Hardware Floppy Disk Emulator

Doug Johnson, Shaun Finlison
dougvj@gmail.com, shaunfin78@gmail.com

May 6, 2013

Contents

1 Introduction

Magnetic data storage media were among the first to offer cheap rewritable and portable mechanisms to store and transfer data from one computer system to another. Although magnetic storage media such as hard drives are still in use to day and are perhaps at the most reliable level that such a medium can obtain, their importance is quickly fading out in favor of solid state devices. This has led to a unique situation where PC floppy disk drives, once ubiquitous, are now rare. There are, nonetheless, situations in which older hardware is still in use and whose replacement may be uneconomical. Among electrical engineers who are familiar with the resources of Utah State University, this situation is apparent when one uses the current logic analyzers which rely principally on unreliable floppy disks for transfer of data. Other situations where this acutely applies is in heavy industry where automated and specialized equipment are powered by hardware that predates the modernity of computer networking and data transfer. In these situations, it is more economical to create hardware that emulates a floppy disk drive than it is to replace the entire system with modern components. Such can be done via the use of modern components to reproduce the behavior of a floppy drive at a very low level.

2 Scope

The scope of this document is to provide the necessary design details of any such hardware floppy disk emulator, and to outline many of the low level details of the functionality that it would have to recreate. A reference implementation on many of the details are given. It was the authors' original intent to create a working reference design, unfortunately this did not come to fruition. The reasons for the failure of the authors' design will be discussed in detail, and possible modifications to the design for success will be discussed at length. This document will not cover any non-relevant specific details of the microcontroller operation chosen, nor will it attempt to discuss at length any non-functional details. The document, however, will provide enough details to completely recreate the authors' design.

3 Design Overview

3.1 Requirements

Below are the requirements of any IBM-PC Compatible floppy disk drive, and of the emulator:

1. The emulator shall provide a mechanism for the user to provide floppy disk data via some other mechanism (IE RS232 or SDIO).
2. The emulator shall emulate the physical components of a floppy drive, including motor on, disk present, and seeking of the disk head via stepper motor, all according to the following figures:

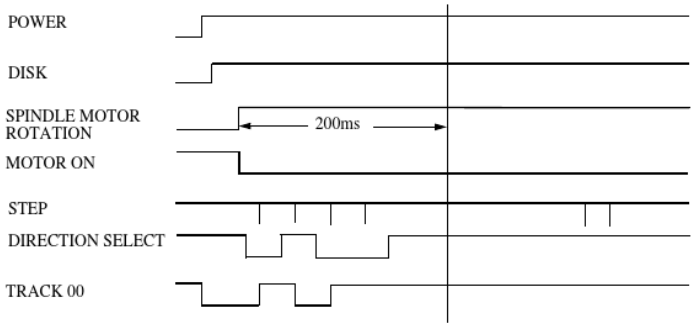


Figure 1: Timing Diagram of Disk Startup

3. The emulator shall keep track internally on which track the head is on and assert the track0 line low if the system is on track 0, according to the following figure:

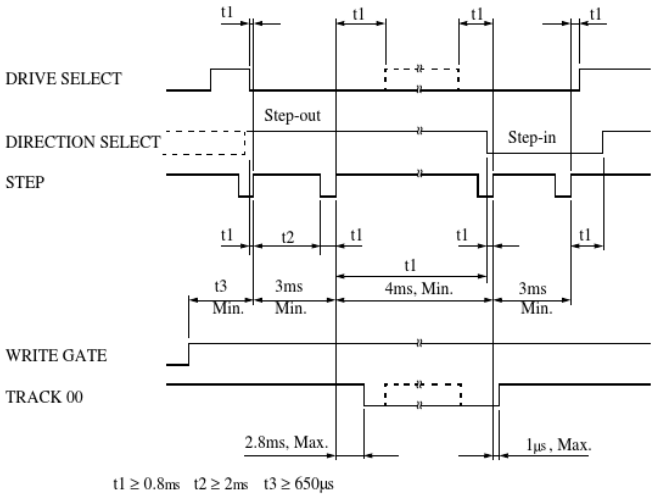


Figure 2: Timing Diagram of Seek Operation

4. The emulator shall output a Frequency Encoded(FM) signal of the contents of a track at 300 RPM, or 500kbps. The timing shall be consistent with the following figure:

