A Zenith Z-100 Emulator

Joseph Matta

Follow this and additional works at: https://vc.bridgew.edu/theses

Part of the Programming Languages and Compilers Commons

Recommended Citation
Copyright © 2021 Joseph Matta

This item is available as part of Virtual Commons, the open-access institutional repository of Bridgewater State University, Bridgewater, Massachusetts.
A Zenith Z-100 Emulator

A Thesis Presented

By

JOSEPH MATTA

MAY 2021

Approved as to style and content by:

Signature: ____________________________________________

Dr. Michael Black

Date

Signature: ____________________________________________

Dr. Haleh Khojasteh

Date

Signature: ____________________________________________

Dr. John F. Santore

Date
A Zenith Z-100 Emulator

By

JOSEPH MATTA

Submitted to the College of Graduate Studies

Bridgewater State University

Bridgewater, Massachusetts

In partial fulfillment of the requirement for the Degree of

Master of Science

in Computer Science

MAY 2021
Copyright, 2021, Joseph Matta
ABSTRACT

The Zenith Z-100 computer was released by the Zenith Data Systems Corporation in 1982 as a competitor to the IBM PC. There are no known complete software emulations of the system. A Z-100 emulator is considered to be complete if it runs all functions of its monitor ROM BIOS program and is able to boot and run its two operating systems. One reason previous emulation attempts are not complete is that they ineffectively implement the floppy disk controller, preventing a proper transfer of the operating system from disk into memory. This project is an attempt to write a complete emulation of the Z-100 system. Improving on previous emulations, a novel implementation of the Western Digital FD-1797 floppy disk controller is developed. Although the disk controller implementation is successfully integrated, only a partial initialization of the Z-DOS operating system is achieved due to an unsupported interrupt procedure. Booting CP/M-85, the other operating system supported by the Z-100, is not attempted. Rationale for the Z-100’s historical preservation through emulation is also presented.
The Z-100 emulator for this project was developed alongside a detailed examination of its monitor ROM BIOS program. The examination consisted of a nearly line-by-line code trace of the BIOS’s startup initialization and diagnostic procedures. A PDF scan of Volume I and II of the source listings for ROM version 2.5 was used as a reference against an instruction level reading of the ROM through the emulator’s debug features. A binary file (zrom_444_276_1.bin) containing a different ROM version was used, however. The version discrepancy between the reference listing (version 2.5) and the binary file (version 2.9) did not result in significant code incongruencies until the BIOS ‘BOOT’ command section. Calling this section of the code is necessary to boot the operating system from disk and is needed to facilitate transfer of system control from the monitor program to the operation system. As such, the obligate tracing through critical sections of this code was done largely without the guide of the Boot ROM Listing.

The emulator program allows two modes of operation: ‘Normal’ and ‘Debug.’ ‘Normal’ mode runs the emulator without breakpoints. In this mode, minimal system state information is printed to the terminal standard output and no instruction breakpoints are allowed to be set. Alternatively, ‘Debug’ mode allows an instruction count number or instruction pointer value break point. Detailed status information is reported with each instruction step including the active processor state, memory locations accessed, and port input/output. With this debug functionality, each ROM instruction involved in the BIOS’s startup
initialization and boot sequence was executed, examined, and compared to the corresponding line in the ROM listing documents.

The instruction step and ROM documentation comparison allowed for a better understanding of the expected monitor program behavior. For example, if the monitor program is expecting specific feedback from a port and the device associated with that port has not been properly implemented, the monitor code trace diverts into an error-handling subroutine. With the help of the ROM source listing documents, code branches and subroutines were more easily recognized. Code comments and section headings in the ROM listing provided valuable information about where the code was heading and what instructions should execute if there was an error.

Sequentially stepping through each individual instruction of the monitor program while simultaneously verifying each instruction with the ROM listing documents proved to be an effective and robust strategy to develop the emulator. Components were systematically added as the monitor program tried to access them. This helped with understanding how the components should behave and made sure the component was properly initialized. Successfully traversing through the initialization and diagnostic routines without errors indicated that the components were implemented correctly from the perspective of the monitor program and therefore would likely function as expected in an operating system environment. A detailed journal was kept throughout the entire ROM examination process.

The bulk of this document describes the implementation details of the
emulator. It examines the emulator’s code from the program’s launch until the last procedure of the main processor loop. Rationale and justification for each part of the reviewed code is presented. It is recommended that the reader follow along with the discussion of code segments by looking at the code base while reading. The code for this project is written entirely in the C programming language and is compiled using a makefile. Development was conducted in a Linux operating system environment using Ubuntu 18.04.5 LTS emulated inside VirtualBox Version 6.1.10. Project code for the Z-100 emulator can be found at the following GitHub repository:

https://github.com/jmatta697/Custom_Z100_Emulator_Mod.git

The project is compiled and run using the following commands at the Linux terminal command prompt after navigating to its project folder:

$ make

$ ./jz100

For a clean compile:

$ make clean

A custom implementation of the Western Digital FD-1797 floppy disk drive controller was developed along with its own testing environment. The test version code for the disk controller implementation can be found at the following GitHub repository:
The FD-1797 controller project is compiled and run using the following commands at the Linux terminal command prompt after navigating to its project folder:

```bash
$ make

$ ./test_jwd
```

For a clean compile use:

```bash
$ make clean
```

To run tests of the FD-1797 implementation, run "./test_jwd" from the Linux command line after compilation. The automated tests run sequentially. Test function definitions are found in the testFunctions.c file. Testing was used for development purposes only and is not optimized for easy user interaction. However, there is a pause between each test to provide the user with an opportunity to investigate the previous test’s output. Chapter IV of this document discusses the FD-1797 implementation tests in detail.

Some points should be made about the style in which this document is written. Citations are given in the standard IEEE format. However, some citations are provided parenthetically at the end of sentences. This style mostly applies to page references from the Z-100 technical manuals, but also applies to pages referenced from the monitor ROM source listing volumes. For source listing
references, the page, code section, and volume are stated. For example, the reference “page 1-17 – MTR100 – Z100 ROM Listing 2.5 Volume I” indicates the information comes from page 1-17 of the MTR100 code section of volume 1 of the ROM listing. It was decided that a new “References” section entry for every page reference was too redundant and would cause a cumbersome reference page. Also, it was thought that the reader would be better served if the page source were seen directly next to the relevant information.

Function names appear as italicized text with a parenthesis pair ("()") attached to the end of their names to clearly indicate that the names refer to functions. The function parameters and their data types are left out of the parentheses to reduce cluttering the text.

Variable names appear in quotes as do some data types. Context distinguishes which names are data types and which ones refer to variables.

Instances of C structs are referred to “objects” for simplicity, and struct variables are referred to as instance variables or sometimes “fields.”

Numerical values in base 16 (hexadecimal) are indicated with ‘0x’ preceding the value. For example, the base-10 value ‘130’ would be written as ‘0x82.’ Binary values are indicated with ‘0b’ preceding the value. As such, the base-10 value ‘130’ would be represented in binary as ‘0b10000010.’

Logical operation names such as ‘AND’ and ‘OR’ are capitalized as shown.

BIOS command names are also capitalized (e.g. BOOT).
File names are expressed with their file type extensions attached (e.g. video.c).

Explaining where a specific function is defined within the project is done by expressing that the function “belongs” to a certain file. For example, the expressions “mainBoard.c’s handleDebugOutput() function” or “the handleDebugOutput() function of mainBoard.c” both mean that the handleDebugOutput() function is defined in the mainBoard.c file.

The overall research strategy used in this project is summarized as follows:

1. Examine previous Z-100 emulator attempts.
2. Develop an emulator from scratch by systematically tracing through the ROM BIOS diagnostic and initialization startup source code.
   Add components and deal with errors as they occur.
3. Develop and test a novel FD-1797 floppy disk controller in isolation.
4. Incorporate the FD-1797 controller implementation into the emulator.
5. Trace the Z-100’s BIOS BOOT command source code until the Z-DOS operating system takes control. (The BOOT command makes use of the disk controller to load the operating system.)
6. Examine the Z-DOS initialization procedures.
Outcomes and contributions resulting from this research are described below:

- An exhaustive, detailed study of the Z-100 ROM version 2.9 initialization and diagnostics startup source code was conducted.
- All BIOS commands function as expected in this project's Z-100 emulator without any errors. No previous emulator attempt accomplishes this. (Exceptions: Composite color pixels in the COLOR command output are not displaying correctly. Additionally, screen scrolling is not implemented in this project.)
- A novel FD-1797 floppy disk controller implementation is tested and successfully integrated into this project's Z-100 emulator. Existing emulators do not have working disk drive controller implementations.
- Z-DOS initialization takes system control from the BIOS after issuing the BOOT command. Although the initialization process hangs at an interrupt detection loop and does not reach the "A:" prompt, it advances further than all other existing emulators.
TABLES AND FIGURES

Figure 1. The Z-100 All-in-One model with built-in monitor (upper) and the Low-Profile model (lower)  

Figure 2. Hand prompt of the Z-100 monitor program contained on its ROM  

Figure 3. Z-100 screen simulated by Rick Button's C# emulator. The hand prompt is shown, and the number displayed in the window title is indicated with a red box and arrow.  

Figure 4. Debug console window of Rick Button's C# Z-100 Emulator  

Figure 5. Register debug window of Rick Button's C# Z-100 emulator  

Figure 6. MS-DOS System Configuration screen that appears when running the BOOT command in Rick Button's C# Z-100 emulator.  

Figure 7. The Z-100 monitor program TEST menu  

Figure 8. MESS main emulator menu showing the Z-100 highlighted and displaying the message that it is not working  

Figure 9. MESS dialogue box stating that the Z-100 emulator does not work  

Figure 10. MESS system description for its Z-100 emulator  

Figure 11. The MESS Z-100 monitor program prompt with error message shown above the prompt  

Figure 12. Static hand prompt with "DEFAULT DEVICE" (?) and debug information displayed above the prompt  

Figure 13. Hexedit inspection of zrom_444_276_1.bin showing its last byte as byte number 0x3FFF
Figure 14. Main entry menu of the JZ-100 emulator. Option 2 (DEBUG) is selected, and the DEBUG mode menu is presented.

Figure 15. System state information after instruction number 1000 has been executed while in debug mode 1. System state information is displayed after each instruction step.

Figure 16. Code block connecting callback functions to the GTK drawing area and window. These lines appear in the `screenInit()` function which is defined in screen.c.

Figure 17. Code block found in the `main()` function of mainBoard.c. These lines set up and start the two threads that comprise the Z-100 emulator.

Figure 18. Excerpt from page 2.12 of the Z-100 Technical Manual (Hardware) showing the possible ROM configurations based on bits 2 and 3 of the memory control latch port (0xFC) byte.

Figure 19. Arrangement of the master/slave PIC circuitry in the Z-100 (page 2.21 - Z-100 Technical Manual - Hardware).

Figure 20. Summary of gate pin operations for counter modes of the Intel 8253 timer chip (Page D.98 - Z100 Technical Manual – Appendices).

Figure 21. Setup of the Z-100’s 8253 timer and surrounding circuitry (page 2.15 - Z100 Technical Manual - Hardware).

Figure 22. Acceptable FD-1797 disk controller track formatting (page D.200, Z100 Technical Manual - Appendices).

Figure 23. IBM modified frequency modulation (MFM), double density (DD) format for 5.25" floppy disks (DOS) reported by The HP 9845 Project.

Figure 24. The first 64 (0x40) bytes of "z-dos-1.img" as shown by the hexedit utility - Loader disk parameter bytes are contained within the red border.

Figure 25. Z-100 main board schematic showing connection between the 8085’s READY pin and the WAIT LOGIC circuitry. The connection is highlighted in blue.
Figure 26. Head load timing of the FD-1797 disk controller (page D.191, Z-100 Technical Manual - Appendices)

Figure 27. General status port bit definitions of the FD-1797 disk controller (page 6.10, Z-100 Technical Manual - Hardware)

Figure 28. The Z-100 standard character display scheme. Each character is 8 pixels wide and 9 pixels high. The displayable area consists of 25 rows of 80 characters.

Figure 29. The Z-100's standard scan line pattern. For each group of 9 scan lines, 7 scan lines worth of VRAM addresses are not displayed.

Figure 30. Placement scheme of the VRAM bits into the "pixels" array. Placement of the first two bytes is shown.

Figure 31. Results of a masterClockTest() run. Since the cumulative sum of the master clock reflects the incoming time slice values, the test is passed.

Figure 32. Results of a getFByteTest() run. The test is passed because all bytes are read as expected.

Figure 33. Command bit patterns for the FD-179X floppy disk controller family. Table B, indicated by the red box, applies to the FD-1797 model.

Figure 34. Two RESTORE command variation test code blocks (left) and successful results reported by the commandWriteTests() function (right)

Figure 35. Two FORCED INTERRUPT command test code blocks within the commandWriteTests() function (left) and their resulting terminal output (right). These tests are successful because the reported FD-1797 object state variables match the expected output.

Figure 36. The 5.25" floppy disk index hole is indicated by the red arrow (left). The index hole marks the beginning of tracks. The start of all tracks is indicated by the red line positioned next to the index hole (right).

Figure 37. Highlighted with a red box, bit 1 (S1) of the type I status appearing in the FD-1797's status register indicates an active index pulse. If the bit has a value of 1, the index pulse is active.
Figure 38. Terminal test output for the `indexPulseTest()` function showing the start of the index pulse and the IP forced interrupt condition being met (left). Terminal test output for the `indexPulseTest()` function showing the expiration of the index pulse timer and the transition from an active IP to inactive via bit 1 of the status register (right).

Figure 39. Terminal output for the `restoreCommandTest()` when starting from track 3. Beginning at the top left pane, cylinder transitions are shown for each transition from cylinder 3 to 0. The bottom panes show the verify delay timer expiration (left) and the head load timer expiration (right). These results demonstrate a successful RESTORE command test.

Figure 40. Terminal output for the `seekCommandTest()` function. From the left, successful step transitions from track 7 to 6 and then from track 6 to 5 are shown. The last pane to the right shows the verify delay timer expiring 30 milliseconds after the completion of the SEEK procedure. This output indicates a successfully passed test of the command.

Figure 41. Terminal output for the `stepCommandTest()` function showing the successful transition from track 6 to 7 (left); `stepInCommandTest()`’s successful transition from track 5 to 6 (middle); `stepOutCommandTest()`’s successful transition from track 5 to 4 (right). In each command test output, the “direction” value (indicted by a red arrow) and the updated track register (red box) are reported correctly.

Figure 42. The READ SECTOR command being tested in the `readSectorTest()` function. This terminal output shows a FORCED INTERRUPT command (interrupt immediately) being issued while the READ SECTOR command is active. The command is interrupted and deactivated, thus demonstrating a successful FORCED INTERRUPT command implementation.

Figure 43. Terminal output for the READ SECTOR test function `readSectorTest()`. The command variation uses the multiple record mode and 30-ms delay. Since the command starts at sector 7 of an 8-sector track, the multiple record mode causes a read of sector 7 (left two panes) and sector 8 (right two panes) until the end of the track. Sector numbers are highlighted in red. The figure only shows the first four and last four bytes read from the sectors. However, all bytes are confirmed and thus this test indicates a properly implemented READ SECTOR command.
Figure 44. Terminal output for a successful test of the READ SECTOR command with single record mode enabled and head 1 (disk side 1) selected. The head selection ID field data is highlighted with a yellow box. A head selection ID field value of 0x00 corresponds to head 1/side 1. The target sector of 3 is highlighted in the ID field data output. Although only the first four and last four bytes are shown to be confirmed, the full test output does correctly confirm all bytes. These results indicate a successful test of this READ SECTOR command variation.

Figure 45. Location of the six address ID field bytes within a DOS-formatted track is highlighted by a red box (top). The ID byte section is magnified to clearly show the meaning of each byte.

Figure 46. Final terminal test outputs for the readAddressTest() function. The left pane shows the result of a READ ADDRESS command issued when the head is positioned above disk cylinder 2. The right pane is output resulting when the head is over cylinder 6 and head 2 (side 2) is active.

Figure 47. Terminal test output for the READ TRACK command tests run in the readTrackTest() function. The last five byte reads and comparison results are shown for each test. Results displayed in the top pane are from the test targeting formatted track number 10. The bottom pane’s output is from the test targeting formatted track 63.

Figure 48. Result of the BOOT command for this Z-100 emulator when attempting to boot Z-DOS.

Figure 49. Result of forcing the 8088 zero flag to 0 at CS=0x40: IP=0xED6 of the Z-DOS initialization process. This causes the hung-up loop reached when using the BOOT command to exit. However, another error occurs when attempting to load the COMMAND.COM module.
TABLE OF CONTENTS

I. Introduction and Background
   A Brief History of the Zenith Z-100 1
   Historical Preservation of the Z-100 4
   Previous Z-100 Emulation Attempts 6
      Rick Button’s C# Z-100 Emulator 8
      MESS Z-100 Emulator 14
      Michael Black’s Z-100 Emulator 18
   “Complete” Z-100 Emulation 21
   Development Approach and Strategy 23

II. Emulator Initialization
   Emulator Entry 25
   Main Emulator and GTK Screen Threads 29
   Device Object Initialization 33
   Keyboard and Video 34
   I/O Ports 35
   ROM Initialization 39
   Parity Circuity Initialization 41
   Processor Object Initialization 43
   8259 Programmable Interrupt Controllers 47
      Modifying Hug’s 8259 Implementation 53
      Modified e8259_inta() Function 58
   8253 Timer Initialization 61
   FD-1797 Floppy Disk Controller Initialization 67
      Creating a Formatted Disk Array 72
   Counts, Timers, and Process Flags 91
III. Processor Loop

Debug Mode 2 Flag Update 96
Active Processor Selection 97
8085 Wait State Logic 97
8088 Wait State Logic 102
updateElapsedVirtualTime() 104
VSYNC Interrupt 105
8253 Timer Clock Cycle 106
FD-1797 Disk Controller Cycling 109
doJWD1797Cycle() 110
Updating the Z-100 Screen 126
handleDebugOutput() 138

IV. FD-1797 Implementation Testing 141
testMain.c 142
masterClockTest() 143
getFByteTest() 144
commandWriteTests() 146
indexPulseTest() 149
restoreCommandTest() 153
seekCommandTest() 154
stepCommandTest(), stepInCommandTest(),
and stepOutCommandTest() 157
readSectorTest() 160
readAddressTest() 167
readTrackTest() 171

V. Conclusion 175
References
I. Introduction and Background

A Brief History of the Zenith Z-100

The Zenith Z-100 is the general name given to the Z-100 series of personal computers released by Zenith Data Systems in 1982. These models were produced to be direct competitors to the IBM PC. Previously, Zenith acquired the Heathkit Company in 1979 to form Zenith Data Systems. [1] Along with this acquisition, Zenith inherited the H-100, a hobbyist computer system sold as a kit to be assembled by the user. Zenith developed the Z-100 as a preassembled version of the H-100. Two models were offered. The All-in-One model has a built-in monitor, while the Low-Profile model requires an external video device (Figure 1). [2] Less tech savvy consumers now had the opportunity to own an H-100 without the burden of assembling the computer. As a result, a new audience for the system emerged. Clarkson College of Technology (now Clarkson University) issued Z-100 computers to all incoming freshmen in the Fall 1983 semester. [3], [4] Clarkson became one of the first colleges to standard issue computers to students. It was also reported that Drexel University contemplated issuing personal computers to students and looked to Clarkson for their example. [5] The Z-100 therefore became a historically significant part of computing education in the early 1980s.
Technological capabilities of the Z-100 were impressive for the time. Although not rising to the ubiquity of the IBM PC, the Zenith system includes several features that set it apart. Perhaps its most lauded feature is its graphics capability. Featuring a 225 vertical by 640 horizontal resolution screen and a selection of eight different possible color choices per pixel, it had an advantage over the IBM PC’s 220 by 640 black and white only screen graphics. [6] The Z-100’s unique graphics system is also capable of handling multipage and interlaced displays up to resolutions as high as 640 by 525 pixels (page 4.2, Z-100 Technical Manual - Hardware). As a result, the Zenith system was the target for software that benefited from its advanced graphics. Autodesk, a company still well known for its computer-aided design (CAD) software, developed a Z-100 version of its 1982 CAD product, Autocad, taking full advantage of the advanced graphics. [7]
Another unique feature of the Z-100 not seen in many other systems is that it included 8-bit and 16-bit dual processors. It includes the Intel 8085 and the Intel 8088 processor. Although the processors are not capable of running in parallel, programs can use either processor by switching at any point in their execution. [8] The reasoning behind including the dual processors was two-fold. The 8-bit 8085 was used to run the CP/M-85 operating system, which is the Z-100 analog of the popular and familiar CP/M-80 version 2.2. [6] The 16-bit 8088 processor runs the Z-DOS operating system, an MS-DOS analog released by Microsoft specifically for use in the Zenith Z-100. [9] Having the two processors available made the Z-100 a versatile machine; it was able to run older 8-bit programs through the 8085 and newer 16-bit programs with the 8088. The timing of this innovation was convenient because many 8-bit programs were still common while the computing world was transitioning to 16-bit architectures.

The historical place of the Zenith Z-100 can be viewed as one of an underdog product that was in many ways superior to the industry leader, the IBM PC. Keeping focus on the transition from 8-bit to 16-bit architectures, Zenith aimed to make a product that could bridge the gap between these eras. The ability to run programs written for 8-bit and 16-bit architectures gave it an edge. This innovation, together with the newly acquired Heathkit hobbyist audience, contributed to Zenith’s success. Success that is evident by a 60% increase in average sales at Zenith Data Systems between the 1979 Heath acquisition through 1981. [10] In the following years, Zenith aimed to appeal to those looking for small business computing solutions by releasing a new product that was
comparable, and in many ways technologically superior, to the IBM PC. This product was the Z-100.

Considering the financial success attributable to the Z-100’s development and the serendipitous choice by at least one institution of higher education to include it as a standard tool in tech education, the Z-100 can be seen as playing a significant role in computing history. The Z-100 was indeed an interesting outlier among the crowded field of small system competitors vying for a spot in the emerging personal computer market.

Historical Preservation of the Z-100

Emulating an obsolete computer system is important not only for preserving the ability to run software specifically made for that system, but also for preserving the user experience associated with that system. Computer hardware degrades over time and is often discarded when the technology becomes irrelevant. Over time, obsolete computer hardware becomes increasingly scarce and the ability to revisit the associated software and user experience fades away. Emulation is an effective way to recreate these experiences that would otherwise be lost if obsolete hardware alone is relied upon. Having a software implementation of the system hardware prolongs its existence almost indefinitely.

There are also numerous practical reasons to emulate historical systems. For the present generation, emulators of past systems can be an insight into
historical contexts such as societal norms, language, and artistic styles.

Emulators can also provide educational value for students of computer architecture. Looking back on an old system in its original operating environment is impossible without proper emulation if the original hardware is not readily available. [11] Furthermore, the process of writing emulators of past systems and the examination of those systems have the potential to reveal past engineering mistakes and design flaws. Being aware of these mistakes instills valuable knowledge and diminishes the likelihood of repeating them. Solutions to many design problems may be found by studying these systems, saving time and effort in trying to re-engineer methods that have already been developed. [12] Finally, some software can never be run without the original system environment, including games that were made specifically for the Z-100. [13] Other good examples of this phenomenon are the operating systems designed for the Zenith Z-100. CP/M-85 and Z-DOS were the two operating systems designed specifically for the Z-100. [6] Although very similar to the standard CP/M and popular PC-DOS, the Z-100 operating systems will not operate as intended on other systems.

The task of developing emulators for past computer systems is an exercise of historical preservation and education. The operating system environments of these systems are quickly disappearing from public awareness. Additionally, the physical deterioration of any existing hardware continues to accelerate. Furthering this process is the eagerness to embrace new technology. With this disappearance also goes the software designed for those
environments. The development of a Z-100 emulator is made more important by the fact that there are no known complete implementations. It is conceivable that software specifically made for the Z-100 has never run in an emulated environment. In the absence of a complete Z-100 emulator, the physical system is needed to run its software. Theoretically, once all physical Z-100 systems are gone, some software may never be run again in the absence of a complete emulator.

**Previous Z-100 Emulation Attempts**

There are no known *complete* Zenith Z-100 emulations, but at least three incomplete implementations exist. These emulators are at different levels of completeness, but all of them have two common shortfalls. The first, and most important shortfall is that the disk drive portions are not functioning. As a result, no external programs can be read into memory and run, including its operating systems. Secondly, none of the emulators incorporate the Intel 8-bit 8085 processor; they appear to bypass the processor switching functionality and rely exclusively on the 16-bit 8088 processor. Despite these shortfalls, the emulators do initialize the system startup code and run the internal monitor BIOS program, though not all commands function properly.
The Z-100 monitor program is contained on its internal ROM chip. The three mentioned existing emulators successfully initiate the monitor program and reach its “hand” image command prompt (figure 2). Although, as mentioned, not all built-in commands function properly on any one of the existing emulators. None of the emulators have successfully implemented the BOOT command. Without this command functioning properly, operating systems cannot be loaded manually. Automatic loading of the operating systems will also not work because the automatic process eventually converges with the manual process code; both processes use the floppy disk controller. Other commands have limited functionality or do not function at all.

![Figure 2. Hand prompt of the Z-100 monitor program contained on its ROM](image)

The three incomplete Z-100 emulators are examined in detail below. Their levels of completeness are evaluated based on how much of the ROM monitor
program is functioning properly. Each of the original Z-100 monitor program’s twelve commands are tested for each of the three emulator implementations. [14]

**Rick Button’s C# Z-100 Emulator**

A Z-100 emulator was implemented by Rick Button in the C# programming language. The project has not been updated since 2016. [15] Of the three examined here, this implementation is the most complete. The README documentation included in its GitHub repository claims that the ROM program boots free of hardware faults, but also states that no disk hardware is implemented. This was found to be accurate. The emulator was compiled and run using Rider, the JetBrains product intended for .NET frameworks and C#. [16]

When the emulator runs, three windows appear: a simulation window that handles the Z-100 screen view (figure 3), a debug console window (figure 4), and a debug register window (figure 5). The simulation window shows the hand prompt of the ROM monitor program. In the upper left corner of the simulation window where the window title would normally appear, there is a three whole-digit, two decimal place number (indicated by a red box and arrow in figure 3). This number constantly changes while the emulator is running. The significance of this number is unclear. The debug console window outputs messages that help track the emulator’s progress, and the register debug window is a graphical user interface that allows stepping through the program’s instructions while
viewing the contents of the processor registers. A major problem exists with the system monitor window: the viewing area does not scroll when reaching the bottom as it should. Therefore, when text reaches the bottom of the screen it becomes invisible.

Figure 3. Z-100 screen simulated by Rick Button's C# emulator. The hand prompt is shown, and the number displayed in the window title is indicated with a red box and arrow.
Figure 4. Debug console window of Rick Button's C# Z-100 Emulator

Figure 5. Register debug window of Rick Button's C# Z-100 emulator
First, the BOOT command was tested by pressing the 'B' key at the monitor program prompt and pressing ‘ENTER.’ This command does not function properly in the Rick Button C# emulator. It seems that no matter what BOOT sequence or drive is selected, an MS-DOS system configuration screen appears (figure 6). At this configuration screen the number in the upper left corner of the system monitor screen window stops changing. The emulator then locks and will not advance any further. If the debugger is used to manually step through or force the program to advance, the processor halts and must be reset. The program also stops responding at seemingly random times.

