|  |
| --- |
| RomWBW Architecture |
| N8VEM Project |
| September 25, 2012 |

Contents

[Background 2](#_Toc336370617)

[General Design Strategy 2](#_Toc336370618)

[Runtime Memory Layout 3](#_Toc336370619)

[System Boot Process 4](#_Toc336370620)

[Notes 4](#_Toc336370621)

[Driver Model 4](#_Toc336370622)

[HBIOS Functions 5](#_Toc336370623)

[Invocation 5](#_Toc336370624)

[Function Overview 5](#_Toc336370625)

[Character Input/Output (CIO) 5](#_Toc336370626)

[Disk Input/Output (DIO) 7](#_Toc336370627)

[Real Time Clock (CLK) 9](#_Toc336370628)

[Video Display Unit (VDU) 9](#_Toc336370629)

[System (SYS) 12](#_Toc336370630)

[Memory Layout Detail 14](#_Toc336370631)

# 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 of 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 two 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.

Phase 1 of booting copies phase 2 code to upper memory and jumps to it to continue the boot process.

Phase 2 of booting manages the setup of the RAM page banks as needed. In the case of a hardware startup, phase 2 just copies the code from ROM page 1 into RAM page 1 and executes the loader. In the case of an application startup (.com file used to load a new copy of the system), phase 2 copies the first 32KB of the application memory space into RAM page 1 and executes the loader.

See 'bootrom.asm' for the implementation of the ROM (hardware) startup. See 'bootapp.asm' for the implementation of the application based startup.

# 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.

# HBIOS Functions

## 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 does not modify the Z80 alternate registers or IX/IY.

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.

## Function Overview

|  |  |
| --- | --- |
| Character Input/Output (CIO) | Character Input – CIOIN  Character Output – CIOIN  Character Input Status – CIOIST  Character Output Status – CIOOST |
| Disk Input/Output (DIO) | Disk Read – DIORD  Disk Write – DIOWR  Disk Status – DIOST  Disk Media – DIOMED  Disk Identify – DIOID  Disk Get Buffer Address – DIOGBA  Disk Set Buffer Address – DIOSBA |
| Real Time Clock (CLK) | Not Implemented |
| Video Display Unit (VDU) | Not Implemented |
| System (SYS) | Get Configuration – GETCFG  Set Configuration – SETCFG  Banked Memory Copy – BNKCPY |

## 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 currently supported devices/units are:

|  |  |  |
| --- | --- | --- |
| **Device** | | **Unit** |
| 0 | UART | Unit = Port (eg. 0=ASCI0, 1=ASCI1, etc.) |
| 1 | PropIO VGA | N/A |
| 2 | ECB VDU | N/A |
| 3 | ECB Color VDU (not implemented) | N/A |
| 4 | ParPortProp VGA | 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=$02 (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. | |

## 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 | IDE Disk | Unit 0 = Master  Unit 1 = Slave |
| 3 | ATAPI Disk (not implemented) | Unit 0 = Master  Unit 1 = Slave |
| 4 | IDE Disk | Unit 0 = Master  Unit 1 = Slave |
| 5 | SD Card | N/A |
| 6 | PropIO SD Card | N/A |
| 7 | ParPortProp SD Card | N/A |
| 8 | 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\_HD | 3 | Hard Disk (LBA) |
| MID\_FD720 | 4 | 3.5” 720K Floppy |
| MID\_FD144 | 5 | 3.5” 1.44M Floppy |
| MID\_FD360 | 6 | 5.25” 360K Floppy |
| MID\_FD120 | 7 | 5.25” 1.2M Floppy |
| MID\_FD111 | 8 | 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 (seeDIOSBA).  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 (seeDIOSBA).  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 – DIOGBA ($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 – DIOSBA ($19)

|  |  |
| --- | --- |
| Input B=$19 (function) HL=Buffer Address | Output A=Status (0-OK, 1=Error) |
| 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. The buffer must be located in high memory (top 32K). | |

## Real Time Clock (CLK)

This function category is not yet implemented.

## Video Display Unit (VDU)

The VDU functions are provided as a common interface to CRT/Keyboard devices. Not all video devices will include keyboard hardware. In this case, the keyboard functions should return a failure status.

The VDU functions require that a video device be specified in the C register. The upper nibble (upper 4 bits) specifies the device. The lower nibble is reserved for future use.

