. In general, application software has simply been included as originally distributed by the vendors and required no adaptation.

The RomWBW distribution package includes all the tools required to easily build the software from the source that is included in the package. The package includes a range of pre-built ROM and disk images. These are usually sufficient to get your hardware up and running simply by programming a ROM and optionally copying disk image(s) to a floppy disk, CF Card, or SD Card. If you wish to highly customize your system software, it is straightforward to modify the source code and build your own. At present, this requires Microsoft Windows XP or greater. All of the tools have counterparts for Linux, so building the software under Linux should be possible with a little effort.

# System Requirements

RomWBW is purely a software project. It assumes you have a fully functional hardware platform on which to host the software. A great deal of information on procuring and building the appropriate hardware is found on the RetroBrew Computing Wiki at <http://www.retrobrewcomputing.org>. Additionally, the RetroBrew Computing Group has a very active forum at <http://www.retrobrewcomputing.org/forum>. This forum is the ideal place to ask questions and get guidance for hardware and software. It is the primary forum for supporting RomWBW.

The starting point for a hardware platform that will appropriately host RomWBW software is one of the following CPU boards:

* **SBC v1 or v2**

This is the original Z80 CPU board produced by the community. It remains a very functional platform and is relatively easy to build. Note that v1 has a design deficiency that may or may not prevent the proper operation of RomWBW (bank switching does not always function reliably). The SBC CPU board features an ECB bus connector which allows it to be expanded with a backplane and peripheral boards.

* **Zeta v1 or v2**

The Zeta is very similar to the SBC board and is generally compatible with it. However, the Zeta platform is optimized to be a compact, standalone system. In addition to the features of the SBC, it includes an onboard floppy disk controller and the form factor of the board allows it to be mounted directly to a 3.5” floppy disk for a complete computing solution. It optionally supports a single daughter board that provides SD Card storage, VGA Monitor interface, and PS/2 keyboard interface. Although it does not have a bus interface, Zeta is powerful, compact, and fully featured. The Zeta v2 primarily adds enhanced bank switching and an interrupt controller which is not required by RomWBW.

* **N8**

The N8 is a very robust SBC. It is significantly larger than the SBC and incorporates a wide range of peripherals right on the one board (although it also supports expansion via ECB bus). The N8 is based on the Z180 CPU and incorporates interfaces for 2 serial ports, 2 parallel ports, IDE Hard Disk / CF Card, SD Card, sound synthesizer, video display, PS/2 keyboard & mouse interface, and floppy disk controller. This board is very powerful, but more challenging to build. It is not compatible with the SBC/Zeta – it implements a different bank switching mechanism.

* **Mark IV**

The Mark IV by John Coffman is similar to the SBC in that it shares the same form factor and ECB for expansion. However, it is substantially more powerful featuring a Z180 CPU and onboard CF Card and SD Card interfaces.

RomWBW fully supports all of the above boards as a starting point. For Zeta, the ParPortProp is supported as an option. The other platforms all support the ECB bus for adding optional peripheral support boards.

In addition to the hardware listed above, RomWBW also runs well on the Microsoft Windows based SIMH AltariZ80 Simulator. This software provides a mode that simulates the SBC v2 host board listed above which allows you to try all of the RomWBW features without any actual hardware. The distribution package contains a copy of the simulator software for MS Windows, so it is very easy to use it (see Getting Started).

Note that RomWBW assumes specific board configuration settings. You must ensure that you set the jumpers/switches of each board as required by RomWBW (unless you modify RomWBW and produce a custom version that supports your custom board configurations). The required board configuration settings are documented in Appendix A.

Note that RomWBW assumes there is 512KB of ROM and 512KB of RAM for all systems. It is fine if your systems has more RAM or ROM than this, but it is problematic if you have less. It would be very rare for a system to have less that these amounts, but be aware of this constraint. These assumptions can be modified via customization later, but the pre-built software must have these minimums.

All of the host boards include a serial port. RomWBW will use this serial port for output when you start your system. By default RomWBW uses 38,400 baud, 8 data bits, 1 stop bit, and no parity. You will need to connect the primary serial port of the host board to a terminal (or PC running terminal emulation software) to see the system output when you start RomWBW.

The use of the ECB bus signals is standardized such that any ECB add-on board can generally be combined with any of the ECB host boards to provide enhanced functionality. Appendix A provides an inventory of the boards supported by RomWBW along with relevant notes and required board configuration settings. Appendix A also includes a compatibility/support matrix between the host boards and the peripheral support boards.