Figure 6. MS-DOS System Configuration screen that appears when running the BOOT command in Rick Button's C# Z-100 emulator.
The HELP command does not work. Pressing the ‘H’ key should bring up the HELP menu text but entering ‘H’ in this implementation does not respond as it should; no help menu appears.

The INPUT command seems to be working fine, but it is uncertain of how to test this without having the ability to see the actual status of devices. Testing the OUTPUT command suffers the same limitation.

The TEST command presents a suite of diagnostic features included with the monitor program. This command is useful for evaluating the functionality of the emulator because it allows the ability to manually test certain components of the system. A five-item menu appears when the TEST command is used (figure 7). The functionality of each menu item for the C# implementation is explained below:

1. The DISK READ TEST does appear to work. This test reads the boot tracks and writes the data to memory. Every time the test is successful, an onscreen counter is incremented. [14] This test seems to be working in this implementation, however, there is no way to abort the test; the counter continues to increment. There is a message that says “TYPE <DELETE KEY> TO ABORT” but pressing the ‘BACKSPACE’, ‘DEL’ key, or any other key does not abort.

2. The KEYBOARD TEST works. However, again there is no way to exit because there is no ‘DELETE’ key available. To exit this and the disk test, the system must be reset.
3. The MEMORY TEST seems to initiate properly, but there is an error reported when the test is run: “FIRST BANK RAM ERROR, CHIP U109.” Again, the user cannot exit because the ‘DEL’ key is unavailable.

4. The POWER-UP TEST seems to work fine, however like the previous tests, the system must be reset to exit out of the test.

5. The EXIT option works properly.

All other monitor program commands appear to work as expected in this C# implementation. One possible exception is the EXECUTE command. It is not possible to test EXECUTE without writing an assembly program manually into memory. A program can be run with the EXECUTE command by specifying the memory address at which the program starts. [14]

Figure 7. The Z-100 monitor program TEST menu
MESS Z-100 Emulator

The Multi Emulator Super System (MESS) is a subset of software that is now part of the MAME project. A MESS project wiki page still exists, but it has not been updated since 2017. [17] Hyperlinks to MESS source code on the wiki page are directed to the MAME project GitHub repository. [18] Originally, MAME was a “Multiple Arcade Machine Emulator,” the name from which it derives its acronym. The MAME project has since expanded to include old personal computers along with its arcade systems and eventually incorporated the MESS project, which contained a multitude of historical system emulators. [19] Since the projects merged, MAME has been the primary developer and archiver of these emulators. A Z-100 emulator is among the list of systems included in MAME. However, multiple files are missing for the Z-100 emulator included in the newest MAME version. Therefore, an older MESS version was used for testing. The MESS Z-100 emulator was tested in the same manner as described above for the Rick Button C# implementation; the functionality of the monitor ROM program was explored.

With the proper libraries and the ROM file location set, the main MESS emulator software initializes, and a menu of systems is presented. The menu includes the Z-100. However, MESS warns that the Z-100 emulator is “NOT WORKING” (figure 8). Attempting to initialize the emulator brings up a dialog box that tells the user the emulator has problems. The message states that “THIS
SYSTEM DOESN’T WORK…There is nothing you can do to fix this problem except wait for the developers to improve the emulation” (figure 9). Continuing despite this message brings up the emulator’s system information. Confirming the fact that this emulator does not include the 8085 processor, MESS reports that the CPU is a “I8088 4.772727 MHz” processor (figure 10). Of course, this is only one of the two processors that should be included in a proper Z-100 emulation; the 8085 processor must also be included. When the system is run, the monitor program prompt appears as it should but with an error reported above the prompt: “INTERRUPT CIRCUIT ERROR, CHIP U208” (figure 11). Despite the error, commands can still be entered. There are no apparent debug tools available.

Figure 8. MESS main emulator menu showing the Z-100 highlighted and displaying the message that it is not working
There are known problems with this system
One or more ROMs/CHDs for this system have not been correctly dumped.
The game lacks sound.

THIS SYSTEM DOESN'T WORK. The emulation for this game is not yet complete.
There is nothing you can do to fix this problem except wait for the developers to
improve the emulation.

Type OK or move the joystick left then right to continue

Figure 9. MESS dialogue box stating that the Z-100 emulator
does not work

Figure 10. MESS system description for its Z-100 emulator

Figure 11. The MESS Z-100 monitor program prompt with
error message shown above the prompt
When the BOOT command is entered, nothing happens. No matter what is entered for the BOOT parameters, the system freezes. Also, pressing the ‘ENTER’ key has no effect at any time. Like the BOOT command, the DUMP and EXAMINE commands do not respond. The FILL command also seems to crash the system. It appears that when the ‘:’ (colon) key is pressed to complete the FILL syntax, the system crashes. For example, the system only allows the entry of “FILL 2000”. When ‘:’ is entered to get the proper FILL parameter syntax (<var>Fnnnn:mmmm-oooo,nn</var>), the system crashes. [14] This is the same for the DUMP and EXAMINE commands; entry of the ‘:’ character seems to crash the system. Alternatively, it is possible that the system is not crashing but simply not responding because of a flawed keyboard implementation.

The INPUT command seems to function and goes back to the hand prompt after returning a value. However, the validity of the returned INPUT value is uncertain. The OUTPUT command also does not crash the system. Again, there is no way to test the command because there seems to be no way to access external devices. For instance, trying to output to the printer port (E8) does not result in an observable response.

Like the Button C# emulator, the TEST command brings up the appropriate menu. The functionality of each menu item for the MESS implementation is explained below:

1. The DISK READ TEST fails and results in a “DEVICE ERROR,” which is reported in the upper left corner of the screen. The ‘DEL’ Key successfully aborts the test and exits to the TEST menu.
2. KEYBOARD TEST: Only lower-case letters and numbers are recognized. No punctuation or special characters register. This may explain the inability to type ‘:\’ when trying to enter memory address ranges for monitor program commands. The ‘ENTER’ key registers as a “NON-DISPLAYABLE CHARACTER.”

3. The MEMORY TEST freezes the system. The ‘DEL’ key does not abort the test as it should. A system reset is required to exit.

4. Initiating the POWER-UP TEST results in “ROM ERROR, CHIP U190” being displayed in the upper left corner of the screen. The test does not proceed as it should but pressing the ‘DEL’ key properly exits to the TEST menu.

5. The EXIT option works properly; the monitor program prompt appears again after exiting the TEST menu.

The rest of the monitor commands work as they should. However, like the C# emulator, screen scrolling is broken. When reaching the bottom of the screen the new lines overwrite the previous lines and the screen does not scroll up to display the new text output.

**Michael Black’s Z-100 Emulator**

Another partial Z-100 implementation was written by Michael Black. Like the Rick Button and MESS emulators, the Black emulator does not include the
8085 processor. It also does not implement the Z-100 display as it should appear, instead printing the prompt and monitor program text to the console using asterisks (*) to represent pixels. Debug information is printed directly to the same console as the monitor program display output. A hard-coded number of processor cycles are run with debug information printed at cycle intervals. After debug information is printed for all cycles, the monitor program prompt screen state is displayed using asterisk characters and the program ends. Above the prompt “DEFAULT DEV...” is displayed where error messages are typically shown (figure 12). The message display is cut off, but it is presumed that the word “DEVICE” is cut off at ‘C’. This effect is apparently caused by a limitation on the display width.

![Figure 12. Static hand prompt with "DEFAULT DEVICE" (?) and debug information displayed above the prompt](image-url)
Keyboard functionality at the monitor prompt could not be tested because the emulator does not allow live keystrokes at the prompt. Key presses must be scripted as hard coded `keyaction()` function calls. Key press sequences are coded at processor cycles 1,425,000 and 2,425,000. Multiple `keyaction()` calls can be initiated in a single cycle, but it is impractical to script a large number of keystrokes. As such, testing the monitor program commands could not be done as thoroughly as the previously discussed implementations, especially for the commands that require parameters that consist of memory address ranges. However, the monitor program was explored as much as possible by scripting a single letter entry for each command at cycle 1,425,000 and an ‘ENTER’ key press (newline character) at cycle 2,425,000. Command parameters were not scripted. The degree to which any of the commands functioned was inferred from the screen state output displayed after the cycle limit was reached.

The BOOT command appears to work. Key press ‘b’ occurring at cycle 1,425,000 and the ‘ENTER’ key (newline character - ‘\n’) occurring at cycle 2,425,000 results in the BOOT command being successfully run. Because no disks are present, the output is “Illegal Boot.” However, this shows that the BOOT command is most likely initializing as it should. The COLOR BAR command, which functioned properly for the other two emulators, also seems to work in the Black emulator. The limited screen functionality does not allow the entire color display to be seen but the command does display the “BLK” bar on screen as expected, albeit as multiple columns of asterisks. The HELP command works as
well; the list of monitor commands is displayed. System information is displayed as expected by the SYSTEM command.

It is not clear if the TEST command works because nothing is displayed on the screen after the cycle limit is reached. This may be the result of the screen width limit as the TEST menu is centered in the middle of the screen (see figure 7). The ROM version is not displayed after running the VERSION command, but control is returned to the prompt as normal. The DUMP, EXAMINE, FILL, INPUT, OUTPUT, and EXECUTE commands are recognized, but command parameters are required to check if they work correctly. Therefore, their full functionality could not be tested.

The Michael Black Z-100 Emulator is incomplete, but it includes many of the hardware component modules needed to implement a complete emulator. Also, the Black emulator is written in the C programming language, which is the language used to develop this project's emulator. Notwithstanding its incomplete screen display and keyboard limitations, the Black emulator seems to be functioning as expected at a fundamental level. Considering this, its overall program structure is used as a guide for development. Hardware component implementations included in the Black project are adapted for use in this project.

“Complete” Z-100 Emulation

As discussed, existing Z-100 emulators are incomplete. However, these emulators show that running the ROM monitor program in emulation is possible.
Rick Button’s emulator written in C# comes the closest to an error-free execution of the monitor program. According to his own admission, however, he has struggled to implement the disk hardware. [13] Without proper emulation of the disk control hardware, programs cannot be read into memory from disk. This limitation cuts functionality of the emulator to the dozen or so commands of the monitor ROM program. Although the monitor program does allow for the manual input and execution of machine code programs through the EXAMINE and EXECUTE commands, this falls far short of the full capabilities of the Z-100. [14]

The true essence of the system is contained in its advanced (for the time) graphics processing and its ability to run either the 8-bit CP/M-85 operating system or 16-bit Z-DOS by virtue of its dual Intel 8085/8088 processors. Without proper disk control hardware emulation, operating systems and programs that take advantage of the system’s capabilities cannot be demonstrated or preserved. As such, the desired outcome for this project is to emulate the full potential of the Z-100 system. “Full potential” in this case means loading and executing the CP/M-85 and Z-DOS operating systems and running programs that take advantage of the Z-100’s graphics capabilities. Although a significant amount of progress is made with the work done for this project, the goal of successful operating system initialization is not fully attained.
Development Approach and Strategy

The Z-100 emulator software for this project is largely written from scratch as understanding of the overall system is attained. However, some software implementations of peripheral devices and integrated chips (ICs) are borrowed from Michael Black’s emulator. Thus, much of the work done in this project concentrates on making these components work together to simulate the Z-100 system. In many cases, the code for these device implementations is modified to fit the Z-100’s unique design.

Successful implementation of the necessary hardware components requires correct and error-free reads and writes from and to their associated ports. The strategy employed to make sure these components are properly set up and incorporated involves tracing the diagnostic tests written into the monitor ROM program; the startup/reset diagnostics ensure the hardware is responding as expected. Tracing the monitor program’s diagnostics provides a valuable guide to properly implementing and incorporating the necessary hardware components.

Once all the necessary hardware components are integrated and the monitor program is running without errors, work begins to incorporate the FD-1797 floppy disk controller and its associated mechanisms. As noted, a working implementation of the floppy disk controller in conjunction with the monitor program BOOT command is a novel contribution by this study. Several components are involved, notably the 8259 interrupt controller and the 8253 timer. The timer is used in the controller’s initialization and the slave 8259
interrupt controller accepts interrupt signals from the disk controller. A considerable amount of time and effort is dedicated to the integration and development of the FD-1797 floppy disk controller emulation.

Assuming the successful implementation of the disk controller, the Z-DOS operating system is booted from a disk image using the BOOT command. Barring major problems while booting Z-DOS, other software can then be loaded and tested. Upon reaching the point of successful operating system initialization, various Z-100-specific software should run without error to consider the emulation "complete."
II. Emulator Initialization

Emulator Entry

As discussed, the basic structure for this project is based on the Michael Black Z-100 emulator attempt. Like the Black implementation, development is conducted in a Linux operating system environment using pertinent command line tools and is written entirely in the C programming language. System bus communication and device interaction generally facilitated by the system motherboard is simulated in a file named mainBoard.c. Crucial system constants and device implementations are also defined and initialized in mainBoard.c.

The ROM size is set as 0x4000 (16384 bytes) to accommodate the ROM binary file. Although various versions of the Z-100 monitor program ROM exist, this emulation is developed using the version 2.9 ROM contained in a binary file named “zrom_444_276_1.bin.” Version 2.9 is the latest version to be officially released by Zenith, and the noted filename contains the original Zenith part number for the ROM chip. [20] It is thought that the inefficiencies and design flaws present in previous ROM versions are remedied by the latest 2.9 version. A hexedit inspection of the version 2.9 ROM binary file (zrom_444_276_1.bin) reveals that the last byte is byte number 0x3FFF (figure 13). Thus, the 0x4000 ROM size constant is justified. RAM size is set at 0x30000 (196608 bytes). According to page 1.4 of the Z-100 Technical Manual (Hardware), 192K RAM is the maximum size memory available by way of three banks of 64K memory.
The main entry point of the emulator program begins with the presentation of a user menu (figure 14). There are two modes of operation: ‘Normal’ and ‘DEBUG.’ Normal mode runs the emulator as expected of the original system hardware; the system runs the processors forever in an infinite while loop. If the ‘DEBUG’ option is selected, the user is presented with two ways to set a breakpoint: by instruction number or by processor instruction pointer value (figure 14). In debug mode 1, the user can enter an instruction number to break at. When the indicated instruction is reached the emulator is paused and the user is allowed to manually step through subsequent instructions by pressing the ‘ENTER’ key. Additionally, information is displayed about the state of the system after the current instruction is executed. For example, if the user enters 1000 at the mode 1 debug prompt, the emulator operation will stop at instruction number 1000 and display a description of the processor state along with other information, including the state of the floppy disk drive controller module (figure 15). To advance, the user must press the ‘ENTER’ key to execute the next instruction. The emulator will break again at the next instruction and display system state.
information. Likewise in debug mode 2, emulator operation pauses when the value of the active processor's instruction pointer register reaches the user-supplied value. System state information is displayed as in debug mode 1 at the break point, and the user may step through subsequent instructions by pressing the ‘ENTER’ key.

Figure 14. Main entry menu of the JZ-100 emulator. Option 2 (DEBUG) is selected, and the DEBUG mode menu is presented.
Several data points relevant to the system state are reported in debug mode after each instruction step. Referring to figure 15, the first three lines list memory read operations carried out by the processor. Since the 8088 processor implementation uses a pipelined instruction flow, read and write operations reported here may not always apply to the current instruction. The fourth line tells the user the number of instructions that have been executed up to the current instruction step. It is important to note that the information displayed in the rest of the lines refers to the state of the system after the execution of the instruction number labeled “instructions done.” Considering this, the information reported in figure 15 is data related to the state of the system after instruction number 1000 has been executed.
Line five of figure 15 contains the current instruction pointer (IP), the last instruction operation code (opcode in x86 assembly), and the instruction name (inst). Since instruction 1000 has already been executed, IP points to the next instruction. Therefore, the IP value 0xC0 shown in figure 15 is pointing to instruction number 1001. The opcode and instruction name refer to instruction that last executed (instruction 1000). Instruction 1000 has an opcode of 0xAD. This instruction is “load word at address DS:(E)SI into AX,” which has the name “lodsw.” The name “lodsw” is listed as ‘inst,’ the last data point in line 5 of figure 15. Since registers AL and AH contain the bytes 0x26 and 0xA1, respectively, it is suggested that the lodsw instruction has loaded the 16-bit word 0xA126 into register AX. Lines 2 and 3 of figure 15 show that the processor read memory addresses 0xFC262 and 0xFC263 which contain the bytes now held by register AX. Noting also that register ES is set to 0xFC00 and register SI has advanced to 0x0264, it is shown that the ‘lodsw’ instruction functioned properly and the debug system state output is accurate and matches expected behavior.

**Main Emulator and GTK Screen Threads**

The GTK graphics library is used to display the Z-100 screen. This graphics library is part of the GNOME project published on Gitlab. [21]

Immediately after the startup menu item has been selected, the function generateScreen() defined in the emulator's video.c file is called. This function allocates memory for an unsigned integer array and places its reference into the "pixels" array variable defined in mainBoard.c. The array's size is determined by
the resolution of the Z-100 screen. As described on page 4.2 of the Z-100 Technical Manual (Hardware), the system’s display is generally considered to be 640 by 225 pixels under normal operation. Therefore, the pixel array, “pixels,” has a size of VWIDTH multiplied by VHEIGHT, where VWIDTH and VHEIGHT are defined in video.h as 640 and 225, respectively, matching the Z-100’s screen resolution. Finally, every element of the “pixels” array is initialized to zero and the generateScreen() function returns to mainBoard.c.

Next, an initialization function is called from mainBoard.c to set up the GTK window. As a standard operation when setting up a GTK window, the function screenInit(), defined in screen.c, calls the GTK function gtk_init(). This function initializes all necessary objects and variables underlying GTK functionalities. Window and drawing area objects are created using GTK functions gtk_window_new() and gtk_drawing_area_new(), respectively. These objects are assigned to the variable names “window” and “drawingArea,” and the drawing area is added to the window using the function gtk_container_add(). The drawing area is the container that holds all updates to the Z-100 screen; it displays the pixels defined in the “pixels” array. The window title is set to “Z-100 Screen” and the default window size is defined. Since development of this project is done on a computer with a full, high definition (HD) 1080 by 1920 resolution screen, a scale is applied to the default window size to maintain the box-like appearance of the original Z-100 display. The scaling factors are applied to the default window dimensions inside the appropriate GTK function: gtk_window_set_default_size(). X_SCALE and Y_SCALE constants are defined
in screen.c as 2 and 4, respectively. These scaling factors are appropriate for a 1080 HD display, though they may be adjusted to accommodate other host display resolutions.

The screenInit() function continues by connecting two callback functions to the drawing area object and one callback function to the window. When gtk_widget_queue_draw() is called via the display() function from the main Z-100 loop in mainBoard.c, it qualifies as a "draw" event. This “draw” event is connected to the drawing area using the g_signal_connect() function. Likewise, the ability to kill the GTK thread by closing the window using the “X” button is added with a connected signal. The callback function gtk_main_quit() is connected to the drawing area. This allows the ability to end the GTK thread by “destroying” the drawing area via closing the window. Closing the window thereby triggers a “destroy” event. A third callback signal is connected to the window itself. This third signal allows key presses on the host machine’s keyboard to be registered by the GTK window. To accomplish this, the on_keypress() function is connected to the GTK window object and is called when a “key-press-event” occurs. Figure 16 shows the code block within the screenInit() function that connects these callbacks. Direction to draw the window is done through a call to gtk_widget_show_all() with the “window” object variable passed to it as its parameter. ScreenInit() then returns to the main() function of mainBoard.c.
With the GTK window thread initialized for displaying the Z-100 screen, another thread is created and started for the main Z-100 computer emulator. The built-in C-language pthread module is used to initiate the thread. A function defined in mainBoard.c, `mainBoardThread()`, is called when the thread is started. The `mainBoardThread()` function subsequently calls `z100_main()` and the emulator starts. The final line of the main function starts the GTK window thread with a call to `screenLoop()`. Defined in screen.c, the `screenLoop()` function simply calls `gtk_main()`. With this call, the GTK window thread is started, and the window is open. A consequence of calling `gtk_main()` is that the GTK window thread never returns except in the case of manually closing (“destroying”) the window by clicking “X” in the window header or force quitting the emulator program by pressing ‘CTRL-C’ in the terminal. Although this behavior is typically undesirable, it is acceptable for the purposes of this project. Summarizing the previous discussion, Figure 17 shows the code block in the `main()` function in mainBoard.c responsible for starting the emulator and screen threads. With these threads running, the Z-100 emulator is ready to initialize its components inside.

```c
    g_signal_connect(G_OBJECT(drawingArea), "draw", G_CALLBACK(on_draw_event), NULL);
    g_signal_connect(G_OBJECT(drawingArea), "destroy", G_CALLBACK(gtk_main_quit), NULL);
    g_signal_connect(G_OBJECT(window), "key-press-event", G_CALLBACK(on_keypress), NULL);
```

Figure 16. Code block connecting callback functions to the GTK drawing area and window. These lines appear in the `screenInit()` function which is defined in screen.c.
the `z100_main()` function.

```c
// allocate memory for array and initialize each pixel element to 0
// (uses generateScreen() function from video.c to set up the pixel array)
pixels = generateScreen();
// call initialization function defined in screen.c to set up a gtk window
screenInit(argc, argv);
// start main emulator thread
pthread_create(&emulator_thread, NULL, mainBoardThread, NULL);
// start GTK window thread
screenLoop();
```

Figure 17. Code block found in the `main()` function of `mainBoard.c`. These lines set up and start the two threads that comprise the Z-100 emulator.

**Device Object Initialization**

The `z100_main()` function is called from the active thread function `mainBoardThread()`. Two sections make up the `z100_main()` function. Device objects and global variables are initialized in the first section and the second section contains the processor loop. Once the devices and global variables are initialized in the first section, the processor loop begins. This loop is an infinite while-loop. Each iteration causes the active processor to execute a single assembly instruction. After each instruction is handled by the active processor, device objects are clocked, and their internal timers are updated (if applicable) according to the time passed during the last instruction. Debug information is printed during each iteration if the emulator is in debug mode and the break point has been reached.
Keyboard and Video

To begin the first section of the z100_main() function, the keyboard device object is initialized using the newKeyboard() function defined in keyboard.c. The keyboard implementation’s basic structure is borrowed from the Michael Black Z-100 emulator. The newKeyboard() function begins by establishing a pointer and allocating memory for a keyboard object. The keyboard object’s “capsLock,” “dataReg” (data register), and “interruptsEnabled” fields are set to 0 (zero) and the keyboardReset() function is called with the keyboard object passed as its parameter. The keyboardReset() function sets the “autoRepeatOn,” “keyClickOn,” “keyBoardEnabled,” and “ASCIImode” fields to a value of 1. It also sets the head (“fifoHead”) and tail (“fifoTail”) fields to 0 (zero). These head and tail fields are index pointers for the emulated key buffer implemented as a simple first-in/first-out queue. The newKeyboard() function returns to z100_main() in mainBoard.c with a new keyboard object pointer. This pointer is assigned to the mainBoard.c global variable “keybrd.”

Next, a video object is initialized by calling the newVideo() function. This initialization function is defined in video.c. The video.c file and its associated header file, video.h, are borrowed from the Michael Black emulator. Like the keyboard initialization, memory is allocated for the video object and a pointer is established. Three fields that toggle the enabled mode of the Z-100 video hardware color planes are set to a value of 1. These fields are named “redenabled,” “blueenabled,” and “greenenabled.” According to page 4.6 of the Z-
100 Technical Manual (Hardware), the color display can produce eight different colors. This is accomplished by a combination of three superimposed pixels from each memory plane of video RAM (VRAM) assigned to each of the three main colors. The video emulation object created by the newVideo() function has the ability to turn these color planes on or off by setting any one of its “redenabled,” “bluenabled,” or “greenenabled” fields to 1 or 0. The “flashenabled” field is able to control the masking of VRAM. When this field is set to 1, the contents of VRAM are masked, and its memory content is not displayed to the screen. Instead, the entire screen appears white; all colors are turned on for all pixels.

There are eighteen registers organized in an array. This array is also included in the video object. The last field set in the new video object is the register select pointer for the array of registers; this is initialized to zero. The purpose of the register array is unclear. The newVideo() function returns to mainBoard.c and assigns the variable named “video” to the new video object.

I/O Ports

Before the emulator operation begins processing assembly instructions, several input/output (I/O) port variables must be initialized. After the video object is created, the initialize_z100_ports() function is called to assist in this initialization. Three I/O ports are represented by global variables in mainBoard.c. The initialize_z100_ports() function initializes these global variables.

The first port variable, set to 0x00, is the DIP switch port; its port designation is 0xFF as seen by the processor. On the actual hardware, this port
connects to a physical DIP switch situated on the Z-100 motherboard. Eight switches make up the physical DIP switch. Each of the eight switches determine the state of each bit of the incoming byte when the port is read. Since the switch is a physical component, the port cannot be written to programmatically. The DIP switch serves two functions: set the video refresh rate and determine the boot mode for the computer. Bits 0 through 2 determine the device from which the Z-100 will boot the operating system. There are two boot modes. The system may either automatically boot from the determined device on startup or the user may boot the OS manually from the monitor program prompt. Bit 3 determines the boot mode. The video refresh rate, which can be set to either 50 or 60 Hz, is determined by bit 7 of the DIP switch. Bits 4 through 6 are not used.

A global variable, “switch_s101_FF,” represents the DIP switch and is initialized to 0x00. The value of “switch_s101_FF” never changes during the emulator’s operation. Modifying its value is not allowed by emulator. If a change in the value of “switch_s101_FF” is desired, it must be hard-coded, which is in keeping with the fact that it is a physical switch on the hardware. A 0x00 value returned from this port sets the video refresh rate to 60 Hz and informs the monitor program that the user will boot the OS manually using the BIOS BOOT command from the hand prompt. The default boot device is set to the 5.25” floppy disk drive.

Next, the processor swap port variable, “processor_swap_port_FE,” is initialized to 0x00. This port determines which of the two core processors are active and signals appropriate adjustments to accommodate the active
processor. Page 2.9 of the Z-100 Technical Manual (Hardware) lists the definitions and functions of the port’s bits. Selecting the desired processor is accomplished by writing to bit 7. If a 1 is written to bit 7, the 8088 is selected. Otherwise, the 8085 is selected if a 0 (zero) is written to bit 7. There is an option available that generates an interrupt on interrupt request line 1 (IR1) of the master 8259 interrupt controller. This option allows an interrupt routine to be executed when the new processor is selected. Writing a 1 to bit 1 (D1) of the processor swap port (0xFE) enables this option. Writing a 0 to D1 will not cause an interrupt and the newly selected processor will continue operation from where the last processor left off. Additionally, a “mask” mode can be applied to the interrupt should it be called. In “mask” mode, a switch to the 8088 processor is forced and the interrupt is handled by the 8088. Setting bit 0 (D0) to 1 forces this switch. If D0 is set to 0 the selected processor will handle the interrupt routine regardless of which processor is selected.