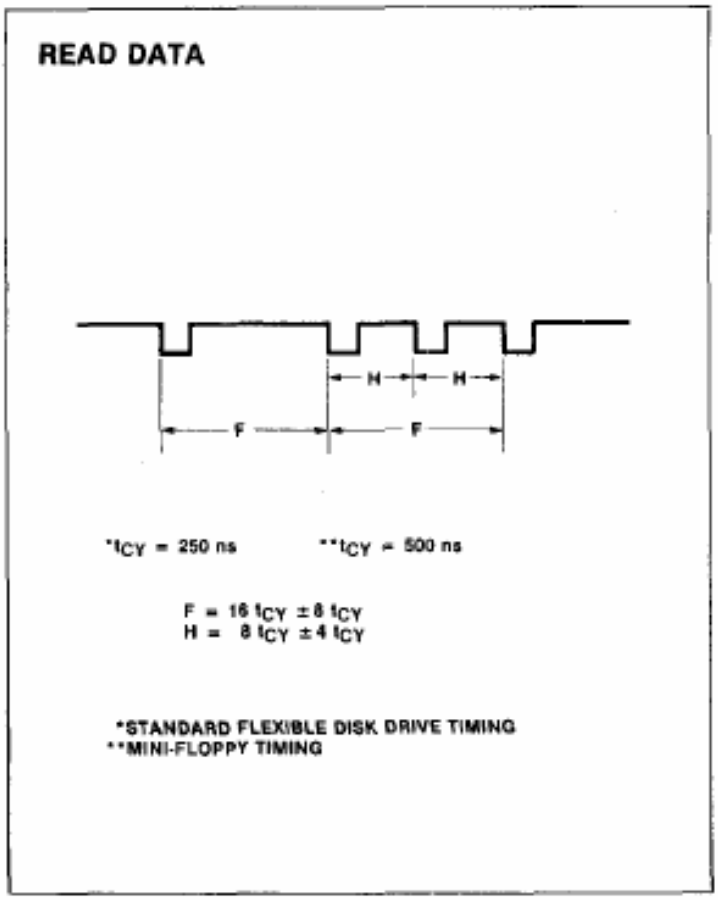


Figure 3: Timing Diagram of Frequency Encoded (FM) Data

5. The emulator shall output a track using the following format:

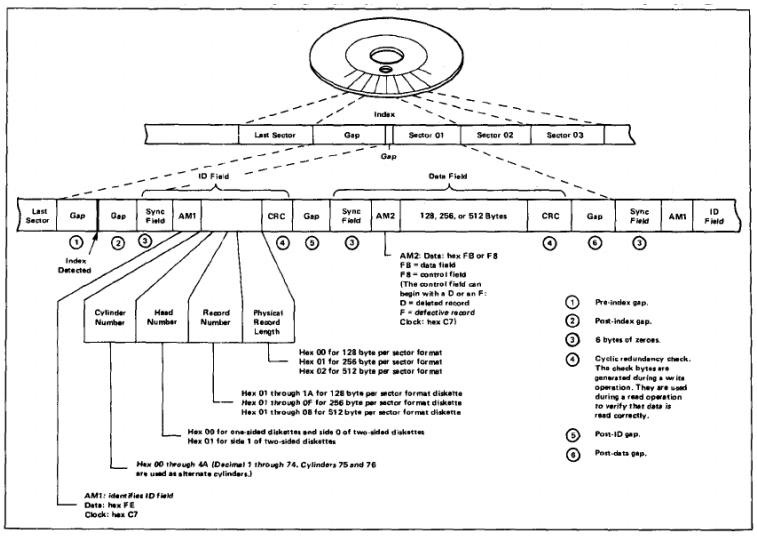


Figure 4: Track Layout and Format

6. The emulator shall send an INDEX pulse at every revolution according to the following figure:

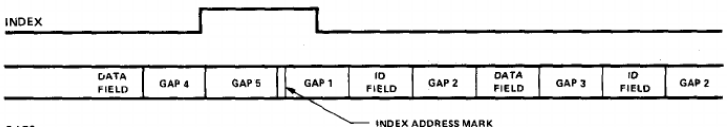
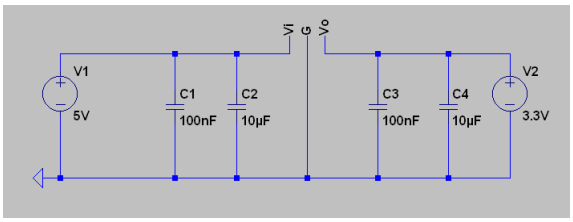


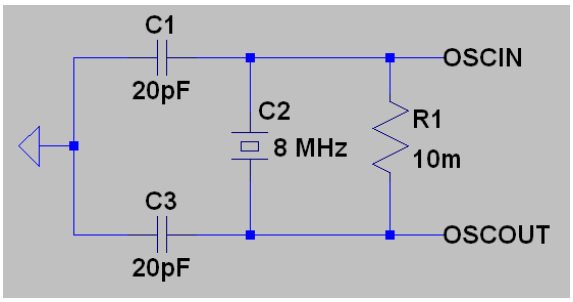
Figure 5: Index Pulse Position

3.2 Dependencies

The design is based on an STM32103ZE Arm Cortex M3 Microcontroller. The microcontroller requires a 5v and 3.3v line. Below is a schematic of the power supply used with capacitors for current load regulation:



Additionally, the microcontroller must be supplied with a 8 MHz crystal for a system clock as below:



This clock will be used in conjunction with an internal Phased-Lock loop to scale the microcontroller's core and peripherals to higher speeds.

An additional trivial dependency is an external computer system to use the emulator with

3.3 Theory of Operation

Upon power-up, the system will assert signals to the PC floppy controller indicating that there is a floppy disk drive present and that it contains a floppy disk. When the PC Floppy controller sends signals engaging the motor and positioning the read/write head, the emulator will read the current track from the storage medium (IE SDIO) and convert it into a format and an encoding that is expected by the computer system. The computer system will be able to seek to a new track at which point the microcontroller will read the data from the SDIO and encode the data. Although there will be a more significant delay in the reading and encoding from the microcontroller than there would be on a real-life system, one can get around that by simply delaying the index pulse until the data is ready.

4 Design Details

4.1 Hardware

4.1.1 Computer-Microcontroller Interface

As part of the development process, a laptop with the following floppy drive ribbon interface was selected:

Pin Nos.	Signals	Pin Nos.	Signals
1	+5V	2	INDEX
3	+5V	4	DRIVE SELECT
5	+5V	6	DISK CHANGE
7	NC	8	READY
9	HD OUT (HD at HIGH level)	10	MOTOR ON
11	NC	12	DIRECTION SELECT
13	1.6MB IN (1.6MB at LOW level)	14	STEP
15	0V	16	WRITE DATA
17	0V	18	WRITE GATE
19	0V	20	TRACK 00
21	NC	22	WRITE PROTECT
23	0V	24	READ DATA
25	0V	26	SIDE ONE SELECT

Figure 6: Floppy Drive Ribbon Interface

As part of the development process, the floppy drive was removed from the laptop, and the module containing the ribbon cable clamp connector was removed and individual wires were soldered onto the connector directly as shown in the following figure:

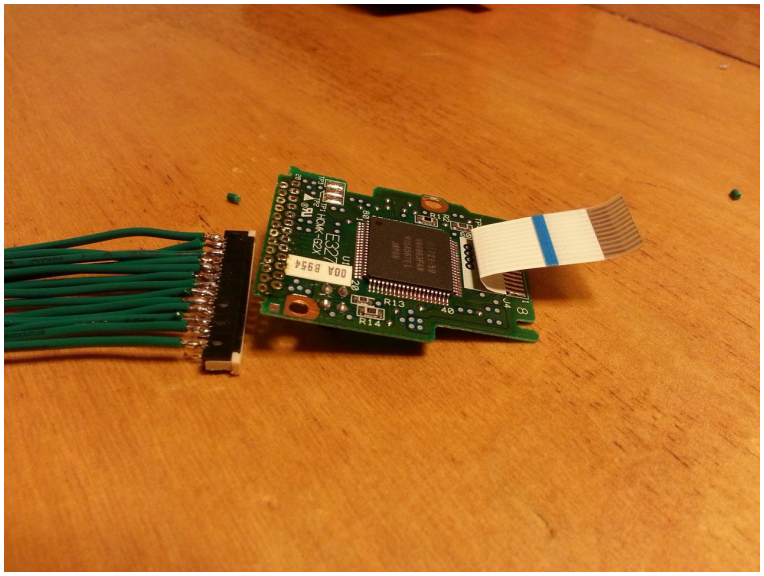


Figure 7: Floppy Drive Module Connector

This method of connecting the micro-controller to the computer appeared to work well, but it must be noted the importance of using a fine tipped soldering iron and flux in order to avoid a short between connections, or from damaging the clamp from over-heating

There is also the small chance that when connecting the larger transmission lines in series with the micro-controller, because of the high clock speed of the controller and the added resistivity of the larger lines, data

could be potentially lost if the lines are made to exceed some length. This however did not appear to be a problem in our design, as the clock speed of the line does not exceed 1MHz.

In order to meet the requirements, a certain subset of pins on the original floppy disk had to be connected to the microcontroller. One notable detail is that the voltage level of the floppy disk interface is 5v, while that of the microcontroller is 3.3v. The microcontroller datasheet indicates that the majority of the GPIO pins on the microcontroller are 5v tolerant. In order to avoid conflicts with any of the SDIO connections, GPIO Port B was selected. Notable on this port is pin13, which is both 5v tolerant and a possible output of a PWM. The following table represents the pin assignments to the microcontroller:

Name	Number	GPIO
INDEX	2	PB2
DISK_CHANGE	6	PB3
DRIVE_SELECT	4	PB4
READY	8	PB6
MOTOR_ON	10	PB7
DIRECTION_SEL	12	PB8
STEP	14	PB9
WRITE_DATA	16	PB10
WRITE_GATE	18	PB11
TRACK00	20	PB12
READ_DATA	24	PB13
WRITE_PROTECT	22	PB14
SIDE1_SEL	26	PB15

Because these pins are active low, the microcontroller is able to simply drive the lines low to assert true. By default, the computer system keeps the lines high; this it does not depend on the microcontroller to assert a voltage high. This would otherwise be a problem because of the voltage differences.