The currently defined video devices are:

|  |  |  |
| --- | --- | --- |
| **VDU ID** | **Value** | **Device** |
| VDU\_NONE | 0 | No video device |
| VDU\_VDU | 1 | ECB VDU board |
| VDU\_CVDU | 2 | ECB Color VDU board |
| VDU\_7220 | 3 | ECB uPD7220 video display board |
| MID\_N8 | 4 | TMS9918 video display built-in to N8 |

#### Video Initialize –VDUINI ($30)

|  |  |
| --- | --- |
| Input B=$30 (function)  C=Device/Unit | 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. | |

#### Video Query –VDUQRY ($31)

|  |  |
| --- | --- |
| Input B=$31 (function)  C=Device/Unit | Output A=Status: 0=Success, otherwise failure  D=Row Count  E=Column Count |
| 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. | |

#### Video Set Cursor Style –VDUSCS ($32)

|  |  |
| --- | --- |
| Input B=$32 (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 Set Cursor Position –VDUSCP ($33)

|  |  |
| --- | --- |
| Input B=$33 (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 Set Character Attribute –VDUSAT ($34)

|  |  |
| --- | --- |
| Input B=$34 (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 table XXX for a list of the available color codes. Note that a given video display may or may not support any/all attributes. | |

#### Video Set Character Color –VDUSCO ($35)

|  |  |
| --- | --- |
| Input B=$35 (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 table XXX for a list of the available color codes. Note that a given video display may or may not support any/all colors. | |

#### Video Write Character –VDUWRC ($36)

|  |  |
| --- | --- |
| Input B=$36 (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 Fill –VDUFIL ($37)

|  |  |
| --- | --- |
| Input B=$37 (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 Scroll –VDUSCR ($38)

|  |  |
| --- | --- |
| Input B=$38 (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 Keyboard Status –VDUKST ($39)

|  |  |
| --- | --- |
| Input B=$39 (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 Keyboard Flush –VDUKFL ($3A)

|  |  |
| --- | --- |
| Input B=$3A (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 Keyboard Read –VDUKRD ($3B)

|  |  |
| --- | --- |
| Input B=$3B (function)  C=Device/Unit | Output A=Status: 0=Success, otherwise failure  E=Key code read |
| 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 key code.  Note that this function returns the **key code** that was read, not an ASCII character. See table ??? for the key codes and their meanings. Key codes must be appropriately mapped for case, control, etc. before being used. | |

## System (SYS)

#### Get Configuration – GETCFG ($F0)

|  |  |
| --- | --- |
| Input B=$F0 (function) C=Config Version (not implemented) DE=Destination address | Output A=Status: 0=Success, otherwise failure |
| Copies the 256 byte block of configuration data into the destination memory address specified in DE. The destination memory address must be in high memory (upper 32K). At present, you will need to consult the source code for information on the contents of the configuration block. | |

#### Set Configuration – SETCFG ($F1)

|  |  |
| --- | --- |
| Input B=$F1 (function) C=Config Version (not implemented) DE=Source address | Output A=Status: 0=Success, otherwise failure |
| Loads a 256 byte block of configuration data into the BIOS from the source memory address specified in DE. The source memory address must be in high memory (upper 32K). At present, you will need to consult the source code for information on the contents of the configuration block.  NOTE: At present, the effects of this function are undefined. The BIOS will not dynamically adapt to a changed configuration. | |

#### Banked Memory Copy – BNKCPY ($F2)

|  |  |
| --- | --- |
| Input B=$F2 (function) DE=Destination address HL=Source address IX=Count of byte to copy | Output A=Status: 0=Success, otherwise failure |
| The function will select the requested memory bank into the lower 32K of CPU address space. Then it executes a memory copy from the source address (DE) to the destination address (HL) of count bytes (IX). It then restores the default bank (application memory) to the lower 32K.  The function does not know or care if you are copying to or from or within a bank. It simply selects the bank and performs the copy. To copy "from" a bank, you would specify a source in the lower 32K and a destination in the upper 32K. To copy "to" a bank, you would specify a source in the upper 32K and a destination in the lower 32K.  It is also possible to copy memory around within a bank by specifying a source and destination in the lower 32K.  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. | |

# 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 | 1900 | <osbios> | OS BIOS (CBIOS, ZBIOS) |
| 7F00 | FF00 | 0100 | 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 | 1900 | <osbios> | OS BIOS (CBIOS, ZBIOS) |

#### 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 | HBank 1 BIOS Extension (drivers) |

#### 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 | 1900 | OS BIOS (CBIOS, ZBIOS) |
| FF00 | FF00 | 0100 | HBIOS Proxy (HiMem Stub) |