Since the Z-100 starts reading 8085 code from the monitor program ROM at address 0x00 on startup or reset (page 1-17 – MTR100 – Z100 ROM Listing 2.5 Volume I), the 8085 should be selected first. Therefore, bit 7 of the “processor_swap_port_FE” variable byte is initialized to 0 (zero). Also, no interrupt is needed. As a result, “mask” mode is not necessary. Considering these conditions, initializing “processor_swap_port_FE” to 0x00 is sufficient for system startup. Bit 7 is 0 (zero), selecting the 8085 as the active processor, and bits 0 and 1 have a value of 0 (zero), indicating no interrupt and disabled “mask” mode. There is one consequence to implementing the processor swap port in
this manner: the active processor must be set to the 8085 with a hard-coded line before the processor loop begins. This manual setting is necessary because the monitor ROM does not write to the processor swap port before starting its execution. Without an explicit selection of the 8085 before the processor loop, the emulator does not know which processor to begin operation with at address 0x00.

The final port variable initialized in the initialize_z100_ports() function is the “io_diag_port_F6.” This variable represents the byte associated with port address 0xF6. Page 10.11 of the Z-100 Technical Manual (Hardware) describes port address 0xF6 as “reserved by ZDS [Zenith Data Systems].” Upon examining the ‘monitor loop’ assembly code of the Z-100 BIOS (page 1-21 – MTR100 – Z100 ROM Listing Volume I), it is discovered that port address 0xF6, designated “IO_DIAG,” is used for communication with a “diagnostic board.” The exact purpose and function of the diagnostic board is unclear. A likely explanation is that the board was an external device used by Zenith engineers to troubleshoot and diagnose problems during development of the Z-100. The monitor ROM tests for the presence of the board on startup/reset by reading from port 0xF6 and applying an AND operation to the incoming byte and the value 0x01. A jump to the diagnostic board code is executed if the result of the AND operation is 0x00. Since the diagnostic board is not implemented in this project, it is necessary to bypassing this jump. Therefore, the “io_diag_port_F6” variable is initialized to 0xFF. As a result, the AND operation yields 0x01 (0xFF & 0x01 = 0x01), and the jump to the diagnostic board code does not occur.
**ROM Initialization**

After the necessary I/O port variables are initialized, each byte of the file containing the monitor ROM program is read into an unsigned character array using the `loadrom()` function defined in mainBoard.c. The array is given the variable name “rom.” Using the typical C-language approach for programmatic file access, the C library function “fopen” is used to read the contents into a FILE stream object using the “read byte” (rb) mode. A for-loop consisting of “ROM_SIZE” iterations is used to load each file byte into the “rom” array.

ROM_SIZE is a constant defined as the value 0x4000 in keeping with the size of the version 2.9 ROM. The C-language library function `fgetc` is used to fetch each byte. When all iterations are complete, the “rom” array is a byte-by-byte copy of the ROM binary file. As previously noted, a file named “zrom_444_276_1.bin” is used as the ROM for this project. The numerical portion of the file name, 444-276-1, is derived from the Heath version 2.9 ROM chip part number. [20] Any memory reads from the monitor ROM program by the active processor is done from the “rom” array.

The next task in the initialization portion of the `z100_main()` function involves setting several global variables. First, the “romOption” variable is set to zero. ROM option 0 makes the ROM appear to be repeated throughout the entire memory space. Initializing this variable to 0 (zero) before the processor loop is necessary because the Z-100 begins at startup/reset in ROM option 0 mode.
More importantly, the emulator’s `z100_memory_read_()` function checks the ROM option setting every time the processor proceeds to read a byte from memory. The processor is directed to the appropriate address in the suitable array ("ram" or "rom") depending on the value assigned to the "romOption" variable. The “romOption” variable is a copy of the isolated bits 2 (D2) and 3 (D3) of the memory control latch (MCL) byte at port 0xFC. These two bits determine the ROM option configuration. If the bits equate to 0b00, the ROM configuration is set to option 0. If the bits equate to 0b01, the ROM configuration is set to option 1, and so on. Figure 18 is an excerpt from the Z-100 Technical Manual (Hardware) which summarizes the ROM options determined by bits 2 and 3 of the memory control latch. It is important to note that this port is not read from before the processor loop begins, so the ROM option must be initialized before the processor loop.
Parity Circuity Initialization

Three more global variables are initialized before continuing with device initialization. These variables are “killParity,” “zeroParity,” and “byteParity.” These variables play a crucial role in simulating the Z-100’s parity circuitry, and like the “romOption” variable, they are sampled every time the active processor reads a byte from memory through the \texttt{z100\_memory\_read()} function call. Also, like the “romOption” variable, “killParity” and “zeroParity” are copies of bits that appear in the memory control latch byte. The “zeroParity” variable represents memory control latch bit 4 (D4) and “killParity,” bit 5 (D5). As mentioned for the ROM configuration variable, the emulator does not read or write to the memory control latch port before the processor loop, therefore these variables must be initialized beforehand.
It is important to understand how the Z-100 deals with parity to ascertain how the global parity variables are used in the emulator. The Z-100 generates a single parity bit in dedicated memory chips for every byte written to RAM. The system also has circuitry that checks the parity of each byte read from memory against the stored parity bit (page 2.12, Z-100 Technical Manual - Hardware). Although the parity memory and associated circuitry is not fully implemented in this project, a logical solution is included in the `z100_memory_read()` function to satisfy the monitor ROM program’s parity diagnostic routine. Every time a byte is read by the processor (i.e., `z100_memory_read()` is called), the parity of that byte is calculated and stored in the “byteParity” variable. Immediately following this calculation, an interrupt may be generated on interrupt request line 0 (IR0) of the master 8259 interrupt controller. The interrupt is triggered under the following conditions: the “zeroParity” variable is set to one, the “killParity” bit is zero, and the calculated byte parity, stored in “byteParity,” has a value of 1. A value of 1 in the “zeroParity” variable equates to forcing the stored parity bit to 0 (zero). If “killParity” is 0, the parity checking circuitry is assumed to be active. With the “zeroParity” and “killParity” control options set as described, if the calculated parity evaluates to 1, an error results because the expected parity is 0. This error generates a parity interrupt. The monitor ROM parity diagnostic routine expects this behavior and is therefore satisfied by the described implementation.

Prior to the first byte being read by the active processor, the parity variables “killParity,” “zeroParity,” and “byteParity,” are all set to 0 (zero). As previously described, setting “killParity” to zero has the effect of enabling the
parity checking circuitry, and the parity bit is not forced to zero while the
“zeroParity” variable holds a 0 value. Initializing “byteParity” to 0 (zero) is a
formality since its value is updated with each processor-read byte.

Summarizing the effects of the global parity variable initializations: the
parity checking circuitry is enabled, the parity bit is not forced to zero, and the
calculated byte parity is initialized to 0 (zero) before the processor loop begins.
Any modifications to these variables are handled by the monitor ROM program
by writing to the memory control latch port. During the startup/reset procedure,
the monitor ROM program is primarily concerned with the parity circuitry for
diagnostic purposes.

Processor Object Initialization

The initialization portion of the z100_main() function continues with device
object setup. Processors are initialized beginning with the 8085. Michael Black’s
8085 is adapted for use in this project. Although some functions are modified, the
overall structure and function is largely preserved from Black’s original emulation.
Deviations from the original 8085 code are discussed in detail when necessary.

Initialization of the 8085 is accomplished by a call to the function
reset8085(). The reset8085() function is defined in 8085.c and is called with the
mainBoard.c “p8085” global object variable passed as its parameter. Resetting
the 8085 object involves setting the object’s program counter (instruction pointer)
to 0x00. The SIGN flag (“s”), PARITY flag (“p”), interrupt pin (INTR), and
HALTED (“halted”) signal are also all set to 0x00. Additionally, the ZERO flag
("z"), AUXILIARY CARRY ("ac"), and CARRY ("c") flags are set to 0x01. After these 8085 instance variables are initialized, the reset8085() function returns to mainBoard.c.

Following the 8085 reset in mainBoard.c is the 8088 processor object initialization. Like the 8085, the 8088 implementation used in this project is a modified version of Michael Black’s implementation. Setting up the 8088 differs from the 8085 in that it involves calls to three functions instead of one, and it uses assigned callback functions to facilitate memory and port access. Also, unlike the 8085 setup, an 8088 global pointer is declared (P8088*) instead of an entire global object variable as with the 8088 (P8085). As a result, the 8088 initialization uses the new8088() function, defined in 8088.c to allocate a “P8088” object and assign it to the P8088* pointer variable “p8088.” The new8088() function uses the standard C-library malloc function to allocate enough memory to accommodate a “P8088” object as defined in the 8088.h header file. The allocated memory is cast to a “P8088” pointer (P8088*) type and returned to the “p8088” pointer variable in mainBoard.c.

With a reference to the new 8088 processor object, “p8088,” the function assignCallbacks8088() is used to direct the processor to mainBoard.c functions that handle memory and port access. The assignCallbacks8088() function is defined in 8088.c and takes five parameters: a “P8088” pointer, two ‘store’ function pointers, and two ‘load’ function pointers. The function pointers are custom types defined in 8088.h. Signatures for these function types match the memory and port access function signatures in mainBoard.c.
Read functions \texttt{z100\_memory\_read()} and \texttt{z100\_port\_read()}, both defined in mainBoard.c, are recognized by the processor as the ‘load’ functions. Load functions accept an address parameter and return a value found at that address. In the case of \texttt{z100\_memory\_read()}, the value residing at the supplied memory address is returned. For reading ports, the value at the port address passed to \texttt{z100\_port\_read()} is returned. These functions’ signatures match the “load\_function” type signature that calls for an unsigned integer as the lone parameter (i.e. the target address), and an unsigned integer as the return type (i.e. the value at the target address). Main board write functions \texttt{z100\_memory\_write()} and \texttt{z100\_port\_write()} signatures match the “store\_function” type signature, where an unsigned integer – the target address, and an unsigned char – the value desired to be written to that address, are expected as the two parameters. There is no return value (void) for the “store\_function” type.

With the “load\_function” and “store\_function” pointer types defined, the \texttt{assignCallbacks8088()} function can accept pointers to the mainBoard.c memory and port access functions. The body of \texttt{assignCallbacks8088()} simply reassigns the mainBoard.c function references to four “P8088” internal instance variables. To the “P8088” object, references to \texttt{z100\_memory\_read()} and \texttt{z100\_memory\_write()} are seen as its instance variables “memory\_read\_x86” and “memory\_write\_x86,” respectively. Likewise, \texttt{z100\_port\_read()} and \texttt{z100\_port\_write()} are assigned to “port\_read\_x86” and “port\_write\_x86.” Functions with the same name as these instance variables are defined in 8088.c.
Each of these functions “call” the mainBoard.c functions through the instance variable references. For example, a function named “memory_write_x86” calls the internal instance “memory_write_x86” variable with the line listed below.

```c
p8088->memory_write_x86(address,data&0xff);
```

As shown in the single line of code from `memory_write_x86()`, the instance variable “memory_write_x86” is used to reference the mainBoard.c `z100_memory_write_()` function. Whenever a “P8088” instruction directive needs access to memory, the internal `memory_write_x86()` function is called. Similarly, internal functions are defined in 8088.c for reading memory and writing and reading from ports.

It is interesting to note that the 8085 processor implementation does not assign callback functions to its own internal instance variables as described for the 8088 implementation. Like 8088.c, there are internal read and write functions for memory and port access defined in 8085.c. However, the 8085.c function definitions call the mainBoard.c functions directly instead of making use of internal variables assigned to callback functions. This approach has the advantage of simplifying the processor implementation, but it requires hard coding the memory and port access functions in 8085.c for different system emulators, making the 8085 implementation less portable. In the case of the 8088 implementation, the `assignCallbacks8088()` function handles internal memory and port access function assignments with one line of code.
The third and final function call to initialize the “P8088” possessor device object is a call to reset8088(). This function takes the “P8088” pointer variable (“p8088”) in as a parameter. Various instance variables are initialized and a call to prefetch_flush() is invoked. Since the reset vector of the 8088 processor is at physical memory address 0xFFFF0, the code segment register (“CS”) is initialized to 0xF000, and the instruction pointer (“IP”) is set to 0xFFF0. All other registers, the flag values, and the “halt” signal instance variables are initialized to 0x00. A character array pointer (char*) instance variable that holds the name of the current instruction begin executed called “name_opcode” is initialized to an empty string (“”).

After these initialization, the prefetch_flush() function is called. This function sequentially sets each element of the “prefetch” array starting with the byte located at the address currently pointed to by “PC” and ending with the address at “PC” plus “PREFETCH_SIZE.” The PREFETCH_SIZE constant is defined in 8088.h and determines the size of the “prefetch” array; it is set to a value of 4. The “prefetch” array acts as a buffer for pipelined bytes that are fed into the processor’s data path. Since the starting address of the 8088 is 0xFFFF0, bytes read from addresses 0xFFFF0 through 0xFFFF3 are the first bytes set as the elements of the array at initialization. The prefetch_flush() function returns to reset8088(), which in turn returns to mainBoard.c.

**8259 Programmable Interrupt Controllers**

Continuing with device object initialization in mainBoard.c, the 8259
programmable interrupt controllers (PICs) are set up next. There are two interrupt controllers included in the Z-100: a master and a slave. One of the eight vectored interrupt lines on the master PIC is cascaded to the slave, which itself can handle up to eight vectored interrupts. The e8259.c file along with its associated header file, e8259.h, are taken from the IBM PC Emulator Project (PCE) written by Hampa Hug. [22] Hug states in the e8259.c code comments that there is no nesting (cascading) support. Since the IBM PC does not have a cascaded slave 8259 PIC, Hug’s emulator only includes a single 8259 PIC. [23] Therefore, the chip’s ability to cascade to a slave PIC is not implemented. As such, a modification is made to the e8259.c function e8259_inta() to facilitate the cascading ability for this project. Details about this modification are examined in later sections when the master/slave PIC dynamic is discussed.

Global pointer variables representing the master and slave PIC are declared in mainBoard.c and each is initialized using the e8259_new() function defined in e8259.c. One parameter is passed to e8259_new(); it is of the type “character array pointer” (char*). This parameter is added as a modification to the Hug’s original implementation and serves as a chip label instance variable. The label variable “label” is used to differentiate between the master and slave PICs. Other than the differing labels, the initial chip instances are identical.

The e8259_new() function makes use of malloc to allocate memory appropriate for the “e8259_t” type defined in e8259.h and assigns a pointer to the local “pic” variable. After the pointer variable assignment, the e8259_init() function, defined in e8259.c, is called. This function takes two parameters: an
“e8259_t” pointer variable (i.e. “pic”) and the “label” variable value passed to e8259_new() from mainBoard.c. The “label” variable is assigned, and the PIC register, control word, and other internal instance variables are appropriately initialized. The e8259_init() function then returns to e8259_new() which in turn returns to mainBoard.c with “e8259_t” type pointers assigned to the “e8259_master” and “e8259_slave” global PIC variables.

With the master and slave PIC global variables established and initialized, the e8259_reset() function is called for each of the PIC devices. The e8259_reset() function takes an “e8259_t” pointer variable as its only parameter. Several of the instance variables initialized in the e8259_init() function are again set to initial values. Another function, e8259_check_int(), defined in e8259.c, is called after variable initialization. This function takes the “e8259_t” pointer variable (“pic”) as its single parameter and checks if its interrupt request register has pending requests after the mask register value is applied. If there are pending interrupt requests, the PIC’s interrupt (“int”) line is set high (i.e. set to 1), otherwise it is set low (zero). Since the main processor loop has not started, a pending interrupt is not expected; the int line is reset to 0 (zero). The e8259_check_int() function then returns to e8259_reset(), which subsequent returns to mainBoard.c.

A final function call is required to complete the PIC initializations. The e8259_set_int_fct() function is needed to set the function to be called when an interrupt is generated by the PIC. Assigning a callback interrupt function is necessary because there must be an interface that causes the processor to
recognize the interrupt signal. The `e8259_set_int_fct()` function takes three parameters: an “e8259_t” type pointer, a generic “void” type pointer, and “void” type function pointer. For the master PIC, its global variable “e8259_master” is passed to the `e8259_set_int_fct()` as its first parameter, “NULL” as its second, and “interruptFunctionCall” as its third. The second parameter is assigned to the PIC object instance variable “intr_ext” and is said to be a “transparent parameter” according to Hug’s code comments. This variable is not used, so “NULL” is passed. The third parameter, “interruptFunctionCall,” is the name of a function, defined in mainBoard.c, that handles the master PIC’s interrupt signal to the processor. The `e8259_set_int_fct()` function assigns a pointer that references the `interruptFunctionCall()` function. This pointer is assigned to the master PIC’s “intr” instance variable.

When an interrupt signal needs to be sent to the processor by the master PIC, the `interruptFunctionCall()` function is called by the master PIC object. A call to the function in the e8259.c file is done by invoking the function through the PIC instance variable (“intr”). This call occurs within the `e8259_set_int()` function defined in e8259.c. True to `interruptFunctionCall()`’s signature, the “inter_ext” instance variable, initialized to “NULL” is passed as `intr()`’s first parameter. A local variable, “val,” which represents the state of the interrupt request pin connected to the processor, is passed as `intr()`’s second parameter.

Referring back to `interruptFunctionCall()`’s definition in mainBoard.c, a 0 (zero) value for “val” causes a return with no effect. If “val” holds 1 as a value, the `e8259_inta()` function, defined in e8259.c, is called. This function retrieves the
interrupt vector from the master PIC and stores it in a local variable “irq.” The vector is passed to the 8088.c \texttt{trap()} function as its second parameter along with the “P8088” object pointer global mainBoard.c variable, “p8088,” as its first parameter. Through the \texttt{trap()} function the processor is directed to the interrupt code by setting its instruction pointer (“IP”) and code segment (“CS”) register variables to appropriate values and updating pertinent flags.

Following the \texttt{e8259\_set\_int\_fct()} function call for the master PIC in mainBoard.c, the same call is made for the slave PIC. To set the slave interrupt function, \texttt{e8259\_set\_int\_fct()}’s first parameter is the mainBoard.c global “e8259\_t” pointer “e8259\_slave,” and like the master PIC interrupt function setup, the second “transparent parameter” is set to “NULL.” However, the slave PIC’s interrupt function does not use the same function definition as the master, instead it uses a mainBoard.c function named “cascadeInterruptFunctionCall.” This function name is passed as \texttt{e8259\_set\_int\_fct()}’s third parameter for the slave PIC. The body of the \texttt{cascadeInterruptFunctionCall()} does not call \texttt{e8259\_inta()} to retrieve an interrupt vector to pass to the processor’s \texttt{trap()} function. Instead, the e8259.c function \texttt{e8259\_set\_irq3()} is called for the master PIC. This call sets the IR3 pin “high” on the master PIC. Consistent with master/slave PIC arrangement in the Z-100, an interrupt request from the slave PIC causes the IRQ3 pin on the master PIC to go “high.” Figure 19 illustrates the Z-100 master/slave PIC arrangement. The figure, found on page 2.21 of the Z-100 Technical Manual (Hardware), shows the INT pin of the slave PIC connected to I3 (interrupt request line 3, IR3) of the master. Any interrupts handled by the
slave PIC are signaled to the processor by the master, however, the processor receives a vector released by the slave PIC.

As previously noted, Hampa Hug's 8259 programmable interrupt controller implementation for the IBM PC emulator, which is used in this project, does not support the master/slave PIC cascading scheme used in the Z-100. Hug's implementation is modified to accommodate interrupt cascading. Cascading interrupt functionality is crucial to building a complete Z-100 emulator because the floppy disk controller is set up as a slave device on the S-100 bus (page 6.2, Z-100 Technical Manual - Hardware). The interrupt request line (INTRQ) of the

Figure 19. Arrangement of the master/slave PIC circuitry in the Z-100 (page 2.21 - Z-100 Technical Manual - Hardware)
floppy disk controller is connected to any one of the Z-100’s eight vectored interrupt lines (VI0-VI7) (page 6.4, Z-100 Technical Manual – Hardware). These lines are attached to the interrupt request pins (IR0-IR7) of the slave PIC (see Figure 19). Therefore, the processor is expecting a vector from the slave PIC for interrupts originating from the floppy disk controller.

Modifying Hug’s 8259 Implementation

The following discussion illustrates how modifications to Hampa Hug’s 8259 implementation intend to emulate the PIC cascade functionality. The effect of a device interrupt on the IR0 pin of the slave PIC is used as a demonstration and code is explained as needed to clarify the program logic.

To enable an external device to trigger an interrupt on the Z-100’s slave PIC, the external device’s INT line is connected to one of the Z-100’s vectored interrupt (VI) lines on its S-100 bus. The eight VI lines in the Z-100 are connected to the interrupt request (IR) lines of the slave PIC. For example, if a device’s INT line is connected to VI line 0 (VI0), that device will cause the IR0 line on the slave PIC to go “high” when it requests an interrupt. It is not necessary to implement the VI/IR line connectivity in the Z-100 emulator. Instead, to cause a slave PIC IR pin to go “high” via a device's INT pin output, the device's emulation code must simply make a single function call. Hug’s PIC implementation provides eight separate functions to set each of the eight IR pins. For instance, the e8259.c function to set the IR0 pin is e8259_set_irq0(). The function takes two parameters: a “e8259_t” PIC device object pointer and an unsigned character
value representing the desired pin state, 0x01 for high and 0x00 for low. If the device calls “e8259_set_irq0(e8259_slave, 0x01),” the IR0 pin of the slave PIC is set “high.”

The definition of e8259_set_irq0() in e8259.c consists of a single line: a call to the e8259_set_irq() function. The third parameter accepted by this function is a relational comparison. The comparison is made between “val,” the desired pin state, and zero (val != 0). Therefore, any value other than zero will be converted to a 1 for this parameter, and an incoming 0 (zero) value will remain 0.

The first and second parameters for e8259_set_irq0() are the “e8259_t” PIC device object pointer and an integer value representing the targeted IR pin number.

Entering the body of the e8259_set_irq() function, an unsigned character mask variable, “msk,” is declared and initialized. The “msk” variable is the value 0x01 bit shifted left by the incoming target pin value (zero) combined with 0x07 with an AND operation (0x00 & 0x07 = 0x00). This operation causes no shift to the 0x01 value and the “msk” variable remains at 0x01. The overall operation here is meant to set the appropriate bit in the eight-bit character “msk” variable to match the physical orientation of the eight slave IR lines. For example, if IR4 is set high, “msk” is initialized to 0b00010000; bit 4 is set to one. In the example being examined here, IR0 is being set, thus the “msk” variable is 0b00000001, reflecting that IR0 is set “high.” Logic then checks if the internal interrupt input indicates that the desired interrupt is already “high” while an external process is trying to set it “high.” In this case, the function returns because no action is
needed. Since there are no pending interrupts, the internal variable “irq_inp,” representing the interrupt request (IRQ) input states, and the interrupt request register variable “irr,” are set to the “msk” value (0x01). Finally, the
\texttt{e8259\_check\_int()} function is called taking the “e8259\_t” PIC device object pointer as its single parameter. It is important to remember that this object pointer is currently pointing to the slave PIC and that all operations just described occur in the slave PIC.

The \texttt{e8259\_check\_int()} function then begins execution. First, the interrupt mask register variable “imr” is combined with the interrupt request register variable “irr” with an AND operation. This operation takes care of possible mask mode settings in the PIC. It is often the case that all interrupts are masked during BIOS initialization and diagnostic processes. However, for the purpose of this demonstration, it is assumed that no masks are applied. With no mask, “irr” remains unchanged at 0x01 and is assigned to a local variable with the same name (“irr”). The current state of the interrupt service register variable “isr” is checked for pending interrupts and the mask register is again applied. For this example, it is assumed that there no pending interrupts indicated by the interrupt service register and no masks are applied.

A local “msk” variable is initialized by a left bit shift of the value 0x01 using the “priority” instance variable. “Priority” is set to zero during the device reset procedure. As a result, the 0x01 value is not shifted and is assigned to “msk.” Inside a while-loop the interrupt service register is checked against “msk” using an AND operation. Since there are no pending interrupts, this condition does not
execute. Another condition checks the interrupt request register ("irr"), which has a value of 0x01, against “msk” (0x01) with an AND operation. This operation results in a value of 0x01, triggering the second condition. The second condition calls the `e8259_set_int()` function, defined in e8259.c. Two parameters are passed: the slave PIC object pointer and the value 0x01. The second value parameter, 0x01, is the state of the interrupt output; it is the state of the slave PIC’s INT pin in the actual physical system.

The last function called in the chain of internal PIC operations is `e8259_set_int()`. The `e8259_set_int()` function takes two parameters: an “e8259_t” PIC device object pointer (i.e. the slave PIC’s pointer) and an unsigned character value representing the state to be applied to the PIC’s INT pin (“val”). Since the PIC is receiving an interrupt request from a device, the state of the INT pin should be set to 1. Therefore, a value of 1 is passed as “val.” An instance variable named “intr_val” holds the status of the PIC’s INT pin. The “intr_val” variable is updated with the incoming state value held in “val” if it is different. In the situation being described here, “val” is equal to 1 and “intr_val” is updated since it is set to 0 (zero) at the PIC object’s initialization.

Next, `e8259_set_int()` checks that there is valid in interrupt function stored in the instance variable “intr.” The mainBoard.c function `cascadeInterruptFunctionCall()` is stored as the slave PIC’s interrupt function, “intr,” and it is a valid function (i.e. not “NULL”). Being a valid function, `cascadeInterruptFunctionCall()` is called from `e8259_set_int()` via the “intr” instance variable and takes two parameters: the PIC object’s “intr_ext” pointer
(not used), and the incoming INT pin state, “val”. If “val” is 0

cascadeInterruptFunctionCall() simply returns with no effect. This case is
equivalent to setting the INT pin to 0; the INT pin is reset, and no request is sent out. The “val” variable is 1 in this case, indicating an interrupt request must be sent out by the slave PIC.

According to interrupt circuitry design of the Z-100, the slave PIC’s INT pin is not connected directly to the processor, instead it connects to the interrupt request line three (IR3) on the master PIC. To emulate a high state in the master’s IR3 pin the cascadeInterruptFunctionCall() function calls

e8259_set_irq3() for the master “e8259_t” PIC device object pointer and passes a value of one as its second parameter.

From here, the internal PIC function chain previously described for the slave PIC repeats for the master. The master PIC’s internal function chain is identical to the slave’s until the master’s e8259_set_int() function call. Unlike the slave PIC, the master's interrupt function pointer, “intr,” does not point to the mainBoard.c function cascadeInterruptFunctionCall() but instead points to the mainBoard.c function interruptFunctionCall(). Like the cascadeInterruptFunctionCall() function, an incoming INT pin state of 0 (zero), which is labeled as the “number” parameter in both interrupt functions, causes the function to return without effect. This passive return does not happen in the situation where an interrupt is requested by an external device; the INT pin is set “high.”

When the master's e8259_set_int() function calls mainBoard.c's
interruptFunctionCall(). With the “number” parameter passed as 1 via its “intr” instance variable, the e8259.c function e8259_inta() is called. The e8259_inta() function is modified from Hug’s implementation. Hug’s e8259_inta() accepted only one “e8259_t” PIC device object pointer as a parameter. The function is modified to accept two such parameters, one for the master PIC object and one for the slave PIC object. Two “e8259_t” PIC device object pointers are required because a recursive call is included as another modification to Hug’s function. This modification is explained below with a trace through this project’s e8259_inta() version.