Having labeled the relevant pins and reassembled the floppy connector, the final result was the following connection:

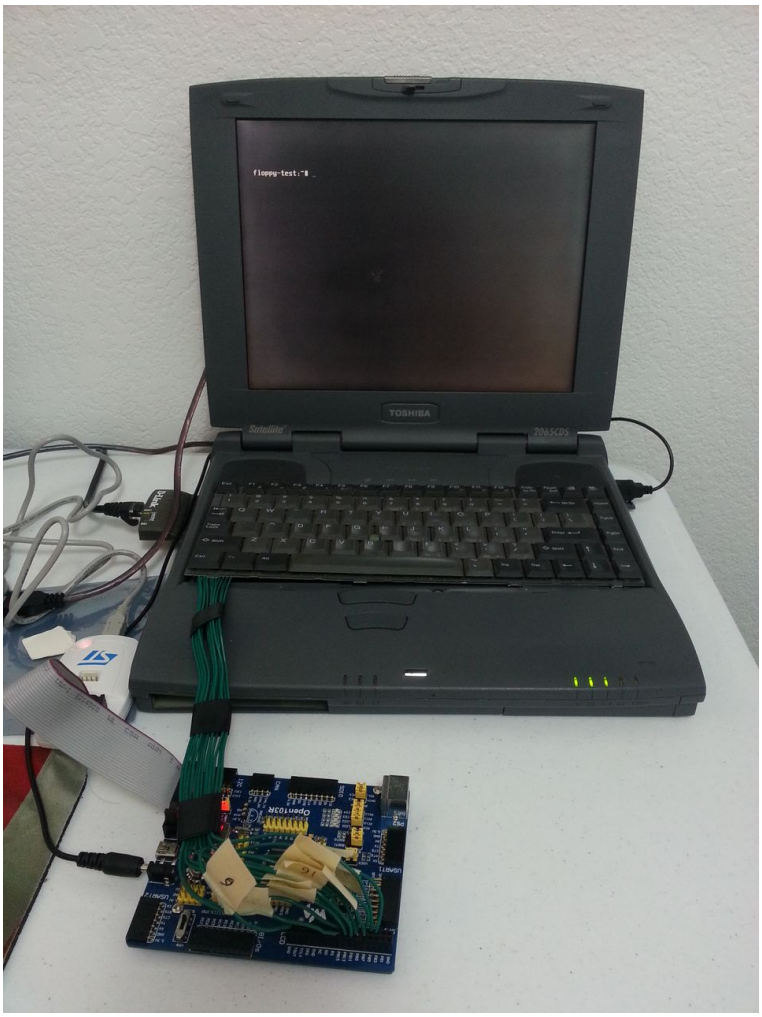


Figure 8: Microcontroller Interface

4.1.2 SDIO Interface

The interface for the SDIO card is given by the following diagram:

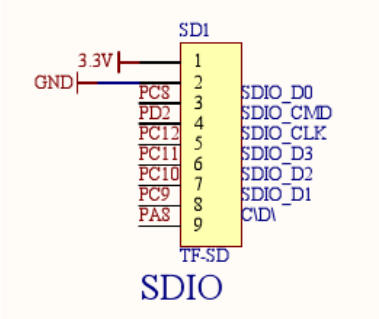


Figure 9: SDIO Interface

4.2 Software

4.3 Program Flow

The software flow is relatively simple, as shown in the following state diagram:

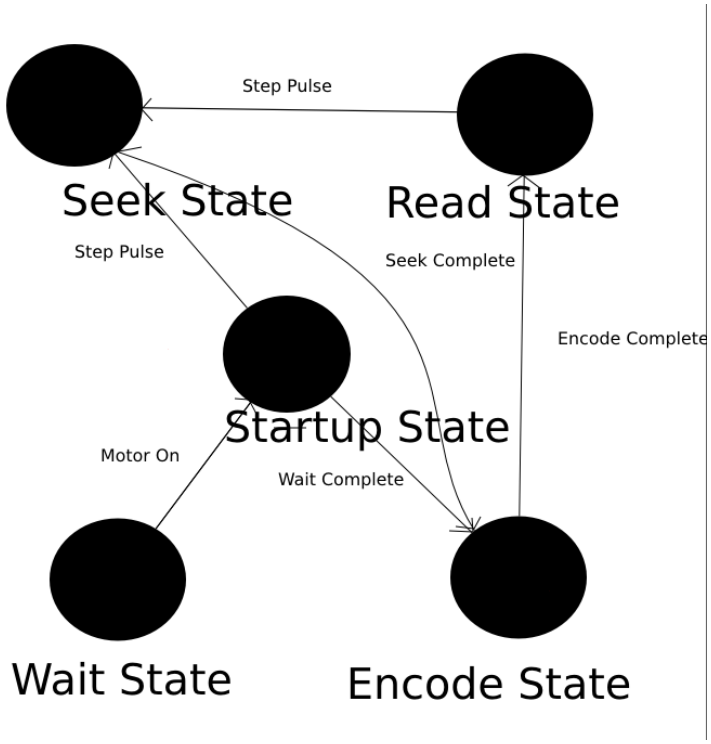


Figure 10: State transition diagram

The wait state is the default state of the software. The software will always be in this state so long as the MOTOR_ON line is false. The startup state is the state that implements the correct timings for asserting the ready lines, it simulates the time it would take to spin up motors and etc. The encode state is the time it takes for the microcontroller to reencode new data for sending across. The read state is the software continually outputting the same track data until there is a step pulse. In the seek state, the software waits a certain number of milliseconds until step pulses have ceased, at which point the software goes back to the encode state.

4.4 Functional Blocks

This section describes the major functional blocks of the software.

4.4.1 Initialization Routine

This routine is called on reset and performs all initialization of the hardware components of the microcontroller system. It initializes the GPIO port B to the relevant inputs and outputs as documented above, including setting the PWM output via selection of the alternate functionality mode for pin 13. It initializes Timer 1 for PWM generation and sets the output to be Pin 13. It determines the auto reload register dynamically via preprocessor macros in C. In addition, the routine initializes the clock counter and timer,

enabling interrupts. The routine also configures and maps the external interrupt line from Pin 9 of GPIO B for the stepper motor pulses. Finally the routine enables the relevant interrupt lines in the NVIC, and sets the priorities where the external interrupt and clock timer have the timer handler has the highest priority, followed by the PWM interrupt then the external interrupt. In addition to all this, the clock was enabled for all of these peripherals. The external oscillator was enabled and set as input to the Phase-Locked Loop. Unfortunately, the microcontroller could only be driven to 48 MHz using this method (see *Design Failure* section).

4.4.2 Encoding Routine

This is a set of several routines. Their job is to decide which data must be fetched from the SDIO, have it fetched, and then format the data with the proper floppy disk format data and encode and buffer the data in the exact encoding expected by the floppy controller. This data will represent the track. The PWM routine, below, will continually cycle through this data in order to output it as though it were data on a physical medium whose disk head travels over the track .