# Acknowledgements

First, I want to be clear that RomWBW is not the only option available for system software on RetroBrew Computing Z80/Z180 hardware. While many similar projects are no longer active, they are very useful and may contain functionality that has not been incorporated in RomWBW. All of the software projects (including RomWBW) are listed in the RetroBrew Computing Wiki.

The UNA Project from John Coffman is the other currently active software project for the Z80/Z180 projects. It is far more advanced than RomWBW in that it can support all 4 host boards with a single ROM image and allows dynamic system configuration via onboard setup. It does not yet support the full range of hardware or video capabilities of RomWBW. Note that RomWBW supports an UNA “hybrid” configuration in which the UNA BIOS is combined with the RomWBW OS and application layers.

The RetroBrew Computing Group has existed in various forms since about 2010 (?). Many individuals have contributed to the community. The original founder of the community has moved on and requested anonymity going forward. However, his initiative is greatly appreciated. While there is no formal structure to the community, Andrew Bingham has taken the mantle of responsibility for the wiki and discussion group. This is a critical function and he deserves substantial credit for this effort.

Earlier in the community’s history, there were multiple branches of software development. Frequently, when a new board was produced, someone would create an independent code branch to support it. This started to lead to a very fragmented set of software that made it very difficult to create an integrated system with selected boards. RomWBW came about as an effort to create a framework that would allow arbitrary hardware to be easily added without creating entirely separate branches of code.

RomWBW essentially became a semi-structured place to incorporate all of the many software efforts of the community. Initially, most of the RomWBW codebase was simply a “cut and paste” of the software produced by others. Over time, much of this software has been repeatedly revised such that it is no longer similar to the original, but RomWBW owes it’s existence to the contributions of many other individuals. A few of those people are listed below and I apologize for anyone that I may have inadvertently omitted. I have intentionally omitted the original founder of the community based on my understanding of his desire to be anonymous going forward.

Douglas Goodall worked in very close collaboration with me during the first year of the RomWBW Project. He produced an excellent set of supporting utility programs and provided a great deal of design input. Regrettably, his utilities no longer have a caretaker and have become unusable as RomWBW has evolved, but their legacy continues within the current codebase. The source for all of these utilities is still available if anyone wants to take responsibility for bringing them back to current status.

John Coffman has personally produced a great deal of the hardware designs within the community. RomWBW contains many portions of code that John contributed over time. Additionally, he has been instrumental in providing advice and guidance to me for many years now.

Dan Werner has been one of the most prolific coders within the community. A great deal of his code was incorporated in the early RomWBW releases. David Giles produced some code that also provided a more integrated set of software for each host board. Over time, much of his code was incorporated in RomWBW. Likewise, Max Scane has produced code that ultimately wound up in RomWBW – specifically, he contributed the CLRDIR application.

It is my belief that all code incorporated into RomWBW has been done so with the express or implied permission of the original authors. I realize there have been many other individuals that have contributed to RomWBW and apologize for not naming all of them.

# RomWBW Distribution Package

RomWBW is distributed as a complete package (a .zip file) that contains everything appropriate for the different hardware variations. In other words, don’t look for a specific distribution for your hardware, you just want the current package. Within the package, you will find documentation, source code, build tools, and pre-built ROM and disk images.

The distribution package is usually hosted at the following locations:

* RetroBrew Computing Wiki:  
  Navigate to [https://www.retrobrewcomputers.org](https://www.retrobrewcomputers.org/doku.php?id=software:firmwareos:romwbw:start). Then, using the navigation menu on the left, choose software 🡪 firmwareos 🡪 romwbw to reach the RomWBW Project Page. At the bottom of the page you will find the distribution files listed for download.
* GitHub:  
  Navigate to <https://github.com/wwarthen/RomWBW> to reach the RomWBW Project on GitHub. Select “releases” to reach the list of distribution files. Note that you will see both Prereleases and Releases listed. Unless you specifically want to test work-in-progress, please download only a Release version.