**Modified e8259_inta() Function**

Upon entering the e8259_inta() function from the call by interruptFunctionCall(), two unsigned character local variables are declared: “irrp” and “irrb.” Next, the e8259_set_int() function is used to clear the master PIC’s INT pin by passing “master_pic” as its first parameter and a 0 (zero) for its second. It is important to note that for the e8259_inta() call from interruptFunctionCall(), the master PIC’s “e8259_t” PIC device object pointer is passed as its first parameter, and the slave PIC’s pointer as its second parameter. This is a crucial point because when “master_pic” is used in this function call it refers to the master PIC object. Likewise, when “slave_pic” is used it refers to the slave PIC’s object pointer.

The highest priority interrupt is then determined for the master PIC by a call to e8259_get_priority(). Since only one interrupt is pending, it is the highest
priority. The priority number is stored into “irrp” and then converted to a register bit representation in “irrb.” Next, the bit in the interrupt request register, “irr,” corresponding to the IR line being serviced is cleared, leaving any other pending IR lines active. Following a check of the “auto end of interrupt” (AEOI) control word option and an increment in an active interrupt counter, e8259_check_int() is called to set up additional pending interrupts. Since only one interrupt is being serviced, this function returns without effect.

Following the e8259_check_int() call in e8259_inta(), the modified code written for this project is executed. A conditional statement checks if the current “master_pic” variable indeed refers to the master PIC’s object pointer by confirming that the PIC object’s “label” is “MASTER.” A check is also done to confirm that the interrupt request bit for the IR3 pin is “high” by testing that “irrp” equals 0x03. If this condition is met, it implies that the master PIC’s IR3 pin is responding to the slave PIC’s INT pin. Therefore, e8259_inta() is called recursively. However, with this recursive call to e8259_inta(), the slave PIC’s object pointer is passed as its “master_pic” parameter and “NULL” is passed as its “slave_pic” parameter.

Recursively calling e8259_inta() causes the previously described code trace through e8259_inta() to be repeated for the slave PIC’s object pointer passed as the “master_pic” parameter. When the conditional statement checks the device label of the “master_pic” parameter, it finds that its “label” variable is set to “SLAVE_.” In this case, the conditional statement body is skipped and the e8259_inta() returns “irrp” added to its “base” variable. The “base” variable
represents the interrupt vector memory location attached to the “master_pic” (here the “master_pic” is the slave PIC object) IR0 request line. Local variable “irrp” adds the appropriate increment to direct the processor to the correct interrupt vector. In the situation being described here, an external device causes an interrupt on the slave PIC’s IR0 line. Therefore, the slave PIC object’s “base” value plus 0x00 (“irrp” = 0) is returned by e8259_inta() in the mainBoard.c

interruptFunctionCall() function.

Continuing with the interruptFunctionCall() function body in mainBoard.c, the slave PIC’s “base” instance variable value plus 0 (zero), which refers to the interrupt vector table location directed to by the slave PIC’s IR0 line, is put into interruptFunctionCall()'s local variable “irq.” The “irq” value is passed to the 8088.c function trap() as its second parameter, “number.” The trap() function's first parameter is a “P8088” pointer for the 8088 processor device object defined in mainBoard.c.

Upon entering the trap() function, the current flag statuses, the code segment (“CS”), and instruction pointer (“IP”) variables values are stored in temporary local variables. If interrupts are enabled for the 8088 processor, the code continues. Otherwise, the trap function returns without effect. Assuming interrupts are enabled, the 8088 object's “IP” and “CS” instance variables are modified to direct the processor to the memory location of the interrupt code. Previous flag, “IP,” and “CS” values are pushed to the stack, interrupts are disabled by clearing the interrupt flag, and the trap flag is also cleared. A call to the 8088.c function prefetch_flush() clears the previously pipelined instructions
and prepares the pipeline for the interrupt code. The \texttt{prefetch\_flush()} function then returns to \texttt{trap()} which subsequently returns to \texttt{interruptFunctionCall()} in \texttt{mainBoard.c}. From this point, \texttt{interruptFunctionCall()} returns and the processor loop continues with the first instruction for the interrupt code and the device interrupt on the slave PIC IR0 line is serviced.

8253 Timer Initialization

Returning to device initialization in \texttt{mainBoard.c}, the next device to be initialized is the 8253 timer. Hampa Hug's 8253 timer implementation for the IBM PC emulation project (PCE) is used with some modification. Operations pertinent to the timer device object are targeted on a global pointer variable, “e8253,” of the type “e8253\_t.” Initialization begins with a call to the \texttt{e8253\_new()} defined in \texttt{e8253.c}. Upon entering \texttt{e8253\_new()}, a local pointer variable, “pit,” of the type “e8253\_t,” is declared and memory is allocated to satisfy the object requirements as defined in “e8253.h.” The local “pit” variable is then passed to the \texttt{e8253\_init()} function defined in \texttt{e8253.c}. This function initializes three counter objects contained within the “e8253\_t” object. Each of the counter objects are stored as an element in an array instance variable of the “e8253\_t” object. Another function, \texttt{e8253\_counter\_init()}, defined in \texttt{e8253.c}, is called with each of the three counter array elements passed to their own \texttt{e8253\_counter\_init()} call. The \texttt{e8253\_counter\_init()} function sets internal instance variables for each of the three “e8253\_counter\_t” type counters. These variables include representations of counter registers, output latches, mode setting, a “counting” indicator, a “gate”
indicator, an interrupt callback function pointer, and a general “value.” The counter registers, output latches, mode, “counting,” “gate,” and “val” are set to 0 (zero). “NULL” is assigned to the interrupt callback function specific to each counter.

After the counter variables are set, the \texttt{e8253\_counter\_init()} functions return to \texttt{e8253\_init()}. There are two more “pit” instance variables labeled “my addition” in the comments. These variables are set to zero; they appear as “pit\_timerZero=0” and “pit\_timerTwo=0.” It is unclear why the comment “my addition” is present. The \texttt{e8253\_init()} function then returns to \texttt{e8253\_new()}, which in turn returns the newly initialized “e8253\_t” object to the “e8253” mainBoard.c global variable.

Following the initialization procedure carried out by \texttt{e8253\_new()} on the “e8253” mainBoard.c global variable, the \texttt{e8253\_set\_gate()} function, defined in e8253.c, is called for each of the three internal counter objects of “e8253.” This function sets the gate pin representation for each counter object. The gate pin allows dynamic control of the counters. Depending on the mode, the count can be started, stopped, or reset by setting the gate pin to a certain value or registering its rising edge. Figure 20, taken from the Intel 8253/8253-5 data sheet, summarizes the effect of the gate pin state based on counter mode. [24] According to the Z-100 monitor ROM 8253 chip initialization procedure (page 1-30 - DIAG – Z100 Rom Listing Version 2.5 Volume I), counter 0 is set to mode 3 and counters 1 and 2 are set to mode 0. All three of the timer’s gates pins are set to 1 (“high”) by passing a value of 1 as the third parameter of \texttt{e8253\_set\_gate()}. 
As noted in figure 20, this enables counting for all three counter objects.

<table>
<thead>
<tr>
<th>Modes</th>
<th>Signal Status</th>
<th>Low Or Going Low</th>
<th>Rising</th>
<th>High</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Disables counting</td>
<td>——</td>
<td>Enables counting</td>
<td>——</td>
</tr>
<tr>
<td>1</td>
<td>——</td>
<td>1) Initializes counting 2) Resets output after next clock</td>
<td>——</td>
<td>——</td>
</tr>
<tr>
<td>2</td>
<td>1) Disables counting 2) Sets output immediately high</td>
<td>1) Reloads counter 2) Initiates counting</td>
<td>Enables counting</td>
<td>——</td>
</tr>
<tr>
<td>3</td>
<td>1) Disables counting 2) Sets output immediately high</td>
<td>Initiates counting</td>
<td>Enables counting</td>
<td>——</td>
</tr>
<tr>
<td>4</td>
<td>Disables counting</td>
<td>——</td>
<td>Enables counting</td>
<td>——</td>
</tr>
<tr>
<td>5</td>
<td>——</td>
<td>Initiates counting</td>
<td>——</td>
<td>——</td>
</tr>
</tbody>
</table>

Figure 20. Summary of gate pin operations for counter modes of the Intel 8253 timer chip (Page D.98 - Z100 Technical Manual - Appendices)

As previously noted, the `e8253_set_gate()` function accepts three parameters: a “e8253_t” programmable interrupt timer (“pit”) object pointer, the counter number (0-2), and desired gate pin state (0 or 1). Entering the function, a check in the form of an if-statement is done to ensure that the target counter number (i.e. the second parameter) is within range. Since the counter number parameter is an unsigned type, the value can never be interpreted as negative. Therefore, the check simply ensures that the value is less than or equal to 2. This number targets one of the three counters by indexing the counter object array within the 8253 timer object. Another function is called after the check, and two
parameters are passed to it. This function is `e8253_cnt_set_gate()`, and its two parameters are the internal counter object array element indexed by the checked counter number and the desired pin state.

Within the `e8253_cnt_set_gate()` function there is first a check that the counter’s gate pin state is not already set to the target state. If this is the case, the `e8253_cnt_set_gate()` function returns without effect. Otherwise, another check confirms that there is a valid function to set the gate value for the counter. This function is contained within a struct called “tab_mode” which is defined inside `e8253.c` itself. The “tab_mode” struct consists of three function pointers and directs incoming state values based on the mode assigned to the counter. Because pin state values have different effects based on counter mode, different functions are defined to carry out the appropriate effect. Since the `e8253_new()` function, through its call chain, sets each counter to mode 0, the function pointers within “tab_mode” point to functions appropriate for that mode. Mode 0’s “set gate” function is `cnt_set_gate_0()`. This function takes a counter object pointer and a target gate pin value and sets the “counting” instance variable to that value. In this case, the incoming value is 1 and counting is enabled. As shown in figure 20, a mode 0 counter’s “counting” operation is enabled when its gate pin is set to 1 (“high”). Following execution of `cnt_set_gate_0()`, the call chain returns to `mainBoard.c`. Each of the three 8253 counters have their gates set “high” with this process.

After the gate pins are set for each of the 8253 device object counters, “OUT” functions are assigned for each counter. These functions are called when
the counter they are assigned to terminates its count. This is necessary to facilitate the connection between the OUT pins of the 8253 counters and the rest of the Z-100 system. The Z-100 has a particular circuitry setup for its 8253 timer logic (figure 21). OUT pins from counter channels 0 and 2 are connected to an 8-bit latch accessible through port address 0xFB. A “high” OUT pin signal from channel 0 (counter 0) sets bit 0 of the latch to 1, and channel 2 (counter 2) sets bit 1. The rest of the latch’s bits (2-7) have no effect on the system. The latch outputs the state of the two active bits (1 and 2) to an “OR” gate. The output of the “OR” gate is connected to the master 8259 interrupt controller’s level 2 (IR2) pin. Thus, if either channel’s OUT pin goes high, a level 2 interrupt is registered by the master 8259. A read from the timer latch (port 0xFB) indicates which channel caused the interrupt. Channel one does not trigger an interrupt at the 8259 but can be used for software interrupts.

Figure 21. Setup of the Z-100's 8253 timer and surrounding circuitry (page 2.15 - Z100 Technical Manual - Hardware)
The functions assigned to the OUT pin signals for the 8253 counter channels simulate the particular behavior of the Z-100’s 8253 timer and its surrounding circuitry. These functions are defined in mainBoard.c and are assigned via the e8253.c function `e8253_set_out_fct()`. For counter channel 0, `e8253_set_out_fct()` accepts the “e8253_t” object mainBoard.c global pointer, “e8253,” an unsigned integer value of 0 (zero), NULL, and the function pointer name “timer_out_0” as its four parameters.

An examination of `timer_out_0()`’s function body reveals two function calls: `e8259_set_irq2(e8259_master, 1)` and `e8253_cascade_clock_ch1(e8253, 1)`. The first call sets the master 8259’s level 2 interrupt pin (IR2) as expected, and the second call clocks the 8253’s channel 1 counter. The `e8253_cascade_clock_ch1()` function is a custom addition to Hug’s 8253 implementation. Hug’s implementation for the IBM PC clocks all counters by one function, thereby simulating the effect of having all three counter channel CLK pins connected to the same clock source. This is not the case in the Z-100. As shown in figure 21, channel 1’s CLK pin is connected to the OUT pin of channel 0; channel 1 is clocked by channel zero’s OUT signal. As a result, `e8253_cascade_clock_ch1()` is added to e8253.c which only clocks counter 1.

The e8253.c function `e8253_clock()` which originally clocked all three channels is modified to effect only channels 0 and 2. Channel 2’s OUT function, `timer_out_2()`, also calls `e8259_set_irq2()` to cause the 8259’s IR2 pin to go high. No function is called for channel 1’s `timer_out_1()`.
As described, the e8253.c function `e8253_set_out_fct()` is used to assign the 8253 device object’s OUT functions. This function takes four parameters: an “e8253_t” object pointer, an unsigned integer value to indicate the target counter channel (0-2), a “void” pointer for the output function’s “transparent parameter” (not used - “NULL”), and a function pointer for the output function. Like the `e8253_set_gate()` function, upon entering `e8253_set_out_fct()`, a check is conducted to ensure that the incoming counter channel number is within range (0-2). If so, the appropriate counter element in the 8253’s counter object array has its “out_ext” instance variable assigned to “NULL” and its “out” instance variable assigned to the suitable OUT pin function. The function then returns to mainBoard.c. The 8253 device object’s initialization is complete after all three of its counter channels are assigned an OUT pin function.

**FD-1797 Floppy Disk Controller Initialization**

The final device object to be initialized is for the Western Digital FD-1797 floppy disk controller. As noted, a novel FD-1797 implementation is developed for this Z-100 project. Although other FD-1797 emulations are incorporated into previous Z-100 emulator attempts, development of a controller emulation from scratch is adopted as a strategy to understand the chip’s interaction with the system while also elucidating its internal mechanisms. Cursory examinations of these previous FD-1797 emulations are conducted as part of this research, but only Michael Black’s version is consulted for code structure. Black’s basic
function skeleton is used as a guide, but otherwise this project’s FD-1797 emulation is developed using only the manufacturer’s data sheet. This data sheet documentation is included in the Z-100 Technical Manual Appendices on pages D.184 through D.206. The data sheet explains the entire family of FD-179X controllers, but only the FD-1797 controller is emulated. In the interest of time, “WRITE” capabilities are not included. Focus is aimed at booting the OS, so only “READ” operations are needed. Taking these limitations into account, an attempt is made to stay true to the device’s expected behavior.

Developing the FD-1797 disk controller essentially from scratch provides valuable insight into possible design flaws of previous implementations. An overarching concern is that these attempts do not accurately replicate the correct functionality to function with the Z-100 monitor ROM’s boot sequence. A common problem with all previous Z-100 emulator attempts is that the boot sequence fails to load the operating system. This may be a failure of these FD-1797 implementations to completely load the necessary OS sectors from the disk. Not properly accounting for the controller’s internal timings and neglecting inherent device delays may explain the failure. Regardless of these potential errors, the strategy of developing the disk drive controller from scratch is thought to mitigate potential design errors. Understanding the intricacies of the controller’s internal mechanisms is an added advantage.

To set up and initialize the FD-1797 device object a “JWD1797” type global pointer variable, “jwd1797,” is declared in mainBoard.c. Initialization is accomplished in z100_main() using the newJWD1797() function defined in
The “JWD1797” struct is defined in jwd1797.h. Allocating sufficient memory to accommodate a “JWD1797” object using malloc, the newJWD1797() function creates a pointer and returns it to the “jwd1797” global variable in mainBoard.c.

With the new controller object initialized, the resetJWD1797() function, also defined in jwd1797.c, is called with the controller’s global pointer variable being passed as its single parameter. Many instance variables are initialized in the reset function. These variables include representations of internal FD-1797 registers such as the data, track, sector, command, and status registers. All register variables are of the unsigned character type and are set to 0x00. Various flag, counter, and pointer variables are also set to a value of 0 (zero). Exceptions occur with the “ready_pin” and “not_master_reset” variables. These variables are set to 0x01 because the drive starts as “ready” and not in a “master reset” condition. Another notable exception is the “rotational_byte_pointer.” This variable is an integer that represents the byte location of the drive’s read/write head within a track; it is set to 2500. Upon reset, the read/write head is situated at byte number 2500 within the current track. The intention is to start the head at a “random” rotational location as would happen in a real drive, instead of at the first byte of the track.

Other than initializing the FD-1797 object instance variables, the resetJWD1797() function handles the crucial task of converting a disk image file to a character array. The array is used by the disk controller object to access bytes that would appear on the disk. To accomplish this transition, the
assembleFormattedDiskArray() function, defined in jwd1797.c, is called in resetJWD1797() after all other reset tasks are done. The assembleFormattedDiskArray() function accepts two parameters: a “JWD1797” object pointer as its first parameter and, as its second, a disk image filename string (character array). The disk image file must be located inside the emulator project folder, and the filename string must include the file’s extension.

Entering the assembleFormattedDiskArray() function body, the first action is a call to the diskImageToCharArray() function as defined in jwd1797.c. The single task of this function is to read all bytes of the disk image file into a character array. Like the assembleFormattedDiskArray() function, it accepts a filename string and a “JWD1797” object pointer, but in reverse order; the filename is its first parameter, and the controller object pointer is its second. The standard C-language library (stdio.h) function fopen() is used to initialize a “FILE” byte stream object from the disk image in “read byte” (rb) mode. To get the total number of bytes contained in the file (i.e. the file size), the standard library function fseek() is used to advance the byte stream pointer position from zero to “SEEK_END.” Reading the current pointer position, which is located at the last byte of the file after execution of the fseek() function, is accomplished by using the standard library ftell() function. A single parameter, the disk image byte stream pointer (FILE*), is passed to the ftell() function. The last position of the file byte stream corresponds to the size of the disk image file. This value is stored in the “JWD1797” object’s “disk_img_file_size” instance variable. Since the position pointer is currently located at the last byte of the file byte stream, the standard
library rewind() function is used to restore the pointer’s position to 0 (zero).

Now that the file byte stream pointer is positioned at byte 0, and the total number of bytes contained in the file is determined, a character array is declared. Using the value of the “JWD1797” “disk_img_file_size” instance variable and the size of a character data type (sizeof(char)), the amount of memory needed for the array is calculated. The multiplication product of the size of a character (8 bits) and the size of the disk image file is passed to the malloc() function. The result is cast as a character array pointer type and stored in the local variable “diskFileArray.” Making use of the standard library function fread(), the “FILE” stream’s bytes are copied into the “diskFileArray.” The target array pointer, “diskFileArray,” is passed as fread()’s first parameter. Its second parameter, the value 1, indicates that each element has a size of one byte. As its third parameter, the “JWD1797” “disk_img_file_size” instance variable, indicates the number of bytes to be read. For its fourth parameter, the disk image “FILE” stream object is passed. The fread() function returns the number of bytes copied. As a check, this value is compared to “disk_img_file_size.” If these numbers do not match, every byte was not copied from the stream and an error message is printed to the console. Otherwise, a message confirming the successful transfer is printed. Finally, the “FILE” steam is closed and the “diskFileArray” pointer is returned to assembleFormattedDiskArray()'s local character array pointer variable “sectorPayloadDataBytes.”

It is important to note that the bytes contained in the disk image file are only the data payload of the actual physical disk media; they do not include the
format bytes necessary for reading the disk programmatically as the Z-100 system expects. As such, the “sectorPayloadDataBytes” array contains only the data payload bytes of the disk. For this reason, the 
assembleFormattedDiskArray()'s main task is to create a character array that accurately represents the bytes contained on a formatted floppy disk.

**Creating a Formatted Disk Array**

To determine an acceptable disk format, the FD-179X data sheet is consulted. According to the data sheet, the controller can handle a standard IBM disk format (page D.184, Z-100 Technical Manual – Appendices). Furthermore, the data sheet explains that “[d]isks may be formatted in IBM 3740 or System 34 formats with sector lengths of 128, 256, 512, or 1024 bytes” (page D.197, Z-100 Technical Manual – Appendices). An acceptable format scheme is confirmed by comparing the IBM system 34 format chart that appears in the FD-179X data sheet (figure 22) to the IBM modified frequency modulation (MFM) double density track format for a 5.25" DOS disk reported for the HP 9845 Emulator Project (figure 23). [25] Nearly every format byte shown in figures 22 and 23 are identical. One exception is the data portion. Figure 22 shows 256 bytes of data, while figure 23 shows 512. However, as noted, the FD-1797 can accommodate various common sector data payloads.
Another exception seems to be the “Gap 4b” length in figure 23. This gap, 652 bytes of 0x4E, does not appear to match the 598 bytes of 0x4E noted in figure 22. Upon further inspection, it is discovered that there is no discrepancy. A note on figure 23’s “Gap 3” states that the gap is not written for the last sector but instead only “Gap 4b” is written. Figure 22 lists this in a different manner, having
this gap written at the end of each sector including the last. Summing the figure
22 terminating sector gap of 54 bytes with the track end gap of 598 bytes yields
652 bytes. This realization allows the track formats shown each figure to be
congruent. Therefore, this track formatting scheme is adopted for the
assembleFormattedDiskArray() function’s operation.

With all data payload bytes from the disk image file transferred to the local
variable “sectorPayloadDataBytes” through the diskImageToCharArray() call and
the proper track format determined, the assembleFormattedDiskArray() function
continues. Next, the disk’s pertinent properties are stored in “JWD1797” instance
variables. Disk properties crucial for constructing a formatted array are the
number of cylinders (i.e. tracks per side), the number of sides (either single or
double sided), the number of sectors per track, and the sector length in bytes.
This information is contained in the loader parameter table located at the
beginning of the disk’s data payload. Bytes 0x03 through 0x18 are reserved for
these parameters.

Figure 24 shows a hexedit utility view of the first 64 (0x40) bytes of “z-dos-
1.img,” the Z-DOS disk image file used in this project. The disk parameter bytes
are highlighted inside the red border. Of the 22 bytes highlighted, only six are
useful for the task of building the formatted disk array: bytes 0x4, 0x5, 0xC, 0xD,
0xF, and 0x15. Sector size is stored in bytes 0x4-0x5. A ‘little-endian’ read of
these bytes shown in figure 24 indicates a sector size of 0x0200 (512) bytes.
This is consistent with a standard IBM format and one of the sector sizes
acceptable to the FD-1797. The total number of sectors on the disk is contained
in bytes 0xC-0xD. Again, a ‘little-endian’ read reveals the total number of sectors to be 0x0280 (640). To obtain the number of sectors per track, byte 0xF is examined and its value is read as 0x08. Therefore, there are 8 sectors contained on each track. Dividing the total number of sectors on the disk, 640, by the number of sectors per track, 8, yields the total number of tracks on the disk: 80 (640/8 = 80). Finally, bit 0 of byte 0x15 is examined to determine whether the disk is single or double sided. Byte 0x15 has a value of 0x01, therefore its 0-bit is 1. This indicates that the disk is double sided. A double sided disk divides its total number of tracks evenly between its sides, resulting in 40 tracks per side (80/2 = 40). Interpretations of disk parameter table bytes are found on page 10.18 of the Z100 Technical Manual (Hardware).

![Figure 24. The first 64 (0x40) bytes of "z-dos-1.img" as shown by the hexedit utility - Loader disk parameter bytes are contained within the red border.](image)

Gathering the necessary disk parameter values into the “JWD1797” object’s instance variables is a matter of accessing the appropriate byte in the assembled “sectorPayloadDataBytes” array. To get the number of disk sides, the element at index 0x15 undergoes an AND operation with the value 0x01. This gets the 0-bit that indicates the number of sides. A 0 value for the bit indicates a single-sided disk, and value of 1 indicates double-sided. For this reason, 1 is
added to the result of the AND operation, which yields the number of sides. The result is stored into the “JWD1797” instance variable “num_heads.”

Setting the “sectors_per_track” instance variable is simply a matter of assigning the value of the element at index 0xF to the variable.

The ‘sector length’ disk parameter is stored in a 16-bit word (two bytes). These bytes are at indices 0x4 and 0x5. As discussed, data words are arranged in ‘little-endian’ order, therefore it is necessary to shift the element at index 0x5 left by 8 bits and then apply an OR operation with the element at index 0x4. This gives the sector length in bytes. It is stored in the “sector_length” instance variable.