4.4.3 Sector Fetching Routine

This routine, when given a valid sector number, is to fetch that sector (512-bytes) from the SDIO card, store it into a buffer, and return that buffer. The buffer is not persistent, ie, the data can be considered processed by the next call to this routine.

NOTE: Due to the inordinate amount of time spent trying to understand the SDIO hardware, and the fact that the authors were running out of time on the entire project, the authors simply ended up hardcoding a bootsector into the code and had this routine return that sector all the time. Such a decision does not effect the design of the rest of the system. See appendix B

4.4.4 PWM Reload Handler

This routine determines the next bit to output on the PWM and programs the PWM compare register accordingly. Because of the necessity of this code being efficient, the following is the clearest implementation I know possible in C.

```
1 //This routine is called on every update to the PWM timer
2 void TIM1_UP_IRQHandler() {
3     uint8_t local_bit = 15 - current_output_bit % 16; //Grab the current local bit
4     uint16_t local_byte = (current_output_bit /16); //Grab the current local byte
5     TIM1->CCR1 = ((output_track[local_byte] >> local_bit) & 1) ? PULSE_WIDTH : 0;//Set the
          PWM output for the next cycle
6     current_output_bit = (current_output_bit + 1) % (OUTPUT_TRACK_SIZE * 16); //Go to next
          output and stay within
7     NVIC_ClearPendingIRQ(25); //Clear Pending
8     TIM1->SR &= ~0x1; //Clear interrupt flag
9 }
```

Clock Timer Handler

This routine is fired every millisecond and updates a counter called "current_time." This counter is used by the state changer to implement the proper delays for motor spin-up time, etc. Because this counter wraps around every 2^{32} milliseconds, or about 50 days, there is a nearly infinitesimally small chance that a delay will overlap this boundary and undefined states may occur. Rather than check for this condition, the authors concluded that probabilistically, it is not worth the performance hit.

Statistics Generator

This routine was created for debugging purposes and is explained in more detail in the *Design Failure* section.

Step Pulse Interrupt Handler

This routine simply increments or decrements the current track pointer in memory depending on the state of the DIRECTION_SEL line. The state change routine then handles any necessary

State Change Routine

This routine continually watches for changes in the DRIVE_SELECT and MOTOR_ON line. It is responsible for rebuilding the current track and for asserting the READY and TRACK0 and INDEX lines, as well as WRITE_PROTECT (asserted true since writing is not yet part of this design) DISK_CHANGE, and other lines. When the system has stabilized onto a track, or in other words, when it is clear that the system is no longer seeking, it calls the encoding routine to generate the current track for output.

5 Testing and Verification

Testing and verification was done via various tools. The first tool is the Linux Kernel and related software for accessing the floppy disk drives at the lowest level possible. Kernel debugging output was enabled in order to see the detailed state of the floppy disk driver, as shown below in the following output:

In addition, a logic analyzer was used to verify the correctness of the different signals as shown in the figure below:

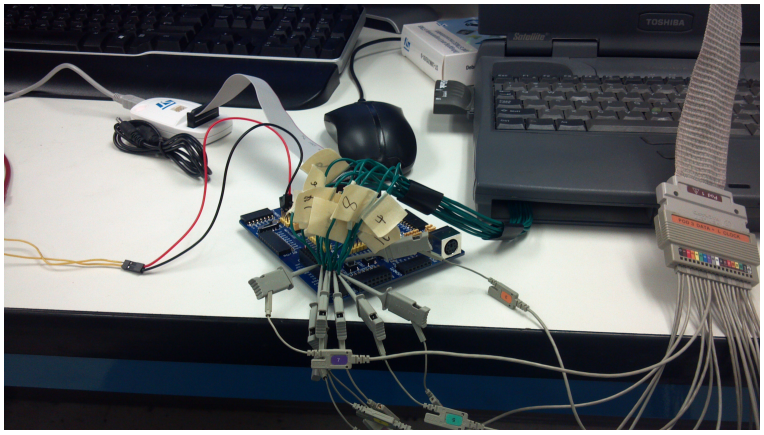


Figure 11: Interface Connection to Logic Analyzer

Additionally, a STM32 Debugger was used for verification of various software behavior and hardware configurations.

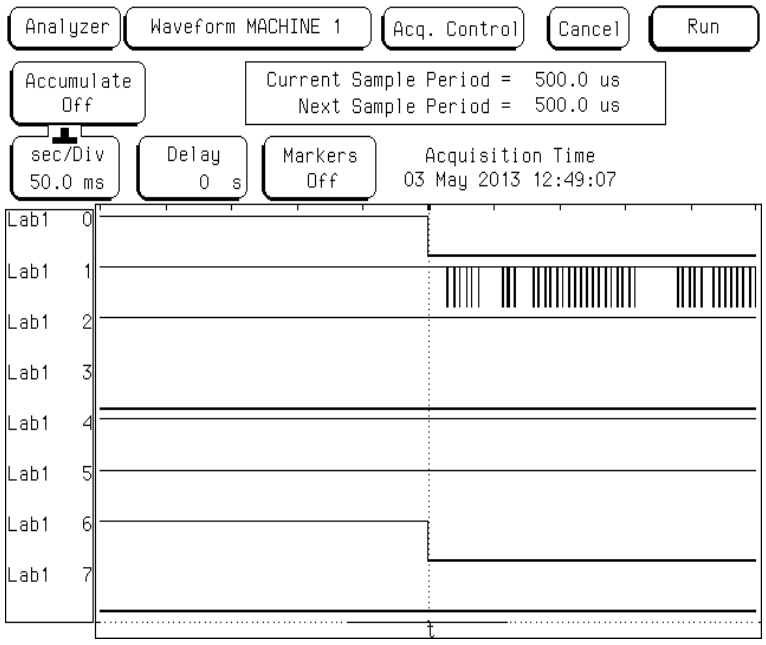
5.1 Hardware

Hardware was verified using one of two ways. The first is via the debugger and Kiel(TM) μVision development environment. Verification that the lines were correctly detected on input was seen in the simulation. When the PC was commanded to read from the floppy disk, it was obvious that the MOTOR_ON line was correctly

read in the microcontroller. Other lines were verified similarly. Additionally, a continuity tester was used to verify correct connections through the ribbon cable in the custom built interface to the microcontroller

5.2 Startup

Below is a logic analyzer output of the motor on sequence:



Line 1 is MOTOR_ON, Line 6 is DISK_SELECTION, and Line 1 is Read Data

Figure 12: Analyzer output of

This verifies some of requirement 2.