The package should be named something like RomWBW-2.8-Package.zip. Using any standard personal computer (Windows, Linux, Mac, etc.), download and extract the contents of the zip file using any of the standard zip tools. You will see that there are several directories that are used to organize the contents. Don’t get overwhelmed. Initially, all you really care about is the Output directory (and possibly the Doc directory):

|  |  |
| --- | --- |
| Directory | Contents |
| Doc | Contains documentation files for various parts of the RomWBW distribution including operating systems, applications, and various aspects of RomWBW itself. In most cases, the name of the file should identify the component being documented. |
| Hardware | Files that are specific to certain hardware components. For example, it has the font ROM images for the video display boards. You do not need any of these files for the host boards used initially. Appendix A describes the contents of this directory for relevant boards. |
| Images | Files that are used to create disk images. Since the disk images are all pre-built, you do not need to worry about this directory until you want to create custom disk images (documented later). |
| Output | The ROM and Disk Images that you need to get started as documented below in Getting Started. |
| Source | The source code files that are compiled or assembled to create RomWBW. Again, the output is pre-built, so you don’t need to worry about this directory until you want to customize your system. |
| Tools | Windows-based applications that are used to build RomWBW. It also contains applications that you can use to copy disk images to floppy disks, CF Cards, SD Cards, etc. It also has the SIMH simulation software. |

In most cases, you will find a ReadMe.txt file in the directory which describes the contents of the directory in more detail.

# Getting Started

Because of the wide variety of hardware combinations, there is no “one case fits all” approach to getting started. The good news is that RomWBW operates very consistently regardless of the specific hardware. The operating systems, applications, and storage formats are all common. However, building and testing your hardware is entirely outside the scope of this document. The RetroBrew Computing Forum (<https://www.retrobrewcomputers.org/forum>) is probably the best place to get advice if you get stuck on hardware issues.

It is not necessary, but I highly recommend running RomWBW under the SIMH Simulator as a first step. This requires no hardware and will allow you to see how it should look when you use it on real hardware. Since the SIMH software is included in the distribution package, you can start it with a single command. Using a command prompt window, navigate to the high level directory of the distribution. Enter the command “sim” and the simulator should start up and you will see the RomWBW boot display similar to the one below.

|  |
| --- |
| ROM Image: 'Output\SBC\_simh.rom'  HDSK0: WARNING: Unsupported disk capacity, assuming HDSK type with capacity 34078KB.  HDSK0: WARNING: Forcing WRTLCK.  HDSK0: WARNING: Fixing geometry.  HDSK1: WARNING: Unsupported disk capacity, assuming HDSK type with capacity 34078KB.  HDSK1: WARNING: Forcing WRTLCK.  HDSK1: WARNING: Fixing geometry.  RetroBrew HBIOS v2.8.0-pre.5, 2016-06-05  SBC Z80 @ 20.000MHz ROM=512KB RAM=512KB  UART0: IO=0x68 8250 MODE=38400,8,N,1  SIMRTC: Tue 2016-06-07 19:55:51  MD: UNITS=2 ROMDISK=384KB RAMDISK=384KB  HDSK: UNITS=2  Unit Device Type Capacity/Mode  ---------- ---------- ---------------- --------------------  Disk 0 MD1: RAM Disk 384KB,LBA  Disk 1 MD0: ROM Disk 384KB,LBA  Disk 2 HDSK0: Hard Disk 65MB,LBA  Disk 3 HDSK1: Hard Disk 65MB,LBA  Serial 0 UART0: RS-232 38400,8,N,1  SBC Z80 Boot Loader  Boot: (C)PM, (Z)System, (M)onitor,  (L)ist disks, or Disk Unit # ===> |

In all cases, you will want to start with a Z80/Z180 host board. Any of the boards listed in System Requirements will work fine. I strongly recommend that you initially work on getting just the single host board running by itself – don’t even plug it into an ECB backplane.

Given a host board that is assembled and passes any hardware checks recommended by the boards designer, you need to make sure the board is configured for RomWBW. Refer to the entry in Appendix A for your board and confirm that all switches and jumpers on the board are set as required by RomWBW.

Your initial goal is to locate and program a ROM image for your host board. The ROM images are located in the Output directory. You are looking for the files that end in “.rom”. Don’t worry about all of the other variations at this point. Refer to the following table to determine the ROM image that you want:

|  |  |
| --- | --- |
| Host Board | Rom Image File |
| SBC v1 or v2 | SBC\_std.rom |
| Zeta v1 | ZETA\_std.rom |
| Zeta v2 | ZETA2\_std.rom |
| N8 (2511 datestamp | N8\_2511.rom |
| N8 (2312 datestamp) | N8\_2312.rom |
| Mark IV | MK4\_std.rom |

Locate the appropriate ROM image file in the Output directory based on the table above. You should see that the file is exactly 512KB in size. As indicated above in System Requirements, your system should have a ROM capacity of 512KB or greater. You need to program the file to your ROM using whatever tool you have. Programming a ROM chip is beyond the scope of this document, but any feel free to ask for help at the RetroBrew Computing Forum. The ROM image files are pure binary and should be programmed into the ROM chip starting at address 0 thru address 7FFFH. Insert the programmed ROM chip in your system.