Since the number of disk cylinders is not explicitly expressed in the loader disk parameter table, attaining a value to store in the “cylinders” instance variable requires a two-step process. First, bytes containing the total number of sectors, which are indexed at 0xC and 0xD, are stored in the local variable “total_sectors.” The same process used for extracting the sector length must be used to properly arrange the total sector bytes because of the ‘little-endian’ order. Dividing the number of total sectors by the number of sectors per track gives the total number of tracks on the disk. To get the number of tracks per side, the result is divided again by the number of sides (heads). The final result is stored in the “cylinders” instance variable. To clarify the calculations for the number of cylinders, the relevant code lines are shown below. The “w” variable holds the JWD1797 object pointer.
int total_sectors =
    sectorPayloadDataBytes[0xC] | (sectorPayloadDataBytes[0xD]<<8);
...

w->cylinders = total_sectors/w->sectors_per_track/w->num_heads

With the loader disk parameters table values extracted and stored in the “JWD1797” instance variables, the next calculation finds the total number of bytes in a formatted track. In addition to the payload data bytes for each sector on the track, the formatted track includes bytes from the gap sections, sync sections, address marks, and cyclic redundancy check (CRC). The number of bytes contained in each section are stored in jwd1797.c as constants declared in preprocessor (#define) directives. For example, gap 4a, as it appears in figure 23, contains 80 bytes of 0x4E. This section is located at the beginning of all tracks. The number of bytes in this section is stored in the “GAP4A_LENGTH” constant and its byte, 0x4E, is stored in “GAP4A_BYTE.” Following gap 4a there is a sync section. This section consists of 12 bytes of 0x00. Constants “SYNC_LENGTH” and “SYNC_BYTE” are declared for the section’s length and byte content, respectively. Similarly, the remaining sections have constants declared for their lengths and byte contents. The tables below summarize the constants declared for each formatted track section. Section names are taken from figure 23.
Gap 4a:

<table>
<thead>
<tr>
<th>GAP4A_LENGTH</th>
<th>80</th>
</tr>
</thead>
<tbody>
<tr>
<td>GAP4A_BYTE</td>
<td>0x4E</td>
</tr>
</tbody>
</table>

Sync:

<table>
<thead>
<tr>
<th>SYNC_LENGTH</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYNC_BYTE</td>
<td>0x00</td>
</tr>
</tbody>
</table>

Index Address Mark:

<table>
<thead>
<tr>
<th>INDEX_AM_PREFIX_LENGTH</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>INDEX_AM_PREFIX_BYTE</td>
<td>0xC2</td>
</tr>
<tr>
<td>INDEX_AM_LENGTH</td>
<td>1</td>
</tr>
<tr>
<td>INDEX_AM_BYTE</td>
<td>0xFC</td>
</tr>
</tbody>
</table>

Gap 1:

<table>
<thead>
<tr>
<th>GAP1_LENGTH</th>
<th>50</th>
</tr>
</thead>
<tbody>
<tr>
<td>GAP1_BYTE</td>
<td>0x4E</td>
</tr>
</tbody>
</table>

ID Address Mark:

<table>
<thead>
<tr>
<th>ID_AM_PREFIX_LENGTH</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID_AM_PREFIX_BYTE</td>
<td>0xA1</td>
</tr>
<tr>
<td>ID_AM_LENGTH</td>
<td>1</td>
</tr>
<tr>
<td>ID_AM_BYTE</td>
<td>0xFE</td>
</tr>
</tbody>
</table>
ID Field Data (lengths only):

- CYLINDER_LENGTH: 1
- HEAD_LENGTH: 1
- SECTOR_LENGTH: 1
- SECTOR_SIZE_LENGTH: 1

CRC (note: CRC byte is hard coded as 0x01):

- CRC_LENGTH: 2
- CRC_BYTE: 0x01

Gap 2:

- GAP2_LENGTH: 22
- GAP2_BYTE: 0x4E

Data Address Mark:

- DATA_AM_PREFIX_LENGTH: 3
- DATA_AM_PREFIX_BYTE: 0xA1
- DATA_AM_LENGTH: 1
- DATA_AM_BYTE: 0xFB

Gap 3:

- GAP3_LENGTH: 54
- GAP3_BYTE: 0x4E

Gap 4b:

- GAP4B_LENGTH: 598
- GAP4B_BYTE: 0x4E
Calculating the total number of formatted track bytes is accomplished by summing the appropriate track section length constants. As shown in figure 23, several sections are repeated for each sector on any given track. Figure 23 also notes that the sector portion is repeated either 8 or 9 times; this reflects the number of sectors per track for DOS format 5.25" 320k and 360k floppy disks, respectively. [26] To account for this in the formatted track byte sum, the section lengths of sync (12), ID address mark prefix (3), ID address mark (1), cylinder (1), head (1), sector (1), sector size (1), CRC (2), gap 2 (22), second sync (12), data address mark prefix (3), data address mark (1), sector payload data (size depends on the sector payload), a second CRC (2), and gap 3 (54) sections are added together. Summing these section lengths gives the number of formatted bytes in one sector. Multiplying the number of formatted bytes for a single sector by the number of sectors contained on track yields the number of formatted bytes for all sectors within a single track. Adding the byte lengths for the gap 4a (80), sync (12), index address mark prefix (3), index address mark (1), gap 1 (50), and gap 4b (598) sections completes the total formatted track byte count. The final sum is assigned to the “JWD1797” “actual_num_track_bytes” instance variable. For clarity, the code segment appearing in the assembleFormattedDiskArray() function implementing the described calculation and assignment is shown below. As with the previous code listing, “w” represents the “JWD1797” object pointer. Dynamic values for the number of sectors per track and sector length are incorporated into the sum calculation by using “JWD1797” instance variables (shown in bold text).
w->actual_num_track_bytes = GAP4A_LENGTH + SYNC_LENGTH
    + INDEX_AM_PREFIX_LENGTH + INDEX_AM_LENGTH + GAP1_LENGTH
    + (w->sectors_per_track) * (SYNC_LENGTH + ID_AM_PREFIX_LENGTH
    + ID_AM_LENGTH + CYLINDER_LENGTH + HEAD_LENGTH + SECTOR_LENGTH
    + SECTOR_SIZE_LENGTH + CRC_LENGTH + GAP2_LENGTH + SYNC_LENGTH
    + DATA_AM_PREFIX_LENGTH + DATA_AM_LENGTH + w->sector_length
    + CRC_LENGTH + GAP3_LENGTH)) + GAP4B_LENGTH;

Immediately following the formatted track byte summation, 
assembleFormattedDiskArray() performs another crucial calculation. To simulate the floppy disk’s rotation, a “rotational byte read limit” is determined and applied. This limit holds the “release” of each successive track byte until a determined amount of time has passed. In effect, the controller will not advance to the next byte until the time limit has expired. The “JWD1797” “rotational_byte_pointer” instance variable holds the current rotational location, and its value ranges from 0 through number of formatted track bytes minus 1 (“actual_num_track_bytes” - 1).

As an example, a 360k DOS-formatted 5.25” floppy disk has formatted tracks that are 6,396 bytes in length. Thus, the “rotational_byte_pointer” instance variable for a “JWD1797” object handling a 360k disk has values in the range 0 through 6,395.

Several operating conditions and assumptions are considered when calculating the rotational byte read time limit. First, a disk rotation of 300
revolutions per minute (rpm) is assumed. Three hundred rpm is the standard rotation speed of 5.25" floppy disks of the early 1980's. [26]–[28] As such, the FD-1797 disk controller implementation for this project simulates a 300 rpm rotation speed. This implies that one rotation takes $1/300$ of a minute or approximately 200 milliseconds. Since all bytes in a formatted track must be “seen” by the read/write head during this time, the total rotation time must be divided evenly among the track bytes.

Using the 360k disk’s 6,396-byte formatted track length as an example, each byte must be read in about 0.03127 milliseconds (31.27 microseconds). Assigning a raw quotient to the rotational byte limit by dividing the track rotation time (200 milliseconds) by the number of total formatted track bytes is not sufficient, however. This insufficiency is a byproduct of the method used to clock the “JWD1797” controller object. The method has the disk controller object accept time slice values from the main processor loop. Time slices are calculated based on the number of processor cycles required to execute the current iteration’s assembly instruction. Since both processors in the Z-100 are clocked at 5 MHz, a single processor cycle takes 0.2 microseconds (200 nanoseconds). The total cycle overhead of the current instruction is multiplied by 0.2 to produce the instruction’s time slice. For instance, if the current instruction requires six processor cycles, a 1.2 microsecond time slice is passed to the “JWD1797” object during that processor loop iteration.

Limited by the 5 MHz clock, the smallest possible time slice that can be passed during each processor loop iteration is 0.2 microseconds (1 clock cycle).
For this reason, a ‘modulo 200’ operation is applied to the raw rotational byte limit quotient and subtracted from the quotient itself. The value ‘200’ is used because the byte time limit is set to the closest 0.2 microseconds (200 nanoseconds), and the limit is expressed in units of nanoseconds. The raw value is calculated using 200,000,000 nanoseconds in place of the 200 millisecond single rotation time as the numerator. Nanoseconds are used instead of microseconds to avoid float and double precision data types. To that end, the raw quotient is also cast to an ‘unsigned long’ type before the modulo operation is applied. Although the time slice is passed to the “JWD1797” object as a double data type, it is immediately multiplied by 1000 and cast to an integer, thereby converting it to units of nanoseconds. Avoiding floating point data types sidesteps any compounding rounding errors inherent in floating point arithmetic. [29]

Again using the 360k disk formatted track for a sample calculation, the raw rotational byte read time limit is calculated as 200,000,000 nanoseconds divided by 6,396 bytes, and the result is cast to an ‘unsigned long’ type. The raw quotient is 31,269 nanoseconds. ‘Modulo 200’ is applied and the final value of 31,200 is stored in the “JWD1797” “rotational_byte_read_limit” instance variable. To check the result against the expected track rotation time, 31,200 nanoseconds per byte is multiplied by 6,396 bytes. The result of this calculation is 199,555,200 nanoseconds. This result is slightly shorter than the expected track rotation of 200,000,000 nanoseconds, but it is less than a 0.3% variation (199,555,200/200,000,000 = 0.997776). A rotational speed within 3% of the expected value is acceptable. [26]
For clarity, the code segment that calculates the rotational byte read limit time is listed below. “JWD1797” instance variables are highlighted in bold text, and the “w” variable represents the JWD1797 object pointer.

```
unsigned long raw_rotational_byte_read_limit =
    (unsigned long)(200000000/w->actual_num_track_bytes);
...
w->rotational_byte_read_limit = raw_rotational_byte_read_limit -
    (raw_rotational_byte_read_limit%200);
```

With the amount of formatted track bytes determined and stored into the “JWD1797” “actual_num_track_bytes” instance variable, the total number of formatted disk bytes is calculated. This calculation gives the number of bytes contained on the entire formatted disk while taking all disk parameters into account. The calculation’s result is used to declare a size for a new local array variable, “fDiskArray.” “JWD1979” unsigned char pointer instance variable “formattedDiskArray” is made to reference the new local array.

Before starting construction of the array that represents all bytes of the formatted disk, two more local variables are declared; they are of the ‘unsigned long’ type and act as array index pointers. One of the variables, “formattedDiskIndexPointer,” keeps track of the “formattedDiskArray” index
currently being written to. The other index pointer variable, “sectorPayloadArrayIndexPointer,” is used to indicate the next index to be read from the sector payload data array. The array index pointers are initialized to zero. Payload data is stored in the “sectorPayloadDataBytes” array variable. As described, this array is constructed as a byte-by-byte copy of the disk image file.

Using the newly initialized array assigned to the JWD1979 “formattedDiskArray” instance variable and the two array index pointers, “formattedDiskIndexPointer” and “sectorPayloadArrayIndexPointer,” the assembleFormattedDiskArray() function begins its final task of construction the formatted disk array. Format bytes defined in jwd1797.c’s preprocessor directive constants and elements of the “sectorPayloadDataBytes” array are used to fill the formatted disk array. The main loop structure consists of a triple nested loop. The overall loop structure is summarized in the truncated code listing shown below. Just the loop condition signatures are shown. As with previous code listings, “JWD1797” instance variables are highlighted in bold text, and the “w” variable represents the “JWD1797” object pointer.
// for each cylinder
for(int cyl = 0; cyl < w->cylinders; cyl++) {
    // flip sides for the next track (for each head)
    for(int h = 0; h < w->num_heads; h++) {
        ...
    }
    // for each sector
    for(int s = 1; s < w->sectors_per_track + 1; s++) {
        ...
    }
}

The main loop structure begins with the cylinder loop. The first cylinder is described as “cylinder 0.” The last cylinder number is equal to the total number of cylinders minus 1. For a 40 cylinder disk, the cylinder numbers run from 0 to 39. Therefore, the cylinder loop initializes its control variable, “cyl,” to 0 and runs the loop while “cyl” is less than the total number of cylinders (“cyl” < w->cylinders). The cylinder loop body begins with the head loop.

The read/write head number active for side 1 is recognized as head “0” and “1” for side 2. The second nested loop in the main disk formatting loop structure handles the active head; its loop variable, “h,” is initialized to 0. The
stop condition for this loop is when “h” equals the value of the “JWD1797” “num_heads” instance variable. For the 5.25” Z-DOS floppy disk, the loader disk parameter table indicates a double sided disk. In the case of these disks, the “num_heads” variable is set to 2. The ‘side’ loop runs for 2 iterations where “h” equals 0 for the first iteration and 1 for the second iteration. Each consecutive track is located on alternating sides of the disk. As such, the head loop variable switches before the next formatted track begins. For example, formatted track 0 (zero) is located on side 1 and is read by head 0 (zero). Formatted track 1 is on side 2, which is read by head 1.

Writing track format bytes begins inside the ‘head’ (side) loop. Referring to the section labels shown in the HP 9845 Project’s track format description in figure 23, gap 4a, sync, index address mark, and gap1 sections must be written at the beginning of every formatted track. A separate loop is used to write each section of format bytes. Inside each of these loops, bytes are written to the formatted disk array (“formattedDiskArray”) at the element indexed by the “formattedDiskIndexPointer.” The pointer is advanced to the next index with each iteration. Once the gap 4a, sync, index address mark, and gap1 sections are complete, the sector loop begins.

Sectors are numbered from 1 to the total number of sectors on a single formatted track. In an 8-sector track, the first sector is labeled 1 and the last is designated as sector 8. For this reason, the control variable for the sector loop, “s,” is initialized as 1. The loop terminates when “s” equals the number of sectors per track plus one (s < “sectors_per_track” + 1). This loop conditional statement
applies the appropriate number to each sector. Every formatted sector area begins with a sync section followed by an ID address mark prefix. These sections are written by their respective for-loops. Cylinder, head (side), and sector ID bytes are assigned based on the current “cyl,” “h,” and “s” loop control variable values. A switch statement then determines the appropriate sector length byte based on the value of the “JWD1797” “sector_length” instance variable. IBM-formatted disks use a single byte to indicate sector size (figure 23). The value 0x00 should appear in the sector length byte location of the ID field to indicate a 128-byte sector, 0x01 indicates 256 bytes, 0x02 for 512 bytes, and 0x03 for 1024. The switch statement code block is listed below to illustrate the decision structure. In the last part of the block, the sector length byte is written to the proper index of the formatted disk array.

Next, two cyclic redundancy check (CRC) bytes are written into the ID field. This FD-1797 implementation does not calculate the CRC bytes. Instead, the value 0x01 is written to the CRC locations as place holders. No functional CRC checks or calculations are performed in this project’s FD-1797 implementation. When the FD-1797 should check the CRC bytes, this implementation confirms that the bytes are read as 0x01.
// write sector length byte

int s_length_byte = 0x00;

switch (w->sector_length) {
    case 128:
        s_length_byte = 0x00;
        break;
    case 256:
        s_length_byte = 0x01;
        break;
    case 512:
        s_length_byte = 0x02;
        break;
    case 1024:
        s_length_byte = 0x03;
        break;
    default:
        printf("%s\n", "ERROR: Non-standard sector length!");

}

w->formattedDiskArray[formattedDiskIndexPointer] = s_length_byte;
Continuing with the sector loop, gap 2 is written via its dedicated loop. The sector loop then moves on to the data field. A loop writes a sync section and then the correct bytes for the data address mark. Immediately following the data address mark, a loop transfers bytes from the array holding the sector payload data ("sectorPayloadDataBytes") into the "data" portion of the sector area. The loop runs for "sector_length" iterations and increments both the "formattedDiskIndexPointer" and "sectorPayloadArrayIndexPointer" values with each iteration. An example is provided in the following description to illustrate a feasible payload data transfer.

For a transfer executed for the first sector's data payload in a 512-byte sector length disk, bytes 0 through 511 from the "sectorPayloadDataBytes" array are copied to bytes 206 through 717 of the formatted disk array ("formattedDiskArray"). The code segment for the data payload loop is listed below for clarity. As before, the "w" variable represents the "JWD1797" object pointer. Immediately after the "data" section, two more 0x01 CRC bytes are written followed by a loop that handles the 54 bytes of gap 3 to end the formatted sector section. The entire sector loop is repeated for the number of sectors that appear on each track.
// write the data payload
for(int ct = 0; ct < w->sector_length; ct++) {
    w->formattedDiskArray[formattedDiskIndexPointer] =
        sectorPayloadDataBytes[sectorPayloadArrayIndexPointer];
    formattedDiskIndexPointer++;
    sectorPayloadArrayIndexPointer++;
}

Upon emerging out of the sector loop the final sub-section of the track loop writes the last part of the formatted track; this is the 652 bytes of gap 4b. The entire head (track)/sector loop structure is repeated for the number of disk cylinders. When all track formatting loop structures have terminated, the assembleFormattedDiskArray() function returns to the resetJWD1797(), which in turn, returns to mainBoard.c’s z100_main() function.

Counts, Timers, and Process Flags

Now that all device objects are set up and initialized, the final task before the processor loop involves the initialization of several mainBoard.c global variables. These variables include counters, timer values, and flags that are crucial for operations contained within the processor loop.
The first variable to be initialized is “instructions_done;” its value is set to 0. The value of “instructions_done” corresponds to the number of instructions that have been executed. Its value is displayed with debug output information. The “instructions_done” variable is also used to stop emulator execution when using the “Break at Instruction number” debug option. For this option, the processor loop pauses at the instruction number entered by the user. From the pause point, the user may step through each instruction. The “instructions_done” variable helps facilitate this debug option.

Simulating and tallying instruction execution time is done using the next three initialized global variables. The “last_instruction_cycles” variable is simply a storage place for the previous instruction's processor cycle count. Its value is multiplied by 0.2 to get the time slice in microseconds that occurred during the last instruction's execution. The resulting time slice is stored in the “last_instruction_time_us” variable, and its value is added to “total_time_elapsed” each processor loop iteration. There is no functional purpose assigned to the “total_time_elapsed” variable, but it is printed to the terminal during debug steps to give the user an approximate gauge of how much virtual time has passed. All three of these global variables are initialized to 0 (zero).

The next two global variables to be initialized are “e8253_timer_cycle_count” and “e8253_timer_overage.” These variables are involved in handling clock pulses of the 8253 programmable interrupt timer object. Given that the 8253 is tied to a 250 kHz clock, the 8253 device object is clocked every 4 microseconds (page 10.8, Z-100 Technical Manual - Hardware).
This is achieved by clocking the 8253 when 20 processor cycles have passed. Twenty cycles are needed because the processors are clocked at 5 MHz, or every 0.2 microseconds (20 cycles * 0.2 µs/cycle = 4 µs). The “e8253_timer_cycle_count” variable keeps track of passing processor cycles for the 8253. When the value of “e8253_timer_cycle_count” reaches or surpasses 20, the 8253 is clocked. Any overage of cycles is stored in the “e8253_timer_overage” so extra cycles are not lost in the case of “e8253_timer_cycle_count” exceeding 20. The “e8253_timer_cycle_count” and “e8253_timer_overage” variables are initialized to 0 (zero).

Input/output (I/O) operations in the Z-100 often necessitate processor “wait states.” A processor wait state occurs when an external device must transfer data to or from the computer at a slower rate that the computer’s processing speed. [30] The processor pauses execution and enters a wait state until the device signals that data is ready to be transferred. Since certain data transfer conditions require processor wait states, there must be a way to signal the processor loop to halt instruction execution. This signaling is handled by the “processor_wait_state” flag, which is the next mainBoard.c global variable to be initialized. When the “processor_wait_state” flag equals 1, the active processor logic responds accordingly and generates a wait state. Otherwise, when the flag is 0, the processor object continues executing instructions. The “processor_wait_state” flag is initialized to 0 before the processor loop begins.

There are two debug mode options available for the emulator. One option allows the user to define a break point by instruction number. The second option
allows break points to be set to the processor’s instruction pointer (IP) register value. These two debug modes involve fundamentally different programming logic because instruction numbers increase sequentially, while IP register values inevitably jump to distant values unpredictably. Instead of having logic that breaks execution at or above a user-entered value as in the instruction number break mode, a flag is needed to indicate that a certain IP register value has been reached. This flag is handled by “debug_mode_2_active,” and it is the next global variable to be initialized. When the target IP register value has occurred, the “debug_mode_2_active” flag is set to 1 and processor execution breaks at every proceeding instruction. The flag is initialized to 0 before the processor loop.

The final global variable to be initialized is of an enumerated type defined in mainBoard.c. The enumerated type is the “active_processor” type and it can hold two possible values: ‘pr8085’ or ‘pr8088.’ A variable of this enumerated type is used as a flag to signal which of the two processors included in the Z-100 is currently active. Since the active processor may be changed at any time, this value must be updated dynamically. A global variable of the “active_processor” type is declared and has the same name as the type. The value of the “active_processor” variable determines which processor handles the next assembly instruction. If the value is ‘pr8085,’ the processor loop logically selects the 8085 processor object to handle the instruction. Alternatively, a value of ‘pr8088’ selects the 8088 processor. In keeping with the Z-100’s method of processor selection, the only way to change the value of the “active_processor” variable is to write the appropriate byte to the processor swap port (0xFE). A byte
written to port 0xFE with a 1 in its 7 bit will select the 8088 processor and thereby change the “active_processor” variable's value to ‘pr8088.’ If the written byte has a 0 in the bit 7 position, the variable's value switches to ‘pr8085.’ In this case, the processor loop logic will select the 8085 processor object to execute the next instruction. When the Z-100 powers on (or is master reset), the first block of monitor ROM instructions to be executed is handled by the 8085. Therefore, “active_processor” is initialized to ‘pr8085,’ causing the processor loop to logically select the 8085 processor object first.
III. Processor Loop

With device objects and necessary global variables set up and initialized, the processor loop begins. It is an infinite while-loop with no programmatic exit. To close the emulator program once in the loop, the user must either close the Z-100 screen GTK window or terminate the process manually in the terminal (e.g. pressing CTRL-C). The processor loop has five main tasks: process assembly instructions, handle debug conditions and output, clock device objects, simulate the VSYNC interrupt, and update the Z-100 screen window. Except for an if-statement decision structure to determine the active processor, the body of the loop consists only of function calls to helper functions that carry out the required tasks.

Debug Mode 2 Flag Update

Upon entering the processor loop, the first function to be called is `handleDebug2Mode()`. This function’s task is to set the “debug_mode_2_active” global variable flag during the appropriate loop iteration. However, the flag will only be set if debug mode 2, “Break at Instruction Pointer (IP) value,” is selected. If the emulator is operating in debug mode 2, the `handleDebug2Mode()` checks which processor is active and whether the active process has reached the instruction pointer register value entered by the user. However, the 8085 processor is said to have a “program counter” (PC) instead of an instruction
pointer (IP) register as in the 8088 (page B.3, Z-100 Technical Manual – Appendices). These registers serve the same purpose; they both point to the memory location of the next instruction when combined with the code segment (CS) register’s value. Therefore, the handleDebug2Mode() function checks the 8085 object’s “PC” instance variable if the 8085 processor is active. When the user-entered break point value equals the 8085’s “PC” value, “debug_mode_2_active” is set to 1. In the case of the 8088 processor being active, the 8088 object’s “IP” instance variable is checked for equality with the break point.

**Active Processor Selection**

After checking the debug mode 2 condition via the handleDebug2Mode() function, the active processor decision structure selects a processor to handle the next instruction. The decision structure checks the value of the enumerated global variable “active_processor.” If the value of “active_processor” is ‘pr8085,’ the function handle8085InstructionCycle() is called. Otherwise, the handle8088InstructionCycle() function is called when “active_processor” has the value ‘pr8088.’

**8085 Wait State Logic**

While the 8085 processor is active, the handle8085InstructionCycle() mainBoard.c function is called in the processor loop. Entering the function, the 8085 object’s “data_request_” instance variable is set according to the
“JWD1797” object’s “drq” instance variable. The “data_request_” variable is used as a signal to the processor that there is a new byte available in the data register of the “JWD1797” object. This is meant to simulate the connection between the data request (DRQ) pin of the FD-1797 disk controller and the READY pin of the 8085 (page 6.29, Z-100 Technical Manual - Hardware). The READY pin is used with wait state mechanisms. [31] Connection between the 8085 READY pin (pin 35) and the WAIT LOGIC module is shown in the Z-100 main board schematic segment in figure 25. [32] As previously discussed, wait states are required when procedures read data from external devices that are slower than the main processor. In the Z-100, this is the case when data is read from the floppy disk controller. While the processor is in a wait state, no further instructions are executed until the external data is ready. Therefore, a signal is needed to resume the instruction execution flow. The 8085 object’s “data_request_” variable receives this signal from the “JWD1797” object’s “drq” variable, which represents the FD-1797’s DRQ pin state.

Continuing with examination of handle8085InstructionCycle(), the function’s main task of processing the current assembly instruction is handled by the doInstruction8085() function. Defined in 8085.c, the doInstruction8085() function emulates instruction processing by simulating the processor’s data path. It takes a “P8085” object pointer as its single parameter. Based on the incoming instruction, values of pertinent 8085 object instance variables are manipulated and I/O operations are handled. These procedures are left largely unchanged from Michael Black’s original 8085 implementation; however, some modifications
are included for this project. These modifications are necessary to account for the processor’s handling of wait state conditions triggered by the FD-1797 disk controller.

The body of the `doInstruction8085()` function begins with a check of the "P8085" object's "halted" instance variable. If its value is 1, the P8085’s “cycle"
instance variable is set to 1 to signify the passing of a single 5 MHz clock cycle and the `doinstruction8085()` function returns without any further processing. In this case, no instruction is executed and the state of the “P8085” object remains unchanged.

Following the “halted” condition check, the `check_wait_state()` function, defined in 8085.c, is called with the “P8085” object pointer passed as its only parameter. This function is a custom addition to Black’s implementation. Inside the `check_wait_state()` function body, two consecutive bytes are read into local variables “pre_check_byte_1” and “pre_check_byte_2.” These bytes are read from memory using the `z100_memory_read_()` function defined in mainBoard.c. The two bytes are gathered from locations pointed to by the “P8085” object’s program counter value ([PC]) and PC plus 1 ([PC+1]). The resulting byte values are passed as the two parameters of the mainBoard.c function `pr8085_FD1797WaitStateCondition()`. This function is called as one of the arguments of the if-statement condition in the body of `check_wait_state()`. The other argument of the condition checks if the “JWD1797” object’s “drq” variable currently holds a value of 0. This simulates a check of the FD-1797’s DRQ pin. The condition arguments are logically connected by an AND operator. Thus, a value of 1 returned from the `pr8085_FD1797WaitStateCondition()` function and a value of 0 stored in the “data_request_” variable will execute the if-statement body which sets the “P8085” “wait_state” instance variable to 1. If one of the conditions returns 0, the “else” body sets “wait_state” to 0.

Hardware configurations in the Z-100 are made to facilitate the wait state
condition for reading data from the FD-1797 floppy disk controller. Since this configuration is organized on the main board, the functions simulating the hardware is declared in mainBoard.c. For the 8085, it is the

\texttt{pr8085\_FD1797WaitStateCondition()} function. Two parameters are passed to \texttt{pr8085\_FD1797WaitStateCondition()}, the opcode of the current instruction and a port number. If the instruction is “IN” (opcode 0xDB for the 8085) and the port number corresponds to the data port of the FD-1797 controller, the wait state condition is met, and the processor is expecting a data byte read from the disk. The \texttt{pr8085\_FD1797WaitStateCondition()} function returns a value of 1, in this case. If the condition is not met, 0 (zero) is returned.

When the 8085 is expecting to read a data byte from the FD-1797’s data port (i.e. assembly instruction 0xDB with immediate value 0xB3) and its “ready_” variable is 0 (zero), the 8085 instance variable “wait_state” is set to 1. This logic state represents a low status on its READY pin as the result of a low on the FD_1797’s data request pin.

Immediately following the return from \texttt{check\_wait\_state()} in 805.c’s \texttt{doInstruction805()} function, the value of “wait_state” is checked. A value of 1 sets the p805 “cycles” instance variable to 1 and the \texttt{doInstruction805()} function returns without any other effect. If “wait_state” has a value of 0, the wait state condition has not been met and the current instruction is processed as normal.

After the \texttt{doInstruction805()} function returns to \texttt{handle805InstructionCycle()}, p805’s “wait_state” instance variable is checked
again. If no wait state is required, the mainBoard.c “instructions_done” variable is 
incremented and the “last_instruction_cycles” variable has its value assigned as 
“p8085.cycles.” If a wait state is imminent, the “last_instruction_cycles” is 
assigned a value of 1 to indicate a single clock cycle has passed. No instruction 
is executed in this case, and the “instructions_done” variable is not incremented. 
This ensures that the system clock advances for other devices. These devices 
must continue their clock cycles even though no instruction is executed. The last 
section of code in the handle8085InstructionCycle() function prints the “p8085” 
instance variables related to the processor’s state. This information is used for 
debugging purposes. Prints statements are only triggered if a debug mode is 
active and the break point has been reached.