5.3 Seeking

Seeking was tested by commanding, from the PC, a seek to a certain track location. Both tracks forward and backward from the current track was tested. The internal microcontroller value indicating the current track was watched via the debugger while a command was issued from the PC to seek to that track. This worked clearly and perfectly as shown below:

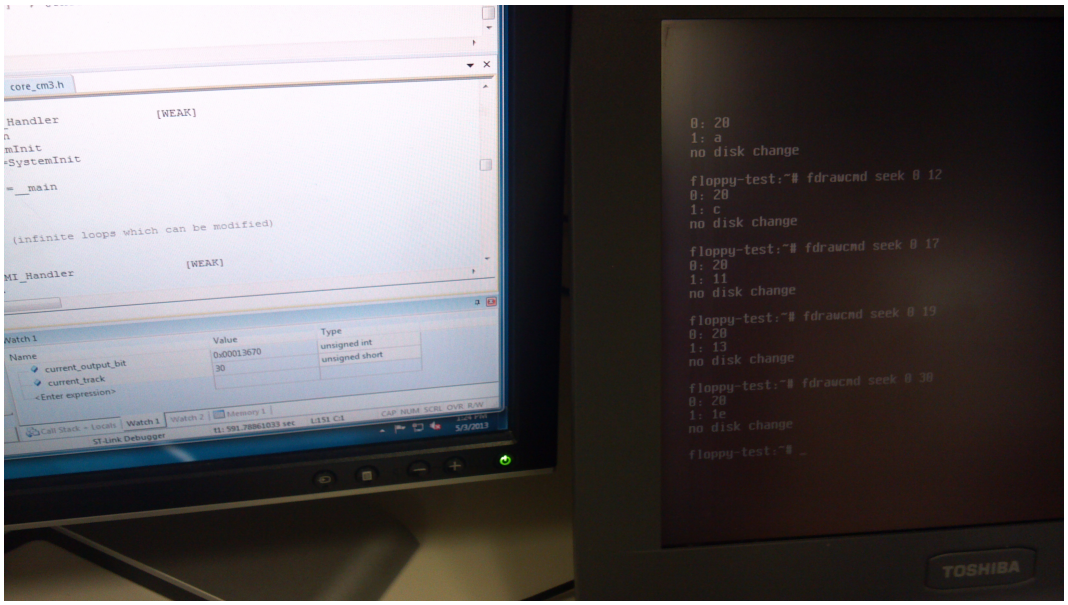
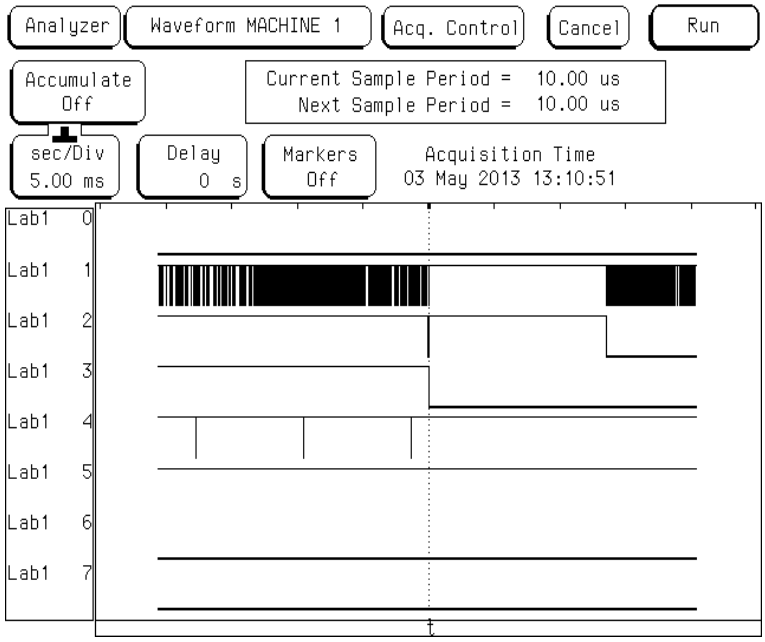


Figure 13: Image of tracking synchronized in both the microcontroller and PC

Additionally, the microcontroller successfully outputs true when the computer seeks to track 0, as in this logic analyzer output below:



Line 4 is STEP pulses,Line 3 is TRACK0, Line 2 is READY, and Line 1 is READ_DATA

Figure 14: Analyzer output of track 0 assertion

This verifies requirements 3 and some of 2.

5.4 Data Output

As documented in the next section, we were unable to generate the correct output signal at the correct timing, however the correct signal is generated at half the correct timing:

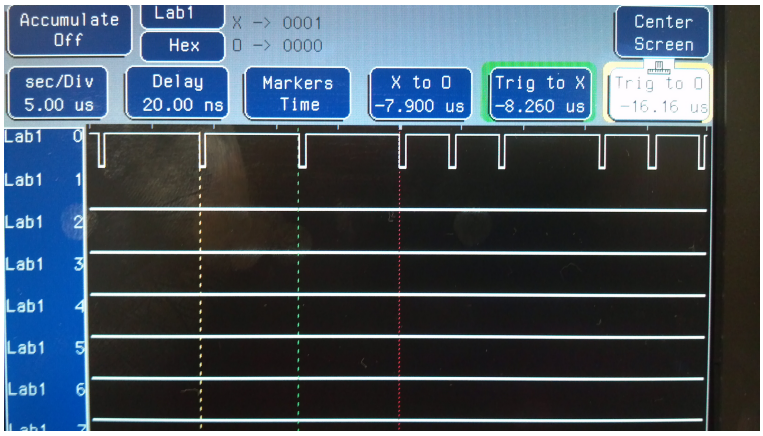


Figure 15: Analyzer output of 000101

NOTE: Floppy disk for Logic Analyzer failed, therefore a photo was taken rather than an image dump

Note the period length is 8 microseconds instead of the prescribed 4 microseconds.

This is a dump of a random spot in the middle of the disk image. Due to constraints on the logic analyzer, it is impossible to collect more data than this waveform without significantly loosing resolution. However, that the beginning of the track data were all 0s, as described by the datasheet's definition of gap, was verified

Additional data encoding verifications were done using the debugger. The index mark was found, which has a unique bit pattern with missing clock encoding, and the data was correct in the waveform output of the debugger. Additionally, each discrete part of the data format was tested at each level. This verifies requirements 4 and 5.

5.5 Index Pulse

The index pulse was verified via both the logic analyzer and the debugger. Because the pulse width is determined by an absolute bit position value, verification in the debugger was simple and determined to be correct. Below is the relevant section of code that implements the pulse boundary:

```
1   if (index == TRUE &&
2                   current_output_bit > index_pulse_end &&
3                   current_output_bit < index_pulse_start) {
4       set_false(INDEX);
5         index = FALSE;
6     } else if(index == FALSE &&
7                   current_output_bit > index_pulse_start) {
8       set_true(INDEX);
9         index = TRUE;
10    }
```

This routine is checked at least every millisecond due to the fact that all spare microcontroller cycles are used to check state changes. This code is found in the state changing routine.

The variables for the index pulse start and index pulse end are set in the track initialization, and are determined to by the datasheet's requirements that the pulse span the index address mark and that the pulse be within the gaps.

This verifies requirement 6.

5.6 Data Input

While the original design called for SDIO, due to testing constraints, an interim solution was created in which arbitrary data could be inputted to the source code. See appendix B for details. Though the method is unorthodox, it verifies requirement 1.