Initially, you will need two external connections to your board. Power and serial port. All of the CPU boards provide an onboard power connection. Refer to the board designer’s notes on the RBC Wiki for more information on the power connection your board requires.

Finally, you must connect the primary serial port of your host board to a terminal using 38,400 baud, 8 data bits, 1 stop bit, and no parity. You can use either a dedicated terminal or use terminal emulation software on your PC (Tera Term works well and is free). When connecting to a standard PC serial port, a null modem cable is required. There is a good document on the Wiki that explains cabling of serial ports at http://???????.

Assuming you have a host board that is assembled and seems

RomWBW provides a complete firmware package for all of the Z80-based systems that are available in the N8VEM Community (see http:// <http://n8vem-sbc.pbworks.com>). 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 N8VEM 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 N8VEM 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 N8VEM 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 N8VEM 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 N8VEM 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 N8VEM Z80 CPU platforms provide a physical address space that is much larger than the CPU address space (typically 512K or 1MB). 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), typically 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 N8VEM 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 N8VEM 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 256 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 normal (bank 0).

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 256 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 tiny 256 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. You can refer to the section of this document called

***System HCB Get Byte – SYSHCBGETB ($F8)***

|  |  |
| --- | --- |
| Input B=$F8 (function) C=Value Index (HCB Offset) | Output E=Value |
| Retrieve the HBIOS Control Block value at the index specified in C and return the byte value in E. | |

***System HCB Put Byte – SYSHCBPUTB ($F9)***

|  |  |
| --- | --- |
| Input B=$F9 (function) C=Value Index (HCB Offset) E=Value | Output |
| Write the HBIOS Control Block value specified in E at the index specified in C | |

***System HCB Get Word – SYSHCBGETB ($FA)***

|  |  |
| --- | --- |
| Input B=$F8 (function) C=Value Index (HCB Offset) | Output DE=Value |
| Retrieve the HBIOS Control Block value at the index specified in C and return the word value in DE. | |

***System HCB Put Word – SYSHCBPUTB ($FB)***

|  |  |
| --- | --- |
| Input B=$F9 (function) C=Value Index (HCB Offset) DE=Value | Output |
| Write the HBIOS Control Block value specified in DE at the index specified in C | |

Memory Layout Detail to help understand the processes.

## 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 page 1 to RAM page 1. 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.

The ROM Boot process is implemented in the source file “bootrom.asm”.

## 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 page 1, 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.

This process is implemented in the source file “bootapp.asm”

# 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.
3. There are still some places in the CBIOS where it is manipulating memory banks directly. This is inappropriate and will eventually be corrected.

# 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. 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 – CIOIN Character Input Status – CIOIST Character Output Status – CIOOST Character I/O Configuration – CIOCFG |
| Disk Input/Output (DIO) | Disk Read – DIORD Disk Write – DIOWR Disk Status – DIOST Disk Media – DIOMED Disk Identify – DIOID Disk Get Buffer Address – DIOGETBUF Disk Set Buffer Address – DIOSETBUF |
| 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 |
| Emulation (EMU) | Emulation Input – EMUIN Emulation Output – EMUIN Emulation Input Status – EMUIST Emulation Output Status – EMUOST Emulation Initialization – EMUINI Emulation Query – EMUQRY |
| 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 Set Bank – SYSSETBNK System Get Bank – SYSGETBNK System Copy – SYSCPY System Extended Copy – SYSXCPY System Get Version – SYSGETVER System HCB Get Byte – SYSHCBGETB System HCB Put Byte – SYSHCBPUTB System HCB Get Word – SYSHCBGETW System HCB Put Word – SYSHCBPUTW |

## Character Input/Output (CIO)

Character input/output functions require that a character device/unit be specified in the C register. The upper nibble (upper 4 bits) specify the device (such as UART). The lower nibble specifies the unit of the device (0=first port, 1=second port, etc.)

The CRT device is a virtual device code that will route characters in/out via the currently active VDA device (see EMUINI function).