**8088 Wait State Logic**

If the 8088 processor is active (i.e. “active_processor” = pr8088), the 
handle8088InstructionCycle() function is called instead of 
handle8085InstructionCycle(). The process described for 
handle8085InstructionCycle() is largely mirrored for the 8088’s procedure. 
Entering handle8088InstructionCycle(), the “p8088” “ready_x86_” instance 
variable is made to reflect the “JWD1797” “drq” instance variable. As with 
“p8085,” this represents the connection between the FD-1797’s data request 
(DRQ) pin and the processor’s READY pin. The 8088.c function 
dolInstruction8088() is then called to process the current instruction. Like 
handle8085InstructionCycle(), the wait state condition is checked with a custom
function. For “p8088,” this is the check_wait_state_x86() function. Inside this function, the current byte being pointed to by the program counter is checked to confirm an “IN” instruction. Unlike the 8085, the 8088 has two versions of the “IN” instruction. One version uses an immediate value for the port number, and the other version uses the value in register DX. Thus, the pre-checked bytes for version 2 of the “IN” instruction must include the 16 bits of the DX register to gather the port number.

The 8088's check_wait_state_x86() calls on its specific wait state condition function defined in mainBoard.c: pr8088_FD1797WaitStateCondition(). “IN” instruction opcodes for the 8088 (0xE4, 0xE5, 0xEC, and 0xED) are confirmed as the current instruction and the port number for the FD-1797’s data register is verified to be the instruction's immediate value. A value of 1 is returned by pr8088_FD1797WaitStateCondition() if these conditions are met, a value of 0 (zero) is returned otherwise. Returning to the second conditional statement included in check_wait_state_x86()’s function body, the wait state signal variable for the “P8088” object, “wait_state_x86,” is set to 1, otherwise it is set to 0 (zero). Returning to doInstruction8088(), the 8088’s wait state signal is checked through the “wait_state_x86” instance variable. No instruction is executed and the doInstruction8088() function returns with no other action if a wait state is signaled. A 0 (zero) value stored in “wait_state_x86” allows the current instruction to be executed as normal and the rest of doInstruction8088()’s function body is performed.

The remaining code logic for handle8088InstructionCycle() is practically
identical to the 8085’s cycle handler function. Again, if no wait state is signaled
(“wait_state_x86” = 0), the “instructions_done” variable is incremented and the
“last_instruction_cycles” variable has its value assigned as “p8088” “cycles”
instance variable. As before, if a wait state is signaled, the instruction count is not
incremented, and the number of the last instruction's cycles is set to 1 to indicate
a single clock cycle (200 nanoseconds) has passed. As before, no instruction is
executed in this case. Debug information describing the “p8088” object’s state is
printed to the terminal if a debug mode is active and the indicated break point is
reached.

**updateElapsedVirtualTime()**

Depending on which processor is active for the processor loop’s current
iteration, either *handle8085InstructionCycle()* or *handle8088InstructionCycle()*
returns to the next function called in the loop. The *updateElapsedVirtualTime()*
function is called next. This function calculates a time slice based on the number
of cycles the last instruction consumed and stores it in the mainBoard.c
“last_instruction_time_us” variable. It is calculated by multiplying the number of
instruction cycles by the amount of time a single 5 MHz cycle takes. Considering
a 5 MHz cycle takes 0.2 microseconds (200 nanoseconds), the number of
instruction cycles is multiplied by 0.2 and the result is stored into
“last_instruction_time_us.” The “total_time_elapsed” value is updated by adding
the instruction time slice to its value. With these assignments complete, the
updateElapsedVirtualTime() function returns to the processor loop.

**VSYNC Interrupt**

A process contained in the Z-100 monitor ROM’s startup initialization sequence requires that a vertical display synchronization (VSYNC) interrupt be detected within a specified timeout period (page 1-29 – INTR_TST – Z100 Rom Listing 2.5 Volume I). This interrupt is triggered via the master 8259 programmable interrupt controller’s (PIC) IR6 line. If this interrupt is not detected within the timeout period, an error sequence is initiated. To satisfy the monitor program’s expectation, the interrupt is generated every 83,333 processor clock cycles. This cycle count is equivalent to 0.0166666 seconds (16,666.6 microseconds). Since the display refresh rate is set to 60 Hz and the VSYNC pulse is coupled with refresh rate, the 83,333 cycle count limit is a very close approximation (1/60). The simulateVSYNCInterrupt() is called in each processor loop iteration to handle the interrupt generation when appropriate.

Entering the simulateVSYNCInterrupt() function, the mainBoard.c “vsync_timer_cycle_count” global variable is updated by adding the value stored in “last_instruction_cycles.” Next, “vsync_timer_cycle_count” is checked to verify if it has reached or exceeded the “VSYNC_TIMER_CYCLE_LIMIT” constant. The constant is defined as a preprocessor directive (#define) and is set to the determined VSYNC cycle limit, 83,333 cycles. Should the limit be reached or exceeded, the e8259_set_irq6() function is called with the “e8259_master” interrupt controller (PIC) object pointer passed as its first parameter and a value
of 1 as its second. Overflow due to the number of previous instruction cycles bringing the value of “vsync_timer_cycle_count” over the cycle limit is tracked. The overflow amount is stored into the “vsync_timer_overage” global variable.

The `e8259_set_irq6()` function call triggers a level 6 interrupt on the master PIC. Count overage is used as a reset value for the VSYNC cycle count; the value of “vsync_timer_cycle_count” is assigned as “vsync_timer_overage.” This assignment preserves any overage so that all processor cycles are accounted for, thereby avoiding a compounding count error. In the case that `simulateVSYNCInterrupt()` is called when the IR6 pin state on the master PIC is “high,” `e8259_set_irq6()` is called with a value of 0 (zero) passed as its second parameter. This resets the master PIC’s IR6 pin as would happen when the VSYNC signal pulse ends. A global variable, “debug_int6,” is used by `mainBoard.c` to keep track the IR6 pin’s state.

**8253 Timer Clock Cycle**

Returning to the processor loop, the `handle8253TimerClockCycle()` function is called. This function’s responsibility is to clock the 8253 timer. Determination of when to clock the timer is based on whether its cycle limit has been reached or exceeded. The process used to count cycles and compare the count to its limit is similar to the VSYNC pulse control. On every processor loop iteration the `mainBoard.c` “e8253_timer_cycle_count” global variable is updated by adding the number of previous instruction cycles (“last_instruction_cycles”). The resulting sum is compared to the `mainBoard.c` preprocessor directive
constant “E8253_TIMER_CYCLE_LIMIT.” This constant is the clock cycle limit for the timer.

According to page 2.80 of Z-100 Technical Manual (Hardware), the timer is attached to a 250 kHz clock pulse. This translates to a clock pulse every 4 microseconds. As with the VSYNC pulse, the 5 MHz main processor clock, which outputs 0.2 microsecond (200 nanosecond) cycles, is used to gauge the cycle limit. To get a 4 microsecond clock rate, the “E8253_TIMER_CYCLE_LIMIT” constant is set at 20 cycles (20 * 0.2 = 4). Therefore, when “e8253_timer_cycle_count” has reached or exceeded 20, the timer is clocked by calling e8253.c’s e8253_clock() with the “e8253” timer object pointer passed as its first parameter and the value 1 as its second. As in the VSYNC clocking procedure, any extra cycles that put “e8253_timer_cycle_count” at a value greater than 20 are stored in a global variable to keep track of the overage. The timer cycle overage is assigned as the reset value for “e8253_timer_cycle_count” so that any extra cycles are not lost, and compounding errors can be avoided.

Hampa Hug’s e8253_clock() function is modified for this project. In the original version, all three of the 8253’s timer channels are clocked when e8253_clock() is called. Since Hug’s implementation of the 8253 timer is purposed for the IBM PC, this suggests that the CLK pins of all three channels are connected to the same exterior clock pulse in the IBM system. This is not the case for the Z-100. Only the CLK pins for channels 0 and 2 are connected to the external 250 kHz pulse; channel 1’s CLK pin is connected to the OUT pin of channel 0 (page 10.8, Z-100 Technical Manual – Hardware).
The Z-100’s timer circuitry makes it necessary to remove the logic advancing channel 1’s counter in Hug’s original `e8253_clock()` function. Thus, the e8253’s internal counter objects corresponding to only counter channels 0 and 2 undergo a check to confirm that they are in a “counting” mode. If so, the e8253.c function `cnt_dec_val()` is called with a pointer that references the appropriate 8253 internal counter object as its first parameter. The incoming value of 1 is passed as the function's second parameter. The `cnt_dec_val()` function checks that counter’s mode setting is valid (i.e. 0 – 5, or less than 6), and preforms the specified decrementing count operation in accordance with its mode and the counter's gate pin state. This is done through a specialized struct defined in e8253.c called “cnt_mode_t.” The struct has three function pointers that reference functions defined in e8253.c

An additional static struct, also defined in e8253.c, is constructed of a size-6 array whose elements are of the “cnt_mode_t” type. Each element of the array corresponds to one of the six possible counter modes. The elements themselves hold the three e8253.c functions wrapped in the “cnt_mode_t” object that carries out the specified mode’s behavior. The second of the “cnt_mode_t” object’s functions, “cnt_dec_val_X,” where ‘X’ is the counter's current mode, is called when the 8253 timer object is clocked in the mainBoard.c processor loop via the `e8253_clock()` function. The task of “cnt_dec_val_X” is to decrement the counter and determine if the terminal condition is met. If the condition is met, the function sets the channel’s OUT pin by a call to the e8253.c `cnt_set_out()` function.
There are two final conditional statements that check if counters 0 and 2 have expired. First the counter is verified to be in “counting” mode and then its value is compared to 0 (zero) to check if its terminal count has been reached. Special “e8253_t” type instance variables, “timerZero” and “timerTwo,” are set to 1 if their corresponding counter values have terminated. With the exception of its three “e8253_counter_t” counter objects, these are the only instance variable defined for the “e8253_t” timer object type. Variables “timerZero” and “timerTwo” are additions to the original Hug implementation for purposes of this project. The use of these variables is for the system to identify which timer channel causes the interrupt on the master 8259 IR2 line. These added variables are used in the custom function e8253_get_status(). This function is also an added modification to Hug’s original implementation. The function is called when the timer status port (0xFB) is read by the processor. The timer status port tells the processor which of the timer channels triggered the interrupt. This port read is necessary because both channels are tied to the same IR line on the master PIC. After values of the “timerZero” and “timerTwo” variables are determined, e8253_clock() returns to the mainBoard.c processor loop.

**FD-1797 Disk Controller Cycling**

The next device object to be clocked is the “JWD1797” floppy disk controller. Cycling the JWD1797 controller object is handled differently than the VSYNC pulse and the 8253 timer and is more complex. The doJWD1797Cycle() function, defined in jwd1797.c, takes two parameters. The first parameter is the
“JWD1797” object pointer (“jwd1797”), and the second is the previous instruction’s time slice in microseconds (“last_instruction_time_us”). According to the Z-100 Hardware Technical Manual (pages D.202 – D.204), FD-1797 read and write operations are timed based either a 1 or 2 MHz clock. However, instead of defining a cycle limit and timer variable based on the controller’s documented clock speed, instruction time slice values are passed to doJWD1797Cycle() directly. For example, if the previous instruction took 6 processor cycles at 5 MHz, the value 1.2 is passed as doJWD1797Cycle()’s second parameter (6 cycles * [0.2 microseconds/cycle] = 1.2 μs).

Virtual time slices provided directly to the “JWD1797” controller object are used to update its internal timing mechanisms according to the time state of the system. It is also important to note that the doJWD1797Cycle() function also updates the values of numerous “JWD1797” instance variables in addition to the internal timer counts. These updates are also described in the following discussion. All variables mentioned in the discussion are “JWD1797” instance variables unless otherwise noted. References to the FD-1797’s data sheet are found on pages D.184 through D.206 of the Z-100 Technical Manual (Appendices) documentation, where the complete Western Digital FD-179X data sheet is included.

**doJWD1797Cycle()**

Immediately upon entering doJWD1797Cycle()’s function body, the “JWD1797” variable “master_timer” is update with the incoming time slice. This
variable acts as a general overall timer and is used only for testing and debug purposes; it has no effect on the JWD1797’s internal mechanisms. Next, bit 7 of the “statusRegister” variable is updated. This variable represents the status register of the FD-1797 controller and its 7 bit is designated as the NOT READY bit for all controller command types. A logical OR operation is applied to the controller’s READY and MASTER RESET pin status. Bit 7 of the status register reflects the inverted OR operation result. The master reset is controlled by a NOT MASTER RESET pin from the computer interface, and the “ready” signal from the drive is communicated via the READY line. The condition for the drive being ready (i.e. status bit 7 set to 0) is an inverted read of the ready pin variable, “ready_pin,” having a value of 1 or the inverted value of “not_master_reset” being 1. If both variable reads have a value of 0, the “statusRegister” variable’s 7 bit is set to 1. The code block shown below handles the NOT READY status bit update. The “w” variable is the JWD1797 object pointer.
/* update status register bit 7 (NOT READY) based on inverted not_master_reset or’d with inverted ready_pin (ALL COMMANDS) */

if((!(w->ready_pin | w->not_master_reset)&1) == 0) {
    w->statusRegister &= 0b01111111; // reset NOT READY bit
}
else {
    w->statusRegister |= 0b10000000; // set NOT READY bit
}

Another “JWD1797” variable is updated next. This variable, named “not_track00_pin,” represents the “NOT TRACK 00” pin on the floppy drive. A check of the read/write head’s track position is used to update the “not_track00_pin” variable. The current track is stored in the “current_track” variable. If the value of “current_track” is 0x00, the head is located over track 0 and the “not_track00_pin” variable is set to 0 (zero). Any other value contained in “current_track” indicates that the head postion is not over track 0. In this case, the value of “not_track00_pin” is set to 1, signaling that the head is not positioned over track 0.

With the ‘forced interrupt’ command, the FD-1797 can immediately terminate an active command. A pending ‘forced interrupt’ causes the
“terminate_command” variable to be set to 1. The next process in the
\texttt{doJWD1797Cycle()} function detects if there is a pending command termination
condition and adjusts the appropriate variables. The process begins by checking
the value of “terminate_command.” If its value is 1, the process then checks if
there is command currently in progress. A flag variable, “command \_done,”
indicates whether the last command has finished. The “command \_done” flag’s
value is set to 1 at the beginning of a command’s execution and reset to 0 (zero)
when the command is complete. The termination process detects if there is a
command currently in progress by checking the value of the “command \_done”
flag. If there is an imminent termination condition (“terminate_command” = 1)
while a command is in progress (“command \_done” = 0), the command is
interrupted by setting the “command \_done” flag’s value to 1. The
“statusRegister” BUSY bit 0 is reset to 0 (zero) to indicate that the drive is no
longer busy. All other status register bits are left as their current values to
preserve any status conditions caused during the interrupted command’s
execution. If there is no command in progress under a command termination
condition, status bits corresponding to BUSY (bit 0), SEEK ERROR (bit 4), and
CRC ERROR (bit 3) are reset to 0 in “statusRegister.” Also, the
“currentCommandType” variable's value is set to 1, forcing “statusRegister” to
reflect a type I status.

FD-1797 commands are organized into four different types: I, II, III, and IV.
Apart from the NOT READY bit (bit 7), there is variation in the bit meanings of the
status register based on the current command type. If a termination condition is
applied when no command is being processed, the type of the last command received, stored in “currentCommandType,” is irrelevant and the command type is set to 1. As a result, “statusRegister” reflects a type I status. However, interrupting an active command requires that the command type and status bits be preserved. This is necessary because the status effects of the interrupted command may be important to the upcoming program processes.

An option allows the FD-1797’s ‘forced interrupt’ command to cause a vectored interrupt on the Z-100’s S-100 bus. The interrupt happens whether or not another command is active. The FD-1797’s interrupt request pin (INTRQ) is connected to one of the vectored interrupt lines (V0-V7) on the S-100 bus. Hardware settings determine the vector line used.

The next code section in the doJWD1797Cycle() function handles the ‘forced interrupt’ mode that causes the vectored interrupt. A check of the “interruptImmediate” variable is performed. This variable is a flag that signals the ‘forced interrupt’ command should also cause a S-100 vectored interrupt. The logic is mostly the same as the non-vectored command termination condition. The “command_done” flag is set as needed, the status register is updated, and the command type is changed if necessary. However, with the vectored interrupt mode active, the “JWD1797” “intrq” variable is set to 1 and the e8259_set_irq0() function is called with the slave interrupt controller pointer, “e8259_slave,” passed as its first parameter. Calling the e8259_set_irq0() function assumes the FD-1797 INTRQ line is connected to the V0 vector on the S-100 bus. V0 is connected to the IR0 pin on the slave 8259 interrupt controller.
The `doJWD1797Cycle()` function continues with resetting the "new_byte_read_signal_" variable to 0 (zero). This variable signals that a new byte has been encountered during the disk’s rotation. In the actual FD-1797 chip, when the read/write head encounters the start of a new byte, there is a period where the head reads the byte’s flux transitions into the FD-1797’s data shift register. The shift register assembles the byte one bit at a time. Once the byte is assembled, it is transferred in parallel to the FD-1797’s data register to be read by the computer. The flux transitions and data shift register process are not emulated. Instead, a timer is used to simulate the bit shift, assembly, and data register transfer process. The timer keeps track of the time between byte encounters. The timer's current value is stored in the "rotational_byte_read_timer" variable. Expiration of the timer happens when the value of "rotational_byte_read_timer" is greater or equal to the byte read time limit. As previously described, the byte read time limit is calculated in the `assembleFormattedDiskArray()` function when the number of formatted bytes per track is determined; its value is stored in "rotational_byte_read_limit."

To update and advance the byte read timer, the incoming time slice value passed to `doJWD1797Cycle()` from the processor loop is added to the value of the "rotational_byte_read_timer." However, the byte read timer limit is stored in units of nanoseconds and the incoming time slice is in units of microseconds. Given this unit discrepancy, the incoming time slice is multiplied by 1000.0 and cast to an integer. The converted value is added to "rotational_byte_read_timer," thereby advancing the timer.
A check to confirm if the rotational byte timer has reached or surpassed the byte read limit is conducted next. If the timer has expired, the process of advancing the track byte pointer and timer reset begins. Like the 8253 clock time process, if the rotational byte timer overshoots the byte read limit ("rotational_byte_read_timer" > "rotational_byte_read_limit"), the overage is stored. The overage is assigned as the reset value of the byte read timer at the end of the process.

Advancing the rotational byte pointer is accomplished by incrementing the "rotational_byte_pointer" variable. The value of "rotational_byte_pointer" should not exceed the number of formatted track bytes. Therefore, a modulo operation is applied to the newly incremented value. The modulo operation's argument is the total number of formatted track bytes ("actual_num_track_bytes"). This ensures the byte pointer remains in the range of 0 through one less that the number of formatted track bytes (0 <= "rotational_byte_pointer" <= ["actual_num_track_bytes" - 1]). It is important to note that the rotational byte pointer only indicates the rotational position of the read/write head on the disk; it does not provide information about which cylinder the head is over or which head is active.

When the "rotational_byte_pointer" variable’s value becomes 0 (zero), the beginning of the track is reached. A check for this condition is performed next in doJWD1797Cycle(). If the rotational byte pointer does indeed point to the first byte (byte 0x00) of a track, the “track_start_signal_” variable is set to 1. A feature of the FD-1797 causes the read/write head to disengage from the media if the
drive is not actively executing a command after 15 rotations of the disk. If the rotational byte value indicates the beginning of a track, the “HLD_idle_index_count” variable is incremented. The value is incremented only if the status register indicates the drive is not busy (status register bit 0 equals 0). The “HLD_idle_index_count” variable, representing the ‘idle head load count,’ holds the number of disk rotations occurring while the drive is not busy. If the drive is busy when the beginning of the track is reached, the “HLD_idle_index_count” variable’s value is reset to 0 (zero).

A verification mode is available for type I FD-1797 commands. Type II commands employ a verification sequence by default. Type I commands involve stepping the read/write head arm to adjacent tracks, and type II commands read from and write to the disk. The verification procedures allow the controller to confirm that the head is over the correct track, has the correct head active, and is at the appropriate sector via a read from the ‘cylinder,’ ‘head,’ and ‘sector’ byte of the address ID field. If the expected verification data is not received after a specified number of disk revolutions, the verification process times out. Bits indicating the relevant error are set in the status register. Inside the conditional block that runs when the rotational byte pointer has a value of 0 (zero), the “verify_operation_active” flag variable confirms that a verification process is active. If a verification process is active, the “verify_index_count” variable is incremented. If the disk completes a rotation while there is no verification process active, the “verify_index_count” variable is reset to 0 (zero).

Reaching the end of the block that is executed when the rotational byte
timer has expired, the flag variable “new_byte_read_signal_” is set to 1. This flag signals when a new byte is available. Comparing this with the actual FD-1797 hardware operation, when the “JWD1797” “new_byte_read_signal_” flag has a value of 1, it is equivalent to the FD-1797’s data shift register having completed its assembly of the byte. At this point, the FD-1797 is ready to transfer the byte from its data shift register to its data register. As such, the rotational byte timer is a stand-in for the bit shift assembly process. In addition to the flag being set, the rotational byte timer is reset by assigning the time slice overage amount to “rotational_byte_read_timer,” thereby accounting for time that put the timer past the limit.

Another ‘forced interrupt’ condition is handled in the next section of the doJWD1797Cycle() function. This option for the ‘forced interrupt’ command allows a termination condition to occur when an index pulse is detected. An index pulse is generated by the FD-1797 when the disk index hole is encountered. The beginnings of disk tracks are physically marked by the index hole. The “JWD1797” “track_start_signal_” flag having a value of 1 coincides with the start of an index pulse because both occur at the start of a track. Thus, if the “track_start_signal_” variable has a value of 1, and a ‘forced interrupt’ command is active with the index hole interrupt mode selected, the condition is met. The command termination and interrupt generation procedure are handled in the same manner as the previously described ‘force interrupt’ termination procedures.

There is a succession of events that occur when the disk’s index hole is
encountered by the FD-1797 controller. Logic must also be processed while the index pulse is active. A specialized function is called next in the \texttt{doJWD1797Cycle()} function’s body to handle these events and mechanisms.

The \texttt{handleIndexPulse()} function takes two parameters, the “JWD1797” controller object pointer as its first parameter and the incoming time slice from the mainBoard.c processor loop as its second. Entering \texttt{handleIndexPulse()}’s body, the “track_start_signal_” flag is used to recognize that the beginning of a track has been reached. The “JWD1797” “index_pulse_pin” variable, representing the FD-1797’s index pulse pin, is set to 1. Since the index pulse should have a specified time limit, the JWD1797 “index_pulse_timer” variable is used to keep track of the pulse’s timer. The timer variable’s value is set to the preprocessor constant “INDEX_HOLE_PULSE_US.” The constant is set to 100.0 microseconds. This is an arbitrary value. However, the time limit is sufficient based on the minimum required index pulse time reported by the FD-1797 data sheet. According to the data sheet, the index pulse should last for a minimum of 20 microseconds when the controller is clocked at 1 MHz.

If the index pulse is active (i.e. “index_pulse_pin” = 1), the “index_pulse_timer” variable is updated by subtracting the incoming time slice. Additionally, the status register must be updated if the current command is a type I command. The status register update is required because bit 1 of the register reflects the state of the index pulse pin. The next section of code in the \texttt{handleIndexPulse()} function executes these updates after a check of the “index_pulse_pin” variable returns a value of 1.
Immediately following `handleIndexPulse()`'s index pulse check code block, the function checks the state of the index pulse timer ("index_pulse_timer"). If timer's value is 0 (zero) or less, the "index_pulse_pin" variable is reset to 0 (zero) indicating the end of the index pulse. Like the last `handleIndexPulse()` block, bit 1 of the status resister variable, “statusRegister,” is set to 0. The `handleIndexPulse()` function then returns to `doJWD1797Cycle()` and continues the “JWD1797” cycling procedure.

The FD-1797 has a specific timing associated with the read/write head’s engagement with the disk media. To load the head against the media, the FD-1797 causes its ‘head load’ pin (HLD) to go “high.” However, the head is not immediately loaded. There is a delay between the time the drive receives the HLD signal and the time at which the head is loaded. After the delay, the head is loaded and the FD-1797’s ‘head load timing’ pin (HLT) pin goes “high.” Figure 26 is an illustration found in the FD-179X data sheet that describes the head load timing (page D.191, Z-100 Technical Manual – Appendices). As shown in the figure, when the HLD pin goes high, there is a delay of 50 to 100 milliseconds before the HLT pin goes “high” to indicate that the head has been loaded.

![Head load timing](image-url)
During command initialization for type I commands (restore, seek, step, step-out, step-in,) the head may be loaded at the time the command is received or sometime later during the command’s execution. In either case, the command causes the FD-1797’s HLD pin to go “high.” The “JWD1797” object represents the state of the HLD pin with the “HLD_pin” variable. When this variable's value is set to 1, another variable acting as a trigger to start the HLT pin timer, “HLT_timer_active,” is set to 1. After the handleIndexPulse() function returns to doJWD1797Cycle(), a function named handleHLTTimer() is called to process the HLT timer’s state and handle the head load timing. The function takes two parameters: the “JWD1797” object pointer and the incoming processor loop time slice.

The first task in the handleHLTTimer() function is to check the value of the “HLT_timer_active” variable. If its value is equal to 1, the HLD pin is “high,” and the head load delay is ongoing. The “JWD1797” variable “HLT_timer” tracks the head delay’s progress. The timer variable is updated by adding the processor loop time slice to its value. After adding the time slice, the updated value of “HLT_timer” is compared to the preprocessor constant “HEAD_LOAD_TIMING_LIMIT.” The constant’s value is set at 55,000 microseconds (55 milliseconds). This value is within the required delay range reported in figure 26. If the value of “HLT_timer” equals or exceeds “HEAD_LOAD_TIMING_LIMIT” (55,000), “HLT_pin” is set to 1 to indicate that the head has been loaded. The head load timer, “HLT_timer,” is reset to 0.0. The “HLT_timer_active” flag is also reset to 0 (zero) to indicate that the head load
timing delay is no longer active. The \texttt{handleHLTTimer()} function then returns to \texttt{doJWD1797Cycle()}. As indicated by figure 26, once the HLT pin goes “high,” it stays “high” until the HLD pin goes “high” again. Thus, the “HLT\_pin” variable’s value remains 1 until the head load timer becomes active again (i.e. “HLT\_timer\_active” = 1).

Status register bits that have not been updated are addressed next in the \texttt{doJWD1797Cycle()} function. A conditional block checks if the current command is a type I command. If the “currentCommandType” variable has a value of 1, the status register reflects a type I status. As such, bit 5 of the status register is the result of an AND operation applied to the state of the HLD and HLT pin. Therefore, the values of the “HLD\_pin” and “HLT\_pin” variables are combined with an AND operation. The result is set as bit 5 of “statusRegister.” Bit 2 of the type I status is set when the read/write head is positioned over track 0. Thus, bit 2 of “statusRegister” is set as the inverted value of the “not\_track00\_pin” variable.