6 Design Failure

6.1 Frequency Scaling Issue

Attempts to drive the microcontroller faster than 48MHz met with either glitches or complete failure on the part of the microcontroller. At 56MHz, the microcontroller enters into a hard-fault at a seemingly random point as shown:

```
163  HardFault_Handler\
164                  PROC
165                  EXPORT  HardFault_Handler
166                  B       .
167                  ENDP
```

Figure 16: Hard Fault encountered at 56MHz

At one point the microcontroller halted inside the NMI handler, whose only explanation is glitching due to too high of frequencies. In spite of the microcontroller datasheet's indicating a maximum core speed of 72MHz, this was never attainable. At such speeds, the debugger would fail right at the moment of switching to the PLL frequency, as demonstrated:

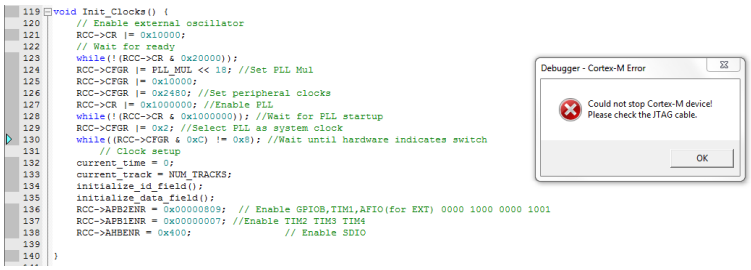


Figure 17: Debugging Failure at frequencies above 56MHz

The authors used a set of preprocessor macros, defined below, to modify the clock and timing configuration for various purposes. These were created to facilitate testing different frequencies and timings.

```
#define TARGET_RPM 300
#define BITS_PER_TRACK 100000
#define TARGET_BPS ((TARGET_RPM*BITS_PER_TRACK)/60)
#define CLOCK 48000000
#define TIMER_CLOCK (CLOCK/2)
#define PULSE_PERIOD (TIMER_CLOCK/TARGET_BPS)
#define PULSE_WIDTH (PULSE_PERIOD/8)
#define PLL_MUL ((CLOCK/8000000)-2)
```

Figure 18: Preprocessor definitions for timing and frequencies

6.2 Target Bitrate Failure

Unfortunately, the authors were unable to drive the microcontroller to speeds fast enough to update by the next PWM period. At 48MHz, various tests were undertaken to determine the maximum bitrate of the microcontroller using this method. The following figures illustrate various tests:

bps	99590	unsigned int
target_bps	100000	unsigned int
bps_percent_error	0	int

Figure 19: Transfer Rate at 100kbps target

bps	149070	unsigned int
target_bps	150000	unsigned int
bps_percent_error	0	int

Figure 20: Transfer Rate at 150kbps target

bps	246430	unsigned int
target_bps	250000	unsigned int
bps_percent_error	1	int

Figure 21: Transfer Rate at 250kbps target

bps	244900	unsigned int
target_bps	500000	unsigned int
bps_percent_error	51	int

Figure 22: Transfer Rate at 500kbps target

bps	239380	unsigned int
target_bps	1000000	unsigned int
bps_percent_error	76	int

Figure 23: Transfer Rate at 1000kbps target

At a target of 500kbps, the rate defined in the datasheets for the floppy disk drive, the interrupt handler is unable complete by the next PWM period. This has the interesting effect of duplicating every attempted bit transmission, and the interrupts are only serviced on half of the PWM update periods. In every case, the testing computer would indicate a floppy disk timeout with a debug dump.

7 Possible Design Solutions

One possible method of allowing the microcontroller to handle the faster data transfer rates while staying at a lower frequency is by buffering the output using a shift register. The clock to the shift register would

be given via a PWM. The clock would be generated such that the pulse of a single bit would be two clock cycles, one on the start of a pulse and one on the end. The PWM would then wait for the correct amount of microseconds until the next pulse occurs. Below is a diagram of the concept:

7.1 Shift Register

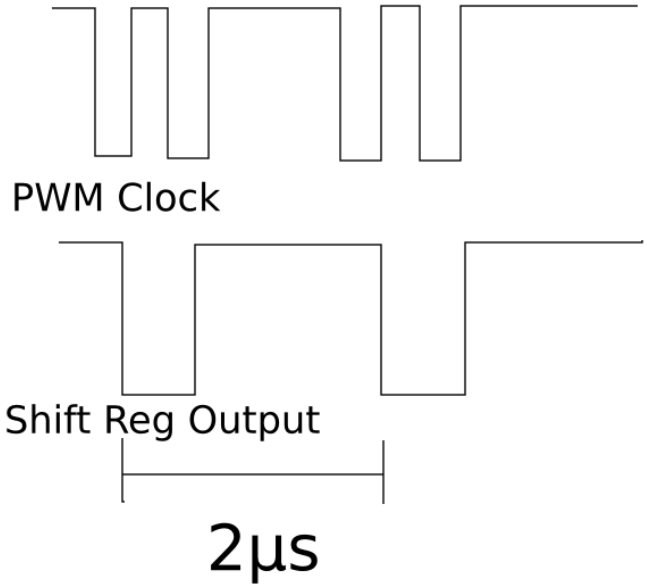


Figure 24: The negative output of a shift register with the value 01010, which changes on rising clock.

The microcontroller could load up to 8-bits at a time into a 16-bit shift register. The reason for the extra bits would be to pad the pulses with 0s. Alternatively, the shift register could be loaded without padding and an alternate PWM output could drive the line low at the correct time in order to generate the pulse output.

7.2 CPLD or FPGA

In addition to a shift register, a CPLD or FPGA could be used to implement the correct custom timing and perhaps even the encoding. This would allow the microcontroller to focus solely on data retrieval and formatting and not on actual signal generation. Indeed this would be a major issue if the microcontroller were to scale to more complicated data encoding methods such as the Modified Frequency Modulation (MFM) method, which requires even tighter timings.

8 Conclusion

This document describes the requirements, design, and failure of a reference implementation of the design of a hardware floppy disk emulator. Although the design failed and the project was not successfully completed, it is clear *ex post facto* that the project was too involved and complicated to complete in the timeframe and guidelines given in the class. Regardless, the project is invaluable in terms of learning and application of microcontroller hardware and software concepts.

9 Appendix A: Personal note from an Author (Doug Johnson)

This note is primarily directed to the graders. (Dr. Gerdes, et al). I wish to address the pedagogical perspective of this project. Although we failed to successfully implement this project, I feel that I have learned substantially not only about microcontrollers but also about the low level data encoding methods and signals used in magnetic storage media. I would have really liked to have had this project working as planned. Had I known that the project was as low level as it was, I probably would not have considered it as a final project and would have chosen something more dependent on software rather than hardware knowledge for a successful completion. Nonetheless, I feel that, as expressed in the document, the design can be extended to be successful, and is a project I may personally take up over the coming summer months.

I would like to also stress the work that was placed even to produce what was created here. Many consecutive post-3AM nights, and even one all-nighter. Countless hours were put into reading datasheets of the floppy disk and floppy controllers for details that I may have overlooked, adjusting the implementation, consulting the floppy disk module source code of Linux, etc. A particularly difficult aspect of this project is the lack of good debugging tools. PC Floppy drive controllers do not offer low enough level output to determine the point of failure inside the controller itself, so you must rely on the datasheets and the debugging tools on the microcontroller side. Had I chosen a less challenging project that did not require such vast consultation of datasheets outside of the microcontroller hardware, I know we would have been successful in its completion.

The design does make successful use of several advanced features of the microcontroller and demonstrates, despite its failure, excellent knowledge of embedded systems programming. It is my hope that the graders also, at least partially, share this perspective.