The currently supported devices/units are:

|  |  |  |
| --- | --- | --- |
| **Device** | | **Unit** |
| 0 | UART | Unit = Port |
| 1 | ASCI | Unit = Port |
| 2 | PropIO VGA | N/A |
| 3 | ParPortProp VGA | N/A |
| F | CRT | N/A |

#### Character Input – CIOIN ($00)

|  |  |
| --- | --- |
| Input B=$00 (function) C=Device/Unit | Output 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)

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

#### Character Input Status – CIOIST ($02)

|  |  |
| --- | --- |
| Input B=$02 (function) C=Device/Unit | Output A=Status: # characters in input buffer |
| Return the number of characters available to read in the input buffer of the device/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 a character available to read. | |

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

|  |  |
| --- | --- |
| Input B=$03 (function) C=Device/Unit | Output A=Status: output buffer space available |
| 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. | |

#### Character Config – CIOCFG ($04)

|  |  |
| --- | --- |
| Input B=$04 (function) C=Speed  E=Framing/Parity | Output A=Status: 0=Success, otherwise failure |
| Not yet implemented.  Sets the speed and framing of the character stream. Register C specifies the speed. Register E specifies the framing and parity characteristics. | |

## Disk Input/Output (DIO)

Disk input/output functions require that a disk device/unit be specified in the C register. The upper nibble (upper 4 bits) specify the device (such as IDE). The lower nibble specifies the unit of the device (0=master, 1=slave, etc.)

The currently supported devices/units are:

|  |  |  |
| --- | --- | --- |
| **Device** | | **Unit** |
| 0 | Memory Disk | Unit 0 = ROM, Unit 1 = RAM |
| 1 | Floppy Disk | Unit 0 = Primary, Unit 1 = Secondary |
| 2 | RAM Floppy | Unit 0 = First Board, Unit 1 = Second Board |
| 3 | IDE Disk | Unit 0 = Master, Unit 1 = Slave |
| 4 | ATAPI Disk (not implemented) | Unit 0 = Master, Unit 1 = Slave |
| 5 | PPIDE Disk | Unit 0 = Master, Unit 1 = Slave |
| 6 | SD Card | N/A |
| 7 | PropIO SD Card | N/A |
| 8 | ParPortProp SD Card | N/A |
| 9 | SIMH HDSK Disk | Unit 0-7 = SIMH emulated hard disk 0-7 |

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 | Hard Disk (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 Read – DIORD ($10)

|  |  |
| --- | --- |
| Input B=$10 (function) C=Device/Unit HL=Track D=Head E=Sector | Output A=Status (0=OK, 1=Error) |
| Read a single 512 byte sector into the buffer previously specified buffer area (see DIOSBA).  For a hard disk device, only LBA addressing is supported. In this case, HL will contain the high 16 bits of the LBA block number and DE will contain the low 16 bits of the LBA block number. | |

#### Disk Write – DIOWR ($11)

|  |  |
| --- | --- |
| Input B=$11 (function) C=Device/Unit HL=Track D=Head E=Sector | Output A=Status (0=OK, 1=Error) |
| Write a single 512 byte sector from the buffer previously specified buffer area (see DIOSBA).  For a hard disk device, only LBA addressing is supported. In this case, HL will contain the high 16 bits of the LBA block number and DE will contain the low 16 bits of the LBA block number. | |

#### Disk Status – DIOST ($12)

|  |  |
| --- | --- |
| Input B=$12 (function) C=Device/Unit | Output A=Status (0=OK, 1=Error) |
| Return the current status of the specified device. | |

#### Disk Media – DIOMED ($13)

|  |  |
| --- | --- |
| Input B=$13 (function) C=Device/Unit | Output A=Media ID |
| Return a media identifier that describes the media format of the current media in the device. If the device supports multiple media types, the media will be examined to determine the specific media format currently installed. | |

#### Disk Identify – DIOID ($14)

Not implemented

#### Disk Get Buffer Address – DIOGETBUF ($18)

|  |  |
| --- | --- |
| Input B=$18 (function) HL=Buffer Address | Output A=Status (0-OK, 1=Error) |
| Get the current buffer address used for disk read/write calls. | |

#### Disk Set Buffer Address – DIOSETBUF ($19)

|  |  |
| --- | --- |
| Input B=$19 (function) HL=Buffer Address | Output A=Status (0-OK, 1=Error) HL=Buffer Address |
| Set the buffer address to be used for subsequent disk read/write calls. Contents of any prior buffer location are not retained. The new buffer area is not initialized. If the buffer address specified is in the lower 32K of RAM, it will be assumed to reside in the HBIOS bank.  If the incoming Buffer Address specified is zero (HL == 0), then this function will reset the buffer address to the default, internal, pre-allocated buffer area in the HBIOS bank and this address will be returned in HL. | |

***Disk Device Count – DIODEVCNT ($1A)***

|  |  |
| --- | --- |
| Input B=$1A (function) | Output A=Status (0-OK, 1=Error) B=Device Count |
| Returns a count of the disk devices in B. | |