When a type II or III command is the current command, bits 2 and 5 of the status register have different meanings as they do for type I commands. The next conditional statement in the \texttt{doJWD1797Cycle()} function checks if the “currentCommandType” variable’s value is either 2 or 3. Only one bit of the status register is updated inside this conditional block. Bit 1 of the status register reflects the state of the FD-1797 data request (DRQ) pin when the current command is type II or III. Therefore, bit 1 of “statusRegister” is set according to the value of “JWD1797” “drq” variable.
A crucial section of the *doJWD1797Cycle()* function then follows. The *jwd1797.c* *commandStep()* function is called if a command is currently being executed. A conditional statement checks the inverted value of the “command_done” variable to determine if a command is active. When the “command_done” variable equals 1, there is no command active. Therefore, the inverted value of “command_done” when it equals 0 (zero) indicates that a command is active. The *commandStep()* function takes the “JWD1797” object pointer as its first parameter and the incoming processor loop time slice as its second. All command processes are handled by this function. The active command is advanced based on the time slice passed from the processor loop and “JWD1797” instance variables are adjusted as needed.

After the *commandStep()* function returns to *doJWD1797Cycle()*, the *handleHLDIdle()* function is called. This function contains logic that determines if the read/write head’s engagement on the disk media has been idle for the timeout period. It takes the “JWD1797” object pointer as its single parameter. As discussed, the HLD pin will reset if the drive is not busy for 15 index pulses. Entering the body of the *handleHLDIdle()* function, the “HLD_idle_index_count” variable is compared to the “HLD_IDLE_INDEX_COUNT_LIMIT” preprocessor constant. The constant is set to 15 index pulses. If “HLD_idle_index_count” equals or exceeds the value of “HLD_IDLE_INDEX_COUNT_LIMIT,” the head load idle timeout occurs. The timeout event causes the “HLD_pin” and “HLD_idle_index_count” variables to reset to a value of 0 (zero). If the “HLD_pin” variable is reset, any new incoming command that requires the head to be
engaged must wait for the head load timing delay to pass before proceeding to
read from or write to the disk. A second condition in the handleHLDIdle() function
checks bit 1 of the “statusRegister” variable. As previously noted, bit 1 of the
status register relays the “busy” state of the drive across all command types. If
the value of this bit is 1, the drive is executing a command and the head load idle
count should be reset. “HLD_idle_index_count” is set to 0 (zero) in this case. The
handleHLDIdle() then returns to doJWD1797Cycle().

Updating the general control status of the “JWD1797” object is the final
task of the doJWD1797Cycle() function. The control status byte of the FD-1797
conveys the general operational status of the device. Interestingly, information
about the meaning of the control status bit positions is not included in the FD-
179X data sheet. However, the definitions are found on page 6.10 of the Z-100
Technical Manual (Hardware) and again as part of sample code given on page
6.16 of the same document. Figure 27 shows the bit definitions of the control
status byte appearing on page 6.10 of the manual. The only bits of concern for
the “JWD1797” emulator are those corresponding to the interrupt request
(INTRQ), MOTORON (5”), and data request (DRQ); bits 0, 1, and 7, respectively.
The updateControlStatus() function, which takes the “JWD1797” object pointer as
its single parameter, updates these bits.
The `updateControlStatus()` function’s body consists of a single line that applies an OR operation to three values and assigns the result to the JWD1797 "controlStatus" variable. The first argument involved in the OR operation is the current value of the variable representing the FD-1797 interrupt request line (INTRQ), the "intrq" variable. Bits 1 through 7 of "intrq" are zeroed by applying an AND operation to the variable and the value 1. The MOTORON bit of the control
status is set to 1. It is unclear what this bit indicates. However, setting it to 1 by bit shifting left by 1 bit allows the emulation to function properly. The third value is the “JWD1797” “drq” variable, which represents the controller’s data request signal (DRQ). Its bits 1 through 7 are masked with an AND operation retaining only the 0 bit. The resulting value is bit-shifted left by 7 to position its value into bit 7 of the status byte. Bits 2 through 6 of the “controlStatus” variable remain 0 (zero). The updateControlStatus() function then returns to doJWD1797Cycle(), which in turn, returns to the processor loop in mainBoard.c.

**Updating the Z-100 Screen**

Continuing with the processor loop in mainBoard.c, the updateZ100Screen() is called. This function updates the GTK window displaying the Z-100 screen. Immediately upon entering the function, a modulo operation is applied to the mainBoard.c global variable “instructions_done” inside a conditional statement. The modulo operation’s argument is 100,000, and the conditional statement triggers when the result of the operation evaluates to 0 (zero). Therefore, the body of the conditional block is executed every 100,000 instructions. The conditional statement’s body consists of two function calls, the first of which is a call to renderScreen(), defined in video.c. The function accepts two parameters: a “Video” object pointer and a pointer directed to the unsigned integer “pixels” array. The “Video” struct is defined in video.h and the “pixels” array holds the state of every pixel appearing on the Z-100’s standard 640 by 225 resolution display. The second call in updateZ100Screen()'s conditional
block is to the screen.c function \textit{display()}. This function triggers a GTK draw event to paint the pixels to the window.

Populating the pixel array is the main task of the \textit{renderScreen()} function. In the Z-100, video data is stored in three consecutive 64k memory pages starting at memory address 0xC0000. Each page stores information pertaining to one of the base red, blue, and green (RBG) colors. These video addresses are not written to or read from the mainBoard.c “ram” variable, which represents the system memory. Instead, they are stored in the video ram array variable, “vram,” which is defined as a “Video” object instance variable in video.h. The overall strategy is to read elements of the video RAM array (VRAM), process each color plane byte, construct a 24-bit RGB color, and write the RGB color into the pixel array. After the array is assembled, the pixels are painted to the GTK window by the screen.c \textit{display()} function.

Entering the \textit{renderScreen()} function, an integer variable named “displayed\_scan\_line” is declared and its value is set to 0 (zero). This variable keeps track of how many scan lines are \textit{actually} displayed to the screen. The Z-100 uses the specific video memory mapping scheme based on the current program’s video mode operation. The standard mode used by the system BIOS allows a 640 by 225 pixel resolution which accommodates 80 character columns by 25 character rows. However, other expanded display modes are possible. The Z-100’s video board has the ability to produce 640 by 525 interlaced displays in addition to multi-page display capabilities (page 4.2, Z-100 Technical Manual - Hardware). For this reason, some video memory addresses are not displayable
in the standard mode. Although the standard resolution consists of 225 pixel
scan lines, the video memory is organized to produce 400 scan lines when
considering the expanded video capabilities. This is taken into account in the
renderScreen() function’s logic.

Each one of the bits of each VRAM byte represents one pixel. A character,
being 8 pixels wide, consumes one VRAM byte for each of its height’s scan lines.
Since characters are 9 pixels in height, each character row takes 9 scan lines.
The character display scheme is shown in figure 28 (page 4.8, Z-100 Technical
Manual – Hardware). Following a character row (9 scan lines), there are 7 scan
lines that are not displayed. This results in an extra 7 scan lines for each group of
9 scan lines. An illustration of the scan line pattern accounting for the non-
displayed lines is shown in figure 29 (page 4.8, Z-100 Technical Manual –
Hardware). With 9 scan lines per character and the screen having a height of 25
characters, this equates to 225 scan lines. However, with the extra 7 lines per
row of characters, there are a total of 400 scan lines worth of VRAM addresses
(225 + (25 * 7)).

Figure 28. The Z-100 standard character display scheme. Each character is 8
pixels wide and 9 pixels high. The displayable area consists of 25 rows of 80
characters.
The `renderScreen()` function’s first loop cycles through the screen’s scan lines. As noted, not all scan lines are displayed in the standard Z-100 display mode. However, the loop runs for 400 (225 + [25 * 7]) iterations thereby accounting for every scan line addressed by the video memory. Scan lines that are not displayed appear as groups of 7 lines between each set of 9 lines that are displayed. This can also be interpreted as every consecutive group of 16 scan lines has only lines 1 through 9 displayed. A conditional statement appears...
just inside renderScreen()'s scan line loop that only selects displayed scan lines. It does so by masking all bits except the 4 low bits of the scan line loop variable, “actual_scan_line,” and checking if the result is less than 9 (0 through 8). For example, if “actual_scan_line” has the value of 19 (0x13) the scan line is selected because the four low bits have a value of 3. Alternately, if “actual_scan_line” has a value of 28 (0x1C), the scan line is skipped because its 4 low bits have a value of 12, which is greater than 8. Therefore, the scan line falls within a group of non-displayable lines.

Each character displayed to the screen in the standard display mode is eight pixels wide. As a result, the 640 pixels that make up a single scan line are divided among 80 characters (640/8 = 80). The next loop in renderScreen() is nested inside the scan line loop and it cycles through each character position in the current line. The loop’s iteration variable is “charXpos.” The first assignment made inside the character position loop is to the “starting_byte_number” variable. This variable holds the starting VRAM byte for the current scan line. It is important to note that for each consecutive section of 128 VRAM memory addresses, only 80 bytes are used to display pixels on the current scan line. These 80 bytes correspond to the 80 characters making up the width of scan lines. For instance, the first scan line is constructed with bytes 0 through 79 of VRAM. The next 48 bytes are skipped. The second scan line begins with VRAM byte 128 and ends with byte 207 (128 + 79). Again the next 48 bytes are skipped before moving on to scan line 3, which starts at byte 256. Since each scan line starts with an address that is 128 bytes from the start of the previous scan line,
the starting byte number for any given scan line is the scan line number multiplied by 128. Therefore, to get the current scan line’s VRAM start address, “starting_byte_number” is set to the value of “actual_scan_line” shifted left by 7 bits. Shifting left by 7 bits is equivalent to multiplying by 128.

With the starting VRAM address calculated and stored into the “starting_byte_number” variable, the position of the character currently being rendered is added. The sum is stored into the “raw_addr” variable. This variable holds the VRAM address of the byte corresponding to the current character on the scan line being rendered. A third nested loop inside the character position loop cycles through each bit of the VRAM byte pointed to by the “raw_addr” variable. For each bit, three integer variables are declared for bits gathered from each of the three color planes. The “blue” variable’s value is set to current bit of the “vram” array byte indexed by “raw_addr.” The “red” variable is similarly assigned, except 0x10000 is added to the “vram” array index (“raw_addr” + 0x10000) to point to the next 64k memory page, as each color plane is represented by consecutive 64k pages. Likewise, the “green” variable is assigned with 0x20000 added to the “vram” array index to access the green color plane.

After each color bit assignment, the status of the ‘color enabled’ option is checked. These ‘enabled’ options are represented by “Video” object instance variables defined in video.h. If the color plane is not enabled, the color bit variable is set to 0 (zero) and the pixel color is turned off. For example, if the “blueenabled” variable’s value is 0, the value of the “blue” bit is forced to 0 (zero).
The code listing below shows the bit assignment logic. The “v” variable is the “Video” object pointer. The “vram,” “blueenabled,” “redenabled,” and “greenenabled” variables are all instance variables of the “Video” object.

```c
int blue = (v->vram[raw_addr]>>bit)&1;
if(!v->blueenabled) blue=0;
int red = (v->vram[raw_addr+0x10000]>>bit)&1;
if(!v->redenabled) red=0;
int green = (v->vram[raw_addr+0x20000]>>bit)&1;
if(!v->greenenabled) green=0;
```

After the color bits are assigned, the value of the Video object’s “flashenabled” instance variable is checked. This variable represents the signal that indicates whether the ‘flash enabled’ mode is engaged. When this mode is active, all color bits are forced to 1. Since each displayed pixel is an overlay of the three colors, the pixel appears white on the screen when all color bit values are set to 1. All VRAM bytes are read as 0xFF while the ‘flash enabled’ mode is on. As a result, the entire screen appears white.

A 24-bit RGB value is constructed from the color bit values. To do this, the next code section in the bit loop converts the color bit variables to 8-bit values and then combines them into a 24-bit integer. Converting the color bit values is a
simple process. If the color bit has a value of 0, it remains 0. Alternatively, if the color bit’s value is 1, its value is converted to 0xFF. The 24-bit RGB color is constructed by shifting the converted “red” variable left by 12 bits and the “green” variable left by 8 bits. The “blue” color bit variable is not shifted; its value accounts for the low 8 bits of the 24-bit color. The shifted values of “red” and “green,” and the initial value of “blue” are combined with an OR operation. The result is stored into the “twentyFourBitColor” variable. This variable is stored as an element of the “pixels” array.

Some manipulation is required to place the newly constructed 24-bit color value in the correct “pixels” array index. The expression shown below calculates the proper index placement of a given “twentyFourBitColor” value.

\[
(displayed\_scan\_line*VWIDTH) + ((\text{charXpos}*8)+(7\text{-bit}))
\]

Considering the current character position first, the \([\text{charXpos} \times 8]\) portion of the expression advances the pixel index to the start of the next set of character bits for the scan line. Since the “charXpos” variable notes the current character column being rendered, and there are 80 characters per scan line, this part of the expression advances the index by 8 for each character. Next, the \([7\text{-bit}]\) portion fills in the bit data backwards. This is necessary because each byte is read from the most significant byte to the least. For example, for the very first set of 8 bits
(i.e. the first byte read from VRAM), bit 0 is put into pixel array index 7, bit 1 into pixel array index 6, bit 2 into index 5, and so on until bit 7 is placed into the first pixel array index. It could be said that the byte is “dropped into place” and each of its bits falls into the indices underneath. The graphic below (figure 30) illustrates this concept for the first two VRAM bytes.

![Diagram of VRAM byte placement](image)

Figure 30. Placement scheme of the VRAM bits into the "pixels" array.
Placement of the first two bytes is shown.

Finally, the current scan line being rendered is considered when targeting the appropriate “pixels” array index. To adjust the target index according to the current scan line, the value of the “displayed_scan_line” variable is multiplied by the pixel width of the Z-100 displayed screen area, VWIDTH. The VWIDTH constant is defined in video.h as the value 640. The “displayed_scan_line” variable is incremented each time a scan line has finished rendering. Therefore, the result of the expression \([displayed\_scan\_line * VWIDTH]\) points to the “pixels” array index corresponding to the first pixel in the scan line row. The row number
is congruent to the “displayed_scan_line” variable’s value.

When all iterations of the bit loop are complete for each iteration of the character byte, and all iterations of the character byte loop are have terminated, the “displayed_scan_line” variable is incremented. In other words, “displayed_scan_line” is incremented each time a displayed scan line is rendered. Once all actual scan line loop iterations are complete, the renderScreen() function returns to the updateZ100Screen() function in mainBoard.c.

Having rendered the Z-100 screen via the renderScreen() function, display() is called next in the updateZ100Screen() function. The display() function is defined in screen.c. This function consists of a single call to the GTK gtk_widget_queue_draw() function. The GtkWidget pointer “window” is passed as its single parameter. The call to gtk_widget_queue_draw() triggers a draw event by causing a cascading call chain that continues with gtk_widget_queue_draw_area() and ends with a call to gtk_widget_queue_draw_region(). Instead of explicitly specifying a draw area, the gtk_widget_queue_draw() function is a specialized version of gtk_widget_queue_draw_area() that indicates drawing to be done for the entire area of the “window” GtkWidget. When gtk_widget_queue_draw_region() is called, a “draw” signal is triggered. The screenInit() function assigns on_draw_event() as the callback function for the “draw” signal. Therefore, the function call cascade starting with gtk_widget_queue_draw() leads to a call to screen.c’s on_draw_event() function.
The on_draw_event() function begins with the declaration of four local variables. The first variable is “p24BitColor,” an unsigned integer that holds the 24-bit color value read from the “pixels” array. The next three variables are of the unsigned character type and hold the separated 8-bit color components of the 24-bit color value. After the variables are declared, renderScreen() may be called again. It is optional that renderScreen() is again called since it is already called in the updateZ100Screen() function. It may be called in either location or both locations. It is unclear which method is more efficient. During the emulator’s development the function is called in both locations.

After the renderScreen() function returns (if it is called in on_draw_event()), a double nested loop structure begins to extract 24-bit color values from the “pixels” array. The outer loop cycles through each of the Z-100’s 225 pixel rows and the inner loop cycles through each of its 640 pixel columns. The row loop increments its “row” loop variable each iteration and the column loop increments its “column” loop variable with each of its iterations. Both loop variables start with a value of 0 (zero).

Entering the column loop during a given row iteration, an element is extracted from the “pixels” array based on the value of the “row” and “column” variables. The “row” value is multiplied by 640 to point to the array index corresponding to the first pixel of the given row. The “column” variable’s value is added to the product of the row multiplication to yield the appropriate array index. A 24-bit color element is accessed from the “pixels” array at the calculated index and its value is stored into the “p24BitColor” variable. Eight-bit color component
values are then extracted from “p24BitColor.” Being the highest 8 bits of the 24-bit color, the red component is attained by right shifting “p24BitColor” by 16 bits and applying an AND operation to the result thereby masking all other bits except the lowest 8 bits. The result is stored in the “red_val” variable. The process is repeated for the green and blue components except that “p24BitColor” is shifted right 8 bit for the green component. No shift is required for the blue component. The “green_val” variable is set to the green component result and “blue_val” is set to the value of the blue component.

Pixel graphics are created using the Cairo vector graphics library. [33] Each pixel is drawn as a “cairo_t” object. First, the RGB data extracted from the 24-bit color is passed to `cairo_set_source_rgb()`, a Cairo library function that sets the RGB sources for a “cairo_t” object. The function takes four parameters: a “cairo_t” object as its first parameter and double type values within the range 0.0 through 1.0 for its next three parameters. The double type values represent the intensity of the “cairo_t” object’s RGB color components. The “red_val,” “green_val,” and “blue_val” values are passed to the function’s color intensity parameters. Even though the color component variables are of the type ‘unsigned character,’ the `cairo_set_source_rgb()` function clamps down values to a maximum value of 1.0. A color component value of 0x00 requires no modification since the minimum color value of the function is 0.0. Components that have a value of 0xFF are clamped to 1.0.

The Cairo function `cairo_rectangle()` is used to create a rectangle for each pixel. The function accepts four parameters. Its first parameter is the “cairo_t”
object. The second and third parameters are the X and Y location of the rectangle’s upper left corner in the drawing area. The last two parameters set the width and height of the rectangle. Two global constants defined in screen.c are used to scale the size of the pixel rectangles. The “X_SCALE” constant determines the host display pixel width of a single Z-100 screen pixel, and the “Y_SCALE” constant determines the host display’s pixel height. Therefore, to get the actual location of a rectangle for any given pixel, the “column” variable is multiplied by “X_SCALE” to get its X position and “row” is multiplied by “Y_SCALE” to determine its Y position. The host-displayed rectangle width parameter passed to cairo_rectangle() is simply the “X_SCALE” constant value. Likewise, “Y_SCALE” is passed as the rectangle’s height. After cairo_rectangle() draws the rectangle for the loop’s current pixel, the cairo_fill() function fills the rectangle with the color sourced by the cairo_set_source_rgb() function. This process is repeated for every pixel element in the “pixels” array through all iterations of the column inner loop and outer row loop. After the last row loop terminates, the on_draw_event() returns a “FALSE” boolean value. The display() function returns to the mainBoard.c updateZ100Screen() which in turn returns to the processor loop.

handleDebugOutput()

Reaching the last function in the processor loop, handleDebugOutput() is called to execute debug print statements. These print statements output information to the terminal when either debug mode is active. The function also
pauses execution in order to step through assembly instructions. As with other
dump code sections, the `handleDebugOutput()` function body begins with
checking if debug mode 1 is active. To confirm the active mode, the `mainBoard.c`
global variable “debug_mode” is checked to see if its value is 1. If debug mode 1
is active, the “instructions_done” variable is compared to the user-defined
“breakAtInstruction” value. If the instruction count is equal to or exceeds the
break point set by the user, the debug condition is met. Alternatively, if debug
mode 2 is active (“debug_mode_2_active” = 1), the debug output condition is
met. Emulator state information is printed to the terminal, specifically the number
of instructions done, the last instruction time slice, and the amount of virtual time
that has passed since emulator startup.

Another function call is included in `handleDebugOutput()` that prints a
detailed description of the “JWD1797” disk controller object's state. The
controller’s debug output function is `fD1797DebugOutput()`; it is defined in
`mainBoard.c`. This function prints the values of numerous “JWD1797” instance
variables, including the rotational position of the read/write head, various register
values, counts related to ID data field detection, and internal timer values.

Immediately after the `fD1797DebugOutput()` function returns to
`handleDebugOutput()`, the standard C-library `getchar()` function is called within a
while-loop conditional statement. The blocking nature of the `getchar()` function is
used to pause execution of the processor loop until the user presses the
‘ENTER’ key. The while-loop continues to call `getchar()` until it returns the
newline character (‘\n’). If the processor loop is paused at a break point and the
user presses the ‘ENTER’ key, `handleDebugOutput()` returns to the processor loop and another iteration is executed until the `getchar()` while-loop is again reached. In the case that a debug mode is active, but the break point is not reached yet, `handleDebugOutput()` returns without effect and the processor loop continues until the break point is reached, at which point the user must press ‘ENTER’ to continue. If the emulator is operating in ‘normal’ mode, `handleDebugOutput()` returns without effect. The processor loop continues forever until the emulator program is manually terminated in the terminal or the Z-100 screen window is closed.
IV. FD-1797 Implementation Testing

A novel implementation of the Western Digital FD-1797 floppy disk controller is integrated into this Z-100 emulator project. The controller implementation is developed from scratch using the “FD 179X-02 Floppy Disk Formatter/Controller Family” data sheet as a guide (page D.184-D.206, Z-100 Technical Manual – Appendices). While building the implementation, a custom testing environment is also developed to ensure added features function as expected. The testing environment constitutes a stand-alone project apart from the Z-100 emulation. Code for the testing implementation is found at the following GitHub repository: https://github.com/jmatta697/WD1797_C_Implementation.git. Reference should be made to the code in this repository while reading through the following discussion.

Two files in the project are central to the testing environment’s operation: testMain.c and testFunctions.c. Each feature of the FD-1797 controller is tested by a function that verifies the feature’s expected behavior. Data sheet feature descriptions are used to verify behavior. Test functions are defined in testFunctions.c. These functions are called sequentially in testMain.c. The testMain.c file includes the testing environment’s main entry point (main()). Before tests are run, main() sets up variables used by the test functions.

Since this project’s goal is to load the operating systems from disk, ‘write’ functionalities of the FD-1797 are not implemented. Therefore, the controller’s
WRITE SECTOR and WRITE TRACK commands are not included or tested. Additionally, the FORCED INTERRUPT commands that trigger when the drive transitions from “ready” to “not ready” and “not ready” to “ready” are not tested or implemented. These FORCED INTERRUPT commands are not relevant because the drive is always “ready” in this implementation. It is unclear whether the exclusion of these interrupt commands is detrimental to operating system initialization.

**testMain.c**

The `main()` function in testMain.c begins by initializing a new “JWD1797” controller object via the `newJWD1797()` function. Reference to the object is assigned to testMain.c’s global “JWD1797” pointer variable “jwd1797.” The pointer’s value is then printed to the terminal to verify that a valid object is created. Next, an array of length 7 is declared and initialized. Seven ‘double’-type values are set as the array’s elements. These values represent seven possible time slices in microseconds (µs) that could be passed to the `doJWD1797Cycle()` function during the Z-100 emulator’s processor loop. The array’s values range from 0.8 to 4.0 µs, correspond to resulting time slices from instructions that take 4 to 20 processor cycles clocked at 5 MHz (0.2 µs per cycle).

Within the test functions, a time slice is chosen randomly and passed to `doJWD1797Cycle()` during simulated processor loops. Time slices are chosen randomly to simulate varied instructions being executed as part of a typical assembly program. As such, the random number generator is seeded with the
system Unix epoch time passed to the standard C-library `srand()` function. After seeding the random number generator, the test function calls begin. As noted, all test functions are defined in testFunctions.c.

**masterClockTest()**

Within a for-loop with 10 iterations, the master clock test takes randomly chosen values from the time slice array and passes them to the `doJWD1797Cycle()` function. In a single iteration, the random time slice value is printed to the terminal before it is passed to the `doJWD1797Cycle()` function. The function updates the “master_timer” instance variable of the “JWD1797” object by adding the incoming time slice. To verify the master clock is updated appropriately, the value of “master_timer” is printed to the terminal; it should reflect the cumulative effect of previously added time slices.

The master clock test verifies that time slices are chosen randomly, the master timer mechanism is functioning properly, and the “JWD1797” object is receiving the correct time slice value. By visually inspecting printed values of the chosen time slices and the resulting master timer counts, proper operation is confirmed. Figure 31 shows printed terminal results of a `masterClockTest()` call. Since the cumulative sum of the master timer accurately reflects the incoming time slices, the test is passed. As figure 31 shows, each iteration output prints a random time slice and then the state of the “JWD1797” object’s master timer.
The next test function called in `main()` is `getFByteTest()`. This test confirms that the `getFDiskByte()` function returns the correct value based on the disk’s relative rotational position. Within a single iteration of a 200-iteration loop, a random instruction time slice is passed to the `doJWD1797Cycle()` function advancing the rotational byte pointer when appropriate. If the read/write head is over a new byte location as indicated by the “JWD1797” “new_byte_read_signal_” variable, the test process is triggered.

Two print statements begin the test process. The first print statement displays the rotational byte pointer location, and the second reports the status of the master timer. After these print statements, the `getFDiskByte()` function is
called. Hard-coded values for the current cylinder ("current_track") and active head ("sso_pin") determine the head’s positioned before the test starts. The \texttt{getFDiskByte()} function retrieves a formatted disk byte based on three criteria: the read/write head’s cylinder position, the active head, and the disk’s rotational position relative to the head. Based on these criteria, the byte read by the head is returned. The current byte is then \textit{explicitly} retrieved based on the same relevant disk position criteria without use of the \texttt{getFDiskByte()} function. A comparison is made between the explicitly retrieved byte and the byte returned by the \texttt{getFDiskByte()} function. If the bytes match, the \texttt{getFDiskByte()} function returned the correct value and a confirmation message is printed to the terminal. Mismatching bytes result in “WRONG BYTE READ” being printed to the terminal. Depending on the random time slice values chosen, the number of bytes read is around 10 to 12.

Figure 32 shows the output of a single \texttt{getFByteTest()} function run with "current_track" (cylinder) set to 19 and the “sso_pin” (active head/disk side) set to 1. An active head of 1 means the head is reading side 2 of the disk. The test results shown in figure 32 indicate a successful test. By default, the disk starts rotating at byte 2500 at startup/reset, and for this reason, rotational byte 2501 is first to be read.
commandWriteTests()

To confirm that incoming command bytes are interpreted properly, the commandWriteTests() is called. Incoming bit patterns are translated to corresponding commands based on the chart shown in figure 33 (page D.190, Z-100 Technical Manual – Appendices). Table B, indicated by the red rectangle in figure 33 applies to the FD-1797 model used in this project.
The `commandWriteTests()` function writes commands to the “JWD1797” object’s command port (0xB0). Flag values related to the command are also printed to the terminal. Every command appearing in figure 33 is tested along with many command options. Before each individual command test is performed, the controller is reset using the `resetJWD1797()` function. The reset function prints its standard disk information report to the terminal before each command is sent to the controller object. Verifying a successful test is a matter of matching the reported command received with the command byte passed to the command port. Flag variables are noted and compared to the command options specified in the written command byte.