10 Appendix B: Bootloader Information

The "Hello World" bootloader for testing was found from a tutorial by Viral Patel at the web-address http://viralpatel.net/taj/tutorial/hello_world_bootloader.php. The authors' original intention was to implement such a bootloader from scratch. Time constraints precluded this.

```
1  [BITS 16]    ;Tells the assembler that its a 16 bit code
2  [ORG 0x7C00]     ;Origin, tell the assembler that where the code will
3      ;be in memory after it is been loaded
4
5  MOV AL, 65
6  CALL PrintCharacter
7  JMP $        ;Infinite loop, hang it here.
8
9
10 PrintCharacter: ;Procedure to print character on screen
11     ;Assume that ASCII value is in register AL
12 MOV AH, 0x0E     ;Tell BIOS that we need to print one charater on screen.
13 MOV BH, 0x00     ;Page no.
14 MOV BL, 0x07     ;Text attribute 0x07 is lightgrey font on black background
15
16 INT 0x10     ;Call video interrupt
17 RET      ;Return to calling procedure
18
19 TIMES 510 - ($ - $$) db 0    ;Fill the rest of sector with 0
20 DW 0xAA55            ;Add boot signature at the end of bootloader
```

Additionally, the following code was 'kludged' together quickly for conversion of arbitrary data into a valid C byte table:

```
1  #include <stdlib.h>
2  #include <stdio.h>
3
4
5
6  int main(int argc, char** argv) {
7      int c;
8      int current_column;
9      int newline_offset;
10     int line_size;
11     int i;
12     if (argc > 1) {
13         current_column = printf("const char %s[] = {", argv[1]);
14     } else {
15         current_column = printf("const char table[] = {");
16     }
17     newline_offset = current_column;
18     if (argc > 2) {
19         line_size = atoi(argv[2]);
20     } else {
21         line_size = 80;
22     }
23     c = getchar();
24     do {
25         if (current_column != newline_offset)
26             current_column += printf(", ");
27         if (current_column > 74) {
28             putchar('\n');
29             for (i = 0; i < newline_offset; i++) {
30                 putchar(' ');
31             }
32             current_column = newline_offset;
33         }
34         current_column += printf("0x%02X", c);
35         c = getchar();
36     } while(c != EOF);
```

```
37      printf("};\n\n");
38  }
```

The code was designed with UNIX in mind. Using NASM and GCC, the valid table is produced via the following series of commands:

```
1  gcc ctable.c -o ctable
2  nasm bootloader.asm -o bootloader.bin
3  cat bootloader.bin | ./ctable BOOT_SECT
```