***Disk Device Information – DIODEVINF ($1B)***

|  |  |
| --- | --- |
| Input B=$1B (function) C=Device Index | Output A=Status (0-OK, 1=Error) C=Device/Unit |
| This function is used to retrieve information about a specified device from the device list. When called, register C must contain the index into the device list. On return, register C will contain the device/unit code of the device. | |

## 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)

|  |  |
| --- | --- |
| Input B=$20 (function)  HL=Time Buffer Address | Output 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)

|  |  |
| --- | --- |
| Input B=$21 (function) | Output 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)

|  |  |
| --- | --- |
| Input B=$22 (function)  C=Index | Output 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)

|  |  |
| --- | --- |
| Input B=$23 (function)  C=Index | Output 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)

|  |  |
| --- | --- |
| Input B=$24 (function)  HL=Buffer | Output 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)

|  |  |
| --- | --- |
| Input B=$25 (function)  HL=Buffer | Output 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. | |

## Emulation (EMU)

The Emulation functions allow setting up the desired emulation (terminal type) as well as the target physical device for emulation. It is not possible to maintain multiple independent emulation states for different physical devices – emulation must be reinitialized to target a new physical device.

#### Emulation Input – EMUIN ($30)

|  |  |
| --- | --- |
| Input B=$30 (function) | Output A=Status (0=OK, 1=Error) E=Character input |
| Wait for a single character to be available at the emulation target device and return the character in E. Function will wait indefinitely for a character to be available. | |

#### Emulation Output – EMUOUT ($31)

|  |  |
| --- | --- |
| Input B=$31 (function) E=Character to output | Output A=Status (0=OK, 1=Error) |
| Wait for emulation target device/unit to be ready to send a character, then send the character specified in E. | |

#### Emulation Input Status – EMUIST ($32)

|  |  |
| --- | --- |
| Input B=$32 (function) | Output A=Status: # characters in input buffer |
| Return the number of characters available to read in the input buffer of the emulation target device/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 a character available to read. | |

#### Emulation Output Status – EMUOST ($33)

|  |  |
| --- | --- |
| Input B=$33 (function) | Output A=Status: output buffer space available |
| 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 emulation target device 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. | |

#### Emulation Initialization –EMUINI ($38)

|  |  |
| --- | --- |
| Input B=$38 (function) C=VDA Device/Unit  E=Terminal Type | Output A=Status: 0=Success, otherwise failure |
| Selects the actual VDA device/unit to be targeted for emulation.. Register C is set to the VDA device/unit to be selected. Register E specifies the terminal emulation to be used (0=TTY, 1=ANSI). | |

#### Emulation Query –EMUQRY ($39)

|  |  |
| --- | --- |
| Input B=$39 (function) | Output A=Status: 0=Success, otherwise failure  C=VDA Device/Unit  E=Terminal Emulation |
| Returns current information about the active emulation session. Register C is set to the VDA device/unit currently targeted. Register E returns the terminal emulation in use (0=TTY, 1=ANSI). | |

## 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 –VDAINI ($40)

|  |  |
| --- | --- |
| Input B=$40 (function)  C=Device/Unit  E=Video Mode (device specific)  HL=Character Bitmap (optional) | Output A=Status: 0=Success, otherwise failure |
| 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.  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). | |

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

|  |  |
| --- | --- |
| Input B=$41 (function)  C=Device/Unit  HL=Character Bitmap Data (optional) | Output A=Status: 0=Success, otherwise failure  C=Video Mode  D=Row Count  E=Column Count  HL=Character Bitmap Data (zero if none) |
| 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 **count** of rows and columns, not the **last** row/column number.  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. | |

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

|  |  |
| --- | --- |
| Input B=$42 (function)  C=Device/Unit | Output A=Status: 0=Success, otherwise failure |
| 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. | |

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

|  |  |
| --- | --- |
| Input B=$43 (function)  C=Device/Unit  D=Start/End pixel  E=Style | Output A=Status: 0=Success, otherwise failure |
| 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.  Register E is reserved to control the style of the cursor (blink, visibility, etc.), but is not yet implemented.  Adjustments to the cursor style may or may not be possible for any given video hardware. | |

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

|  |  |
| --- | --- |
| Input B=$44 (function)  C=Device/Unit  D=Row  E=Column | Output 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)

|  |  |
| --- | --- |
| Input B=$45 (function)  C=Device/Unit  E=Character Attribute Code | Output 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)

|  |  |
| --- | --- |
| Input B=$46 (function)  C=Device/Unit  E=Color Code | Output 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)

|  |  |
| --- | --- |
| Input B=$47 (function)  C=Device/Unit  E=Character | Output 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 **not** scroll if the end of the screen is exceeded. | |