Figure 34 shows code blocks and their corresponding outputs for two RESTORE command test variations. The results reported in figure 34 demonstrate successful tests because the output messages and reported option
variables are consistent with the bytes written to the command port. Figure 35 shows code blocks and their corresponding test results for two FORCED INTERRUPT commands. Like the RESTORE command results, the tests demonstrated in figure 35 are successful because expected status outputs match the command writes. Results for the remaining tests of the `commandWriteTests()` function are also successful; the entire test passes when run.

Figure 34. Two RESTORE command variation test code blocks (left) and successful results reported by the `commandWriteTests()` function (right)
As shown in figure 36, an index hole is positioned at the start of 5.25” floppy disk tracks. The FD-1797 controller detects when the index hole is reached and registers the detection in the status register when a type I command is active. Type I commands include RESTORE, SEEK, STEP, STEP-IN, and STEP-OUT (figure 33). When the status register reflects a type I status, bit 1 of the status byte indicates that the index pulse is active. The index pulse is conveyed from the drive to the controller via the controller’s not index pulse (IP) input as described in figure 37 (page D.204, Z-100 Technical Manual – Appendices). Bit 1 of the type I status byte is an inverted copy of the IP input. When the bit equals 1, the index hole is being detected. The index pulse should
last for at least 20 µs when using a 1 MHz clock for the controller (page D.203, Z-100 Technical Manual – Appendices).

Figure 36. The 5.25" floppy disk index hole is indicated by the red arrow (left). The index hole marks the beginning of tracks. The start of all tracks is indicated by the red line positioned next to the index hole (right).

<table>
<thead>
<tr>
<th>STATUS FOR TYPE I COMMANDS</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIT NAME</td>
</tr>
<tr>
<td>----------</td>
</tr>
<tr>
<td>S7 NOT READY</td>
</tr>
<tr>
<td>S6 PROTECTED</td>
</tr>
<tr>
<td>S5 HEAD LOADED</td>
</tr>
<tr>
<td>S4 SEEK ERROR</td>
</tr>
<tr>
<td>S3 CRC ERROR</td>
</tr>
<tr>
<td>S2 TRACK 00</td>
</tr>
<tr>
<td>S1 INDEX</td>
</tr>
<tr>
<td>S0 BUSY</td>
</tr>
</tbody>
</table>

Figure 37. Highlighted with a red box, bit 1 (S1) of the type I status appearing in the FD-1797’s status register indicates an active index pulse. If the bit has a value of 1, the index pulse is active.
The `indexPulseTest()` function assists in ensuring the status register reflects the proper index hole transition when the beginning of a track is encountered. The function also verifies that the appropriate amount of time has passed before the index pulse expires. An internal “JWD1797” object constant ("INDEX_HOLE_PULSE_US") defines the timer limit for the index pulse. The constant is set to 100 µs, which is sufficient to satisfy the minimum index pulse time required for proper operation.

Entering the `indexPulseTest()` function, the “JWD1797” object is first reset and then “current_track” is initialized to 5. This forces the read/write head over cylinder 5. A RESTORE command is then written to the command port (0xB0). On each iteration of a 150,000-iteration loop, a random time slice is passed to the `doJWD1797Cycle()` function. Almost all iterations of the loop pass with no output to the terminal. Output begins to print when the rotational byte pointer reaches two bytes before the start of the track. A number of “JWD1797” object instance variables related to the index pulse are printed. The output continues until the rotational byte pointer is positioned at track byte number 2.

Since the master clock time and status register byte are printed to the terminal, the index pulse duration can be verified. Furthermore, the index pulse timer status is also printed, allowing the index pulse timing to be verified by a second source. Figure 38 (right) shows terminal output of the test loop iterations where the IP timer expires and bit 1 of the status register transitions from 1 to 0. This bit transition indicates the end of the index pulse. The left portion of figure
38 shows the IP start. At the pulse’s start the IP timer begins and the status register bit 1 transitions from 0 to 1. A successful test of the index pulse mechanism is demonstrated in figure 38.

Additionally, the FORCED INTERRUPT command is tested with the IP interrupt option enabled. The command is written to the “JWD1797” object's command register at iteration 6,000 of the indexPulseTest() test loop. Figure 38 (left) shows the interrupt occurring when the index pulse begins. This indicates that the FORCED INTERRUPT command with enabled index pulse option is functioning as expected.

Figure 38. Terminal test output for the indexPulseTest() function showing the start of the index pulse and the IP forced interrupt condition being met (left). Terminal test output for the indexPulseTest() function showing the expiration of the index pulse timer and the transition from an active IP to inactive via bit 1 of the status register (right).
restoreCommandTest()

The next "JWD1797" controller object test verifies that the RESTORE function is operating as expected. Regardless of the read/write head's cylinder position, the RESTORE command steps the head back to cylinder 0x00. Several step timings are available. However, the Z-100 monitor ROM BIOS sets the step timing to 30 milliseconds (ms) during the BOOT command sequence (page 1-14 – BOOT_207 – Z100 ROM Listing 2.5 Volume II). As such, 30 ms step timing is used in the test.

Two RESTORE command variations are tested in the restoreCommandTest() function. The first variation does not load the head when the command starts and applies a 30 ms verify delay at the end of the command. Similarly, a 30 ms verify delay is applied for the second variation, but the head is loaded at the beginning of the command. Head loading takes 55 ms.

As with previous tests, the "JWD1797" controller object is reset prior to starting each of the RESTORE variation tests. For both variations, the "current_track" (cylinder) and "trackRegister" variables are forced to 3. This causes the head to start above cylinder 3. The track register must be updated because the verification procedure associated with the RESTORE command uses the track register to confirm that track 0x00 is reached.

After the "JWD1797" variables are set, the RESTORE command is written to the command port. A 500,000-iteration loop then commences. Again, as with other tests, a random time slice is passed to the doJWD1797Cycle() function for each iteration. As the step timer approaches each of the three 30 ms step
intervals necessary to reach cylinder 0x00, controller state information is printed to the terminal. The displayed information allows the controller’s operation to be monitored while the head steps to the next cylinder. To view the verify timer’s expiration, an additional 30 ms interval is investigated after the command's completion. Head load timing expires 55 milliseconds after cylinder 0x00 is reached. Therefore, an additional 55 ms interval is examined to view the head timer’s expiration. For the second RESTORE command variation, the head load timer is started when command execution begins. As a result, an additional 55-ms offset interval is not examined after cylinder 0x00 is reached in the second test.

Figure 39 shows terminal output during cylinder transitions. Verify timer expiration and head load delay timer expiration are indicated in the output. Since these transitions happen at the appropriate timings and the internal signals, track resister, and status register are updated properly, the test is successfully passed.

**seekCommandTest()**

The SEEK command steps the head to a destination track from the current track position. In keeping with the 30-ms step timing set by the BIOS’s boot initialization procedure, the SEEK command test outputs terminal information at 30-ms timer intervals. One SEEK command variation is tested in the `seekCommandTest()` function. The head is loaded at the start of the command and the verify option is enabled. Therefore, it is not necessary to monitor expiration of the head load timer 55 ms after the target track is found.
Since the verify option is enabled, an additional 30-ms interval is examined after the target track is located.

![Figure 39. Terminal output for the `restoreCommandTest()` when starting from track 3. Beginning at the top left pane, cylinder transitions are shown for each transitions from cylinder 3 to 0. The bottom panes show the verify delay timer expiration (left) and the head load timer expiration (right). These results demonstrate a successful RESTORE command test.](image-url)
Entering the `seekCommandTest()` function, the “JWD1797” controller object is first reset using the `resetJWD1797()` function. The “current_track” variable and the track register are both set to 0x07. A value of 0x05 is written to the data register. The SEEK command uses the value held in the data register as its destination track. With this setup, the test ensures that the SEEK command steps from cylinder 5 to cylinder 7 and then updates the track register with the destination track. Testing the SEEK command is similar to the RESTORE command test procedures in that verify and head load timings are identical in both commands. Since the head load and verify options are tested in the RESTORE command test, they are not examined in detail for the SEEK command.

As shown in figure 40, successful transitions are reported for the jump from track 7 to 6 and again from track 6 to 5. All flags are appropriately set, and registers are updated accordingly. The verify timer also expires 30 milliseconds after the command action is complete. Figure 40 therefore demonstrates a successfully passed SEEK command test.
The STEP command functions like the SEEK command except that it executes a single cylinder transition instead of multiple transitions. The direction of the last step is carried over to the current STEP command. For example, if the previous command were “SEEK track 5 from track 7,” issuing a subsequent STEP command would execute one more step in the same direction, towards track 0x00. Therefore, the STEP command would cause the head to transition from cylinder 5 to cylinder 4 following the SEEK command.

Again, as in previous tests, a random time slice is passed to the
doJWD1797Cycle() function during each iteration of the 500,000-iteration loop included in the stepCommandTest() function. A single track transition is examined at the first 30-ms time interval. Relevant information is printed to the terminal to make certain the transition mechanism specific to the STEP command is functioning as expected. Head load and verify timings are not examined. These timings are handled by the same “JWD1797” object code with all type I commands. Since the head load and verify timing mechanisms are tested in previous type I tests, it is not necessary to examine them again.

For the stepCommandTest() function, the “current_track” variable is set to 6. The “direction_pin” variable is set to 1 to force the step direction to “out,” away from track 0x00. In this case, the STEP command transitions from track 6 to track 7. Figure 41 (left) shows a successful STEP command transition from track 6 to 7. This indicates that the STEP command implementation is functioning as expected.

The STEP-IN and STEP-OUT commands function in the same way as the STEP command except they force the ‘direction’ pin to a state that allows the commands to carry out their respective step procedures. In the case of the STEP-IN command, the “direction_pin” variable of the “JWD1797” controller object is forced to a value of 1. When the ‘direction’ pin’s state is 1, the head is stepped to the center of the disk, toward cylinder 39 in a 40 cylinder disk. Setting the “direction_pin” variable to 0 allows the head to be stepped away from the center of the disk, or toward track 0x00. In both the stepInCommandTest() and stepOutCommandTest() functions, “current_track” is set to 5 before the controller
is clocked in their test loops. For `stepInCommandTest()`, the head is stepped from cylinder 5 to 6. Alternatively for `stepOutCommandTest()`, the step is from cylinder 5 to 4.

Terminal output reporting cylinder transitions for the `stepInCommandTest()` and `stepOutCommandTest()` functions is shown in figure 41 along with the STEP command test result. In all three command tests the expected behavior is observed. A value labeled “Direction” in the terminal output is indicated with a red arrow. This value is the state of the “direction_pin” variable. This variable represents the actual ‘direction’ pin output from the FD-1797 controller to the floppy disk drive. The test results show that the “direction_pin” variable is consistent with the step direction carried out by the commands. Also, because the “track update” flag is set for all three commands, the track register reflects the destination cylinder when it is reached. The track register value set after the command terminates is highlighted with a red box in each of the command test terminal outputs.
readSectorTest()

A RESTORE command is issued to the “JWD1797” controller object as the first task of the readSectorTest() function. The RESTORE command is advanced to completion by passing random time slices to the doJWD1797Cycle() function within a 100,000-iteration loop. After the command completes, the values contained in the status register and track register are printed to the terminal to confirm that the head is settled at track 0x00.

Following the RESTORE command, a SEEK command is used to place the head above cylinder 3. A value of 0x03 is explicitly written to the track register port (0xB3) and the SEEK command is then written to the “JWD1797” object’s command port. Like the RESTORE command, the SEEK command is
advanced to completion by passing random time slices to the `doJWD1797Cycle()` function within a 100,000-iteration loop. To confirm the command's completion, the status register's value is printed to the terminal. Verifying that the head is above cylinder 3, the track register's value is also printed.

Byte data is then retrieved from the “Z_DOS_ver1.bin” disk image file. This file contains only sector payload data appearing on the 5.25” floppy boot disk for version 1.0 of Z-DOS. The disk image file does not include format bytes. All bytes contained in the file are read into an array using the `diskImageToCharArray()` function defined in jwd1797.c. The array is assigned to `readSectorTest()`'s “payload_test_data” local variable. This array is used as a control group for testing the READ SECTOR command. The bytes contained in the array are assumed to be an accurate representation of the sector payload data appearing on the Z-DOS boot disk. Each sector payload byte read by the READ SECTOR command is compared to its corresponding byte in the test data control group array.

The first READ SECTOR command variation to be tested has the 30-millisecond “delay” and “multi-record” options enabled. The “delay” option causes a 30-ms delay before the head’s engagement to the disk media is checked. The command procedures do not continue until the head is engaged. The 30-ms delay provides an extra time buffer for the head engagement. Sectors are read until the end of the track is reached when the “multi-record” option is enabled.

Sector 7 is targeted by explicitly writing 0x07 to the sector register. Read/write head 2 is activated by setting bit 1 of the of the READ SECTOR
command byte to 1. Since the previously executed SEEK command positioned the head over cylinder 3, and head 2 is active, formatted track 7 is targeted. A value of 0x07 in the sector register means that the READ SECTOR command starts reading at sector 7 of formatted track 7. “Multi-record” being enabled means that the command continues to read sectors until the end of the track is reached. “Z_DOS_ver1.bin” is an image file of a DOS-format 320k disk. This disk format provides 8 sectors per formatted track. Therefore, the first READ SECTOR test reads sectors 7 and 8 of formatted track number 7. Before the test begins, a local pointer variable, “test_byte_pointer,” is set to the starting index of the “payload_test_data” array. The starting array index is determined by the target starting sector. In doing so, “test_byte_pointer” should point to the element in “payload_test_data” that corresponds to the first sector payload byte read from the formatted disk.

The first READ SECTOR command test begins with a 300,000-iteration loop. As with all previous tests, random time slices are passed to doJWD1797Cycle() to advance the “JWD1797” object’s cycling. When a sector byte is encountered and loaded into the controller object’s data register, a data request (DRQ) is signaled by bit 1 of the status register transitioning to a value of 1. If a DRQ signal is detected during one the test loop iterations, the readJWD1797() is called on the controller object’s data port. The byte retrieved from the data register is assigned to the “r_byte” local variable. The local “r_test_byte” variable is then set to the “payload_test_data” array element pointed to by “test_byte_pointer.” The value read from the data register
("r_byte"), is compared to the test array byte ("r_test_byte"). If the two bytes are equal the READ SECTOR command successfully read the correct byte and a confirmation message is printed to the console. Otherwise, if the bytes are not equal, “WRONG BYTE” is printed. In this case, the READ SECTOR command has malfunctioned.

The first READ SECTOR test is interrupted by a FORCED INTERRUPT command at loop iteration number 27,000. This is done to test the effectiveness of the FORCED INTERRUPT command to immediately stop an active command. On the next iteration (27,001) the type II status register value is printed to the terminal to show that the command was successfully interrupted. “WRONG BYTE” messages are then printed because the command is no longer loading new bytes into the data register after being interrupted.

Effects of the FORCED INTERRUPT command test are shown in figure 42. Read bytes compare correctly with test bytes until the FORCED INTERRUPT command is issued. The behavior illustrated in figure 42 demonstrates a successful test of the FORCED INTERRUPT command's ability to immediately stop an active command. Numbers in the left column indicate the sector byte number count. In a 512-byte sector, for example, these numbers run from 1 to 512.
The same READ SECTOR command variation with delay and multi-record enabled is tested for second time but without the FORCED INTERRUPT command being issued mid execution. Figure 43 shows a successful read of sector 7 and sector 8 of formatted track number 7. The left pane of figure 43 shows the beginning and end portions of the sector 7 read, and the right pane shows the same for the sector 8 read. Verification output is also shown for each sector. The sector number read from the address ID field is highlighted with a red box.

Figure 42. The READ SECTOR command being tested in the readSectorTest() function. This terminal output shows a FORCED INTERRUPT command (interrupt immediately) being issued while the READ SECTOR command is active. The command is interrupted and deactivated, thus demonstrating a successful FORCED INTERRUPT command implementation.
A third test of the READ SECTOR command is conducted inside the `readSectorTest()` function. This test ensures the command byte’s “side select” bit is properly registered. Bit 1 of the READ SECTOR command byte indicates whether head 1 or head 2 should be active. A value of 0 in bit 1 activates head 1 (side 1), and a value of 1 activates head 2 (side 2). Formatted track numbers increment on alternating sides of each cylinder. For instance, cylinder 0 hosts formatted track number 0 on side 1 and formatted track 1 on side 2. Continuing inward to cylinder number 1, side 1 holds formatted track 2 and side 2 has formatted track 3. Cylinder 2 hosts formatted tracks 4 and 5, and so on.

When setting up the READ SECTOR command, if the head is over cylinder 3 as in the last test, an active head 1 reads sectors from formatted track number 6 on side 1 of the disk. An active head 2 over cylinder 3 reads sectors
from side 2 and therefore targets formatted track number 7. The previous test sets bit 1 of the command byte to a value of 1; formatted track 7 is read. In the third test, the side-select bit is set to 0, and sectors from formatted track 6 are read.

In addition to testing the side-select bit, “single record” mode is tested in the third test. Single record mode is enabled by setting bit 4 of the command byte to 0. Previous tests enabled “multiple record” mode, and thus sectors were read until the end of the track was reached. With single record mode, the READ SECTOR command reads the sector indicated by the value contained in the sector register and then terminates. The third test ensures that when single record mode is enabled, only the sector indicated by the sector register’s value is read.

Figure 44 shows successful test results of the READ SECTOR command when head 1 is selected and single record mode is activated. A value of 0x03 is written to the sector register and bits 4 and 1 are set to 0 to activate head 1 and enable single record mode, respectively. Since all bytes are confirmed and the command terminates after reading sector 3 of formatted track 6, the intended behavior is observed. The sector number is highlighted by a red box in the address ID field data output, and the active head number is highlighted with a yellow box. Note that value of 0x00 in the head ID field refers to head 1/side 1.
readAddressTest()

Reading the current address on a DOS-formatted 5.25" floppy disk is accomplished by issuing a READ ADDRESS command to the FD-1797 controller. This command collects the next set of address ID bytes encountered by the read/write head. Each byte is read into the controller's data register. A data request signal (DRQ) is sent to the computer for each byte. The address ID field bytes appear before each data field section on a formatted track. Sector payload data is contained within the data field section. Figure 45 shows where the address ID field bytes are located on the formatted track. [25] As the figure
indicates, the collection of six ID bytes appears before the gap 2 and sync byte sections preceding the data address mark and data payload. Information about the head’s current location is conveyed by the six ID bytes. Which cylinder, disk side (head), and sector the head is positioned over is indicated by the first three bytes. Sector length is reported by the fourth byte followed by two cyclic redundancy check bytes.

![Figure 45. Location of the six address ID field bytes within a DOS-formatted track is highlighted by a red box (top). The ID byte section is magnified to clearly show the meaning of each byte.](image)

When a READ ADDRESS command is issued, the controller immediately begins searching for bytes of value 0x00. Once four consecutive 0x00 bytes are detected, the controller must find an address ID mark within 16 bytes or the process is timed out. The ID address mark is labeled “IDAM” in figure 45. As the
figure shows, IDAM consists of three consecutive 0xA1 bytes followed by an 0xFE byte. The “JWD1797” object’s READ ADDRESS command implementation must initiate the IDAM search and timeout process accordingly.

The `readAddressTest()` function verifies that the READ ADDRESS command’s IDAM search is properly conducted by reporting when the 0x00, 0xA1, and 0xFE bytes are detected. A 6-element array is declared as the “JWD1797” object’s “ID_field_data” instance variable. The array holds the six address ID data bytes; its elements are populated as each ID byte is encountered. The `readAddressTest()` function tracks the array elements’ values as the command proceeds.

Two READ ADDRESS command tests are conducted in the `readAddressTest()` function. The first test begins with a SEEK command positioning the head over cylinder 2. After the SEEK command completes, the status of the controller is printed to confirm that the read/write head is over cylinder 2. A READ ADDRESS command is then issued to the “JWD1797” controller with its command byte side-select bit (bit 1) set to 0. Head 1, which reads disk side 1, is activated when the command’s side-select bit is set to 0. Therefore, formatted track 4 is read since head 1 is active over cylinder 2. When two 0x00 bytes are encountered, relevant controller state information begins to output to the terminal. When an ID byte is loaded into the data register, the `readJWD1797()` command is called on the data register port to clear the DRQ signal. Figure 46 (left) shows the last test terminal output for the first READ ADDRESS test. The last line of the output reports the six values of the
“ID_field_data” array. Since the array elements appear as expected, byte counts of the IDAM are accurate, and no errors are reported, the test results reflect a successful READ ADDRESS command execution.

Before the second READ ADDRESS test is conducted, the “JWD1797” controller object is reset. A SEEK command then brings the head to cylinder 6. When the READ ADDRESS command is issued for the second test, its command byte’s side-select bit (bit 1) is set to 1, enabling head 2. With the head over cylinder 6 and head 2 activated, the next address ID field is read from formatted track number 13. Test output for the second READ ADDRESS variation test is handled like output for the first test. Figure 46 (right) shows the final “JWD1797” object state after the READ ADDRESS command is complete. As before, the six elements of the “ID_field_data” array are displayed on the terminal output’s last line. Element 0 (zero) of the array for this test shows the head is over cylinder 6. Verification of head 2’s activation is seen by noting that the second element of “ID_field_data” has a value of 0x01. Since all relevant byte counts are incremented as expected, and the “ID_field_data” array element values are collected successfully without error, the second READ ADDRESS command test is passed.
The final test function called in testMain.c is `readTrackTest()`. This function tests the “JWD1797” object’s READ TRACK command. Mostly used during diagnostic procedures, the READ TRACK command reads every byte of a single formatted track. After being issued, the command begins its operation when an index pulse is detected. Since the index pulse starts when the head is moved over the first byte of a formatted track, the command utilizes the pulse to know when to start reading bytes. Every new byte encountered is read into the “JWD1797” object’s data register. A data request signal (DRQ) is sent to the computer for each byte. Every byte of the formatted track is read, including all
gap, sync, mark, and data bytes. The command is complete and its operation ceases when the next index pulse is detected.

Two READ TRACK command variations are tested in the `readTrackTest()` function. The first variation test sets the command byte’s side-select bit (bit 1) to 0. Like the READ ADDRESS and READ SECTOR commands, setting the side-select bit to 0 actives head 1, which reads side 1 of the disk. A SEEK command is executed to step the head to cylinder 10 before the first test loop. With head 1 selected and positioned over cylinder 10, formatted track 20 is targeted for the first test. For the second test, the command’s side-select bit is set to 1, and a SEEK command is issued to bring the head to cylinder 31. Since head 2 is active over cylinder 31, formatted track 63 is targeted for the second test. As with previous tests, the “JWD1797” controller object is first reset before SEEK commands are issued.

To verify that each byte is read correctly during the READ TRACK command's execution, the formatted disk byte array (“formattedDiskArray”), assembled during “JWD1797” reset procedure, is accessed for byte comparisons. When a byte is read by the command, it is compared to an explicitly accessed byte from the array. While the bytes are equal, the test loop continues to the next iteration. If a byte read is inconsistent with the explicitly accessed array byte, the loop is immediately exited, and the test is aborted. A message reading “WRONG BYTE - READ TRACK FAILED!!” is displayed in the terminal if the test is aborted.

Before each test loop begins, a starting index pointer is calculated based
on which cylinder the head is positioned over and the value of the side-select bit. The index pointer is set to access the array element corresponding to the first byte of the targeted formatted track. Each time a byte is read by the READ TRACK command, a comparison is made, and the array index pointer is incremented.

Figure 47 shows the last five byte reads and comparison results for the two READ TRACK command variation tests. The numbers in the left-most column correspond to the rotational byte pointer’s value for the given byte. The rotational byte pointer takes on values in the range 0 to 5767 because the 320k disk image used for the tests has formatted track lengths of 5768 bytes. The second column of numbers in the terminal output refer to the number of bytes read. This number will always be one greater than the rotational byte pointer because the count starts at a value of 1.

Pertinent state information for the “JWD1797” controller object is displayed below the byte read results. An important value to note in the state information is bit 0 of the status register, which should hold a value of 0 to indicate the command is complete. The track register value should reflect the head’s cylinder position matching the target cylinder. The last byte read from the track should be contained in the data register. In both test cases, all values are correctly reported. Additionally, because both tests reach the last byte of their respective formatted tracks without terminating, the READ TRACK command implementation is considered to be successfully tested.
Figure 47. Terminal test output for the READ TRACK command tests run in the `readTrackTest()` function. The last five byte reads and comparison results are shown for each test. Results displayed in the top pane are from the test targeting formatted track number 10. The bottom pane's output is from the test targeting formatted track 63.
V. Conclusion

Ultimately, the goal of fully booting and running the Z-100’s two operating systems, Z-DOS and CP/M-85, and subsequently running their native software was not attained. However, Z-DOS was partially booted, and the custom FD-1797 disk controller implementation was largely successful. Screen scrolling is also not yet implemented and composite colors are not rendering or clearing properly when using the COLOR command. Otherwise, all BIOS commands function as expected.

The point at which Z-DOS is initialized when using the BOOT command demonstrates that the FD-1797 disk controller implementation is functioning as expected. This observation is bolstered by favorable results obtained when using the disk controller emulation’s custom testing environment. The Z-DOS bootloader is successfully loaded and executed. Furthermore, evidence for the successful loading and execution of several other Z-DOS initialization modules is evident by the screen output shown in figure 48. The figure shows the emulator’s screen after executing the BOOT command. The process hangs after printing the BIOS release information.
Further investigation suggests that Z-DOS's initialization is expecting an unimplemented interrupt where the process hangs. The series of assembly instructions shown below constitutes the loop where the BOOT process hangs. It is thought that the “TEST” instruction is looking to a memory location that is modified when a specific interrupt routine is executed. The “JE” (jump when equal) instruction exits the loop if the “TEST” instruction clears the zero flag. This is confirmed by forcing the zero flag to clear by manipulating the 8088 processor object's zero flag instance variable at the “TEST” instruction. Figure 49 shows the
result of forcing the cleared zero flag. As shown in the figure, the hung up loop contained in the assembly listing above is exited by force-clearing the zero flag. However, another error occurs when trying to load the COMMAND.COM module. Although forcing the zero flag to clear is not a legitimate method to reach this point, it further demonstrates that the FD-1797 disk controller is most likely functioning properly.

Figure 49. Result of forcing the 8088 zero flag to 0 at CS=0x40 : IP=0xED6 of the Z-DOS initialization process. This causes the hung up loop reached when using the BOOT command to exit. However, another error occurs when attempting to load the COMMAND.COM module.
Assembly loop where the BOOT command hangs in figure 48:

CS = 40

IP = ECF   CALL       imm16
IP = ED0   CLI
IP = ED6   TEST r/m8  imm8
IP = EDF   JE         imm8
IP = EE0   STI
IP = EEA   RET
IP = EE7   JAE        imm8

With further investigation building on the results reported here, it is expected that a successful Z-DOS initialization for a Z-100 emulator is possible. It is not unreasonable to expect the issues pertaining to screen scrolling and color rendering to be solved without difficulty. The experience gained through the research and development done for this project suggests that a complete Z-100 emulator can become a reality.
REFERENCES