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

|  |  |
| --- | --- |
| Input B=$48 (function)  C=Device/Unit  E=Character  HL=Count | Output A=Status: 0=Success, otherwise failure |
| 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 **not** scroll if the end of the screen is exceeded. | |

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

|  |  |
| --- | --- |
| Input B=$48 (function)  C=Device/Unit  D=Source Row  E=Source Column  L=Count (max 255) | Output A=Status: 0=Success, otherwise failure |
| 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 **not** scroll if the end of the screen is exceeded. Copying beyond the active screen buffer area is not supported and results in undefined behavior. | |

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

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

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

|  |  |
| --- | --- |
| Input B=$4A (function)  C=Device/Unit | Output A=Status: # key codes in keyboard buffer |
| 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. | |

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

|  |  |
| --- | --- |
| Input B=$4B (function)  C=Device/Unit | Output 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)

|  |  |
| --- | --- |
| Input B=$4C (function)  C=Device/Unit | Output A=Status: 0=Success, otherwise failure  C=Scancode  D=Keystate  E=Keycode |
| 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.  The scancode value is the raw scancode from the keyboard for the keypress. Scancodes are from scancode set 2 standard.  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:  Bit 7: Set to indicate key pressed was from the num pad Bit 6: Set to indicate Caps Lock was active Bit 5: Set to indicate Num Lock was active Bit 4: Set to indicate Scroll Lock was active Bit 3: Set to indicate Windows key was held down Bit 2: Set to indicate Alt key was held down Bit 1: Set to indicate control key was held down Bit 0: Set to indicate Shift key was held down  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. | |

## System (SYS)

#### System Set Bank – SYSSETBNK ($F0)

|  |  |
| --- | --- |
| Input B=$F0 (function) C=Bank ID | Output A=Previously active Bank ID |
| Activates the Bank ID specified in C and returns the previously active Bank ID in A. The caller MUST be invoked from code located in the upper 32K and the stack MUST be in the upper 32K. | |

#### System Get Bank – SYSGETBNK ($F1)

|  |  |
| --- | --- |
| Input B=$F1 (function) | Output A=Active Bank ID |
| Returns the currently active Bank ID in A. | |

#### System Memory Copy – SYSCPY ($F2)

|  |  |
| --- | --- |
| Input B=$F2 (function) DE=Destination address HL=Source address IX=Count of bytes to copy | Output A=Status: 0=Success, otherwise failure |
| This function will perform a bulk memory copy. If SYSXCPY is invoked before this function, the memory copy will performed to/from the memory banks specified in the SYSXCPY function. Otherwise, the memory will be copied to/from the default user memory bank (typically TPA).  WARNING: The memory copy is performed from low byte to high byte, so be careful of a memory copy where the source range overlaps the destination range.  WARNING: Directly manipulating memory banks can easily corrupt critical areas of the system.  WARNING: This function uses the IX register as a parameter. The caller should save the original value of IX and restore it upon return from this function to comply with normal CP/M register handling expectations. | |

#### System Extended Memory Copy – SYSXCPY ($F3)

|  |  |
| --- | --- |
| Input B=$F3 (function) D=Destination Bank ID E=Source Bank ID | Output A=Status: 0=Success, otherwise failure |
| This function controls the source and destination banks to be used by the SYSCPY function. The source bank is set using E and the destination bank is set using D.  Note that the SYSCPY function will reset the source and destination bank numbers after it is called, so this function must be called prior to every invocation of SYSCPY that needs to perform an interbank memory copy. | |

#### System Get Version – SYSGETVER ($F6)

|  |  |
| --- | --- |
| Input B=$F6 (function) | Output A=Status: 0=Success, otherwise failure  DE=Version |
| 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. | |

***System HCB Get Byte – SYSHCBGETB ($F8)***

|  |  |
| --- | --- |
| Input B=$F8 (function) C=Value Index (HCB Offset) | Output E=Value |
| Retrieve the HBIOS Control Block value at the index specified in C and return the byte value in E. | |

***System HCB Put Byte – SYSHCBPUTB ($F9)***

|  |  |
| --- | --- |
| Input B=$F9 (function) C=Value Index (HCB Offset) E=Value | Output |
| Write the HBIOS Control Block value specified in E at the index specified in C | |

***System HCB Get Word – SYSHCBGETB ($FA)***

|  |  |
| --- | --- |
| Input B=$F8 (function) C=Value Index (HCB Offset) | Output DE=Value |
| Retrieve the HBIOS Control Block value at the index specified in C and return the word value in DE. | |

***System HCB Put Word – SYSHCBPUTB ($FB)***

|  |  |
| --- | --- |
| Input B=$F9 (function) C=Value Index (HCB Offset) DE=Value | Output |
| Write the HBIOS Control Block value specified in DE at the index specified in C | |

# Memory Layout Detail

#### ROM Page 0

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Loc** | **Org** | **Size** | **Source** | **Contents** |
| 0000 | 0000 | 0100 | pgzero.asm | Page Zero |
| 0100 | 0100 | 0100 | bootrom.asm | ROM Bootstrap |
| 0200 | 0100 | 0200 | syscfg.asm | System Configuration |
| 0400 | 8400 | 0C00 | loader.asm | Loader |
| 1000 | 1000 | 3000 | romfill.asm | Filler |
| 4000 | C000 | 1000 | dbgmon.asm | Debug Monitor |
| 5000 | D000 | 0800 | <ccp> | Command Processor (CCP, ZCPR, etc.) |
| 5800 | D800 | 0E00 | <dos> | Disk Operating System (BDOS, ZSDOS, etc.) |
| 6600 | E600 | 1600 | <osbios> | OS BIOS (CBIOS, ZBIOS) |
| 7C00 | FC00 | 0400 | hbfill | Filler for HBIOS Proxy |

#### ROM Page 1

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Loc** | **Org** | **Size** | **Source** | **Contents** |
| 0000 | 0000 | 0100 | pgzero.asm | Page Zero |
| 0100 | 0100 | 0100 | bootrom.asm | Reserved (unused) |
| 0200 | 0200 | 0200 | syscfg.asm | System Configuration |
| 0400 | 0400 | 0C00 | loader.asm | Reserved (unused) |
| 1000 | 1000 | 7000 | bnk1.asm | Bank 1 HBIOS Extension (drivers) |

#### COM File Image

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Loc** | **Org** | **Size** | **Source** | **Contents** |
| 0100 | 0100 | 0100 | bootapp.asm | Application Bootstrap |
| 0200 | 0200 | 0200 | syscfg.asm | System Configuration |
| 0400 | 8400 | 0C00 | loader.asm | Loader |
| 1000 | 1000 | 7000 | bnk1.asm | Bank 1 HBIOS Extension (drivers) |
| 8000 | C000 | 1000 | dbgmon.asm | Debug Monitor |
| 9000 | D000 | 0800 | <ccp> | Command Processor (CCP, ZCPR, etc.) |
| 9800 | D800 | 0E00 | <dos> | Disk Operating System (BDOS, ZSDOS, etc.) |
| A600 | E600 | 1600 | <osbios> | OS BIOS (CBIOS, ZBIOS) |
| BC00 | FC00 | 0400 | hbios.asm | HBIOS Proxy |

#### RAM Page 0 (Applications)

|  |  |  |  |
| --- | --- | --- | --- |
| **Loc** | **Org** | **Size** | **Contents** |
| 0000 | 0000 | 0100 | Page Zero |
| 0100 | 0100 | 7F00 | Application (TPA) |

#### RAM Page 1 (HBIOS Extension – Drivers)

|  |  |  |  |
| --- | --- | --- | --- |
| **Loc** | **Org** | **Size** | **Contents** |
| 0000 | 0000 | 0100 | Page Zero |
| 0100 | 0100 | 0100 | Reserved (unused) |
| 0200 | 0200 | 0200 | System Configuration (dynamic) |
| 0400 | 0400 | 0C00 | Command processor cache area |
| 1000 | 1000 | 7000 | HBIOS |

#### RAM Page N (Fixed 32K Upper Memory Area)

|  |  |  |  |
| --- | --- | --- | --- |
| **Loc** | **Org** | **Size** | **Contents** |
| 8000 | 8000 | 4000 | TPA (continued from lower memory) |
| C000 | C000 | 1000 | TPA/Debug Monitor |
| D000 | D000 | 0800 | Command Processor (CCP, ZCPR, etc.) |
| D800 | D800 | 0E00 | Disk Operating System (BDOS, ZSDOS, etc.) |
| E600 | E600 | 1600 | OS BIOS (CBIOS, ZBIOS) |
| FC00 | FC00 | 0400 | HBIOS Proxy (HiMem Stub) |