11 Appendix C: Reference Software Implementation

```
1  //  ECE 3710 Microcontroller project
2  //  Floppy disc emulator
3  //  Utah State University
4  //  Written by Doug Johnson
5  //  and Shaun Finlinson
6  //
7
8  #include "stm32f10x.h"
9  #include "core_cm3.h"
10
11 //This is simply a boot sector we have test the code without SDIO
12 //It contains a copy of a simple "Hello World!" bootsector for x86
13 const char BOOT_SECT[] = {0xB0, 0x41, 0xE8, 0x02, 0x00, 0xEB, 0xFE, 0xB4, 0x0E,
14                           0xB7, 0x00, 0xB3, 0x07, 0xCD, 0x10, 0xC3, 0x00, 0x00,
15                           0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
16                           0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
17                           0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
18                           0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
19                           0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
20                           0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
21                           0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
22                           0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
23                           0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
24                           0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
25                           0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
26                           0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
27                           0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
28                           0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
29                           0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
30                           0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
31                           0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
32                           0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
33                           0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
34                           0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
35                           0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
36                           0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
37                           0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
38                           0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
39                           0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
40                           0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
41                           0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
42                           0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
43                           0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
44                           0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
45                           0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
46                           0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
47                           0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
48                           0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
49                           0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
50                           0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
51                           0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
52                           0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
53                           0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
54                           0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
55                           0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
56                           0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
57                           0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
58                           0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
59                           0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
60                           0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
61                           0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
62                           0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
63                           0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
64                           0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
65                           0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
```

```
                                0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
                                0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
                                0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
                                0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x55, 0xAA};


#define INDEX           0x4         // Output   PB2                         Pin 2
#define DISK_CHANGE     0x8         // Output   PB3                         Pin 6
#define DRIVE_SELECT    0x10            // Input    PB4                         Pin 4
#define READY           0x40          // Output   PB6                       Pin 8
#define MOTOR_ON        0x80          // Input    PB7                       Pin 10
#define DIRECTION_SEL   0x100           // Input    PB8                       Pin 12
#define STEP            0x200             // Input    PB9                         Pin 14
#define WRITE_DATA      0x400           // Input    PB10                      Pin 16
#define WRITE_GATE      0x800       // Input        PB11                      Pin 18
#define TRACK00         0x1000        // Output   PB12                      Pin 20
#define READ_DATA       0x2000              // AFIO PWM    PB13                        Pin
    24
#define WRITE_PROTECT   0x4000          // Output   PB14                      Pin 22
#define SIDE1_SEL       0x8000          // Input    PB15                      Pin 26


#define TRUE 0x1
#define FALSE 0x0

#define TARGET_RPM 150
#define BITS_PER_TRACK 100000
#define TARGET_BPS ((TARGET_RPM*BITS_PER_TRACK)/60)
#define CLOCK 48000000
#define TIMER_CLOCK (CLOCK/2)
#define PULSE_PERIOD (TIMER_CLOCK/TARGET_BPS)
#define PULSE_WIDTH (PULSE_PERIOD/8)
#define PLL_MUL ((CLOCK/8000000)-2)


#define NUM_TRACKS 80
#define SECTORS_PER_TRACK 9
#define INDEX_PULSE_SIZE 35 //Size (in bytes) of the index pulse


volatile uint32_t current_time = 0;
volatile uint16_t current_track = NUM_TRACKS;
volatile uint16_t current_sector = 0;
volatile uint8_t output_enabled = 0;


//Possible entries of the length in the ID field
#define LENGTH_128 0x00
#define LENGTH_256 0x01
#define LENGTH_512 0x02


//These are the different address mark informations we need
#define AM_OFFSET 0x6
#define ID_AM_CLOCK 0xC7
#define ID_AM_DATA 0xFE
#define DATA_AM_CLOCK 0xC7
#define DATA_AM_DATA 0xF8
#define INDEX_AM_CLOCK 0xD7
#define INDEX_AM_DATA 0xFC

volatile struct ID_FIELD {
    char address_mark;
    char cylinder;
    char head;
    char record;
    char length;
```

```
133     short CRC_d;
134 } id_field;
135
136 volatile struct DATA_FIELD {
137     char address_mark;
138     char data[512];
139     short CRC_d;
140 } data_field;
141
142 short find_crc(uint8_t* data, uint16_t size) {
143   //Not Implemented. Return 0xFF which tells the controller we dont have a CRC
144   return 0xFFFF;
145 }
146
147 void initialize_id_field() {
148
149   id_field.address_mark = ID_AM_DATA;
150   id_field.head = 0x01;
151   id_field.length = LENGTH_512;
152 }
153
154 void initialize_data_field() {
155   data_field.address_mark = DATA_AM_DATA;
156 }
157
158 uint8_t* generate_id_field(uint8_t record, uint8_t track) {
159   id_field.record = record;
160   id_field.cylinder = track;
161   id_field.CRC_d = find_crc((uint8_t*)&id_field, sizeof(struct ID_FIELD)-2);
162   return (uint8_t*)(&id_field);
163 }
164
165 uint8_t* generate_data_field(uint8_t* data) {
166   int i;
167   uint32_t* data_source = (uint32_t*)data;
168   uint32_t* data_dest = (uint32_t*)data_field.data;
169   for(i = 0; i < 128; i++) {
170     data_dest[i] = data_source[i];
171   }
172   data_field.CRC_d = find_crc((uint8_t*)&data_field, sizeof(struct DATA_FIELD)-2);
173   return (uint8_t*)(&data_field);
174 }
175
176 void Init_Clocks() {
177   // Enable external oscillator
178   RCC->CR |= 0x10000;
179   // Wait for ready
180   while(!(RCC->CR & 0x20000));
181   RCC->CFGR |= PLL_MUL << 18; //Set PLL Mul
182   RCC->CFGR |= 0x10000;
183   RCC->CFGR |= 0x2480; //Set peripheral clocks
184   RCC->CR |= 0x1000000; //Enable PLL
185   while(!(RCC->CR & 0x1000000)); //Wait for PLL startup
186   RCC->CFGR |= 0x2; //Select PLL as system clock
187     while((RCC->CFGR & 0xC) != 0x8); //Wait until hardware indicates switch
188     // Clock setup
189   current_time = 0;
190   current_track = NUM_TRACKS;
191   initialize_id_field();
192   initialize_data_field();
193   RCC->APB2ENR = 0x00000809;  // Enable GPIOB, TIM1, AFIO(for EXT) 0000 1000 0000 1001
194   RCC->APB1ENR = 0x00000007; //Enable TIM2 TIM3 TIM4
195   RCC->AHBENR = 0x400;         // Enable SDIO
196
197 }
198
199 void SystemInit(void)
200 {
```

```
201   Init_Clocks();
202
203
204   // GPIO setup
205   GPIOB->CRL = 0x43043300;
206   GPIOB->CRH = 0x43F34444; //PB13 is set to PWM output. CH1N of TIM1
207
208   // SDIO setup
209
210   // Timer setup    - 500Kb/s
211   //Set up PA/0 to alternate func
212   GPIOA->CRL &= ~0xF;
213   GPIOA->CRL |= 0xA;
214
215
216   TIM1->CR1 = 0x80; //Set autoreload
217   //TIM1->ARR = (TIMER_CLOCK / 500000); //Set to 500k per second (two times per bit, clock
          and data).
218   TIM1->ARR = PULSE_PERIOD;
219   TIM1->CCMR1 = 0x68; //Enable PWM mode on channel 1
220
221   TIM1->CCER = 0x05;//Enable chanel one outputs
222   TIM1->BDTR = 0xC000; //Enable breaking output
223   TIM1->CCR1 = 0;//Disable output
224   TIM1->EGR = 0x1; //Enable update event generation
225   TIM1->CR1 |= 0x1; //Set to enable
226   TIM1->DIER = 0x0; //DOnt enable interrupt yet.
227     output_enabled = 0;
228
229
230   TIM4->CR1 = 0x90; //Set to dec, autoreload
231   TIM4->ARR = TIMER_CLOCK / 1000; //Set to 1ms
232   TIM4->CR1 |= 0x1; //Set to enable
233   current_time = 0;
234   TIM4->DIER = 0x41; //Start counting
235
236   //Configure EXT Line
237   EXTI->FTSR = 0x00000000; //No falling trigger
238     EXTI->RTSR = 0x00000200; // rising trigger
239     AFIO->EXTICR[2] = 0x00000010; //Map EXT9 to PB
240     EXTI->IMR = 0x00000200; //Enable IRQ
241
242   NVIC_EnableIRQ(23); //EXT LIne
243   NVIC_EnableIRQ(25); //TIM1
244   NVIC_EnableIRQ(30); //TIM4
245   NVIC_SetPriority(23, 1);
246   NVIC_SetPriority(25, 2);
247   NVIC_SetPriority(30, 0);
248   GPIOB->ODR = 0xFFFFFFFF; //Set everything to false (active low)
249 }
250 void delay(uint32_t ms) {
251   uint32_t t1 = current_time;
252   while (current_time - t1 < ms);
253 }
254
255 void set_true(uint16_t pin) {
256   GPIOB->BRR = pin;
257 }
258
259 void set_false(uint16_t pin) {
260   GPIOB->BSRR = pin;
261 }
262
263 uint16_t get_pin(uint16_t pin) {
264   return !(GPIOB->IDR & pin);
265 }
266
267
```

```
268 uint16_t generate_mfm_encoding(uint8_t byte) {
269   //Not Implemented
270   return 0;
271 }
272
273 void enable_output() {
274   TIM1->DIER = 0x91;
275     output_enabled = 1;
276 }
277
278 void disable_output() {
279   TIM1->DIER = 0;
280     TIM1->CCR1 = 0; //Disable PWM
281     output_enabled = 0;
282 }
283
284 //There 6250 unformatted bytes per track (100000/16)
285 #define OUTPUT_TRACK_SIZE 6250
286 uint16_t output_track[OUTPUT_TRACK_SIZE];
287 uint16_t output_track_ptr;
288 uint16_t queue_size = 0;
289 uint32_t current_output_bit;
290 uint32_t index_pulse_start;
291 uint32_t index_pulse_end;
292
293
294 uint16_t calc_fm_encoding(uint8_t data, uint8_t clock) {
295   int i;
296   uint16_t toret;
297   if (data == 0xFF && clock == 0xFF)
298     return 0xFFFF;
299   if (data == 0x00 && clock == 0xFF)
300     return 0xAAAA;
301   toret = 0;
302   for (i = 0; i < 8; i++) {
303     toret |= (((clock>>i) & 0x1)<<(i*2 + 1)) | (((data>>i)& 0x1)<<(i*2));
304   }
305   return toret;
306 }
307
308 void reset_queue() {
309   queue_size = 0;
310   current_output_bit = 0;
311 }
312
313 void set_index_pulse_boundry(uint32_t start, uint32_t end) {
314   index_pulse_start = start * 16;
315   index_pulse_end = end * 16;
316 }
317
318 uint8_t* get_sector(uint16_t num) {
319   return (uint8_t*)BOOT_SECT; //For now only use the bootsector until we get SDIO
320 }
321
322 void queue_data(uint8_t* data, uint16_t size, uint8_t am_clock, uint8_t am_data, uint8_t
      am_offset) {
323   int i;
324   uint8_t clock;
325   for (i = 0; i < size; i++, queue_size++) {
326     if (i == am_offset)
327       clock = am_clock;
328     else
329       clock = 0xFF;
330     output_track[queue_size] = calc_fm_encoding(data[i], clock);
331
332   }
333 }
334
```

```
335 void queue_gap(uint8_t data, uint8_t clock, uint16_t size) {
336   uint16_t target;
337   for (target = queue_size + size; queue_size < target; queue_size++) {
338       output_track[queue_size] = calc_fm_encoding(data, clock);
339   }
340 }
341
342 void queue_id_field(uint8_t record, uint8_t track) {
343   generate_id_field(record, track);
344   queue_data((uint8_t*)&id_field, sizeof(struct ID_FIELD), ID_AM_CLOCK, ID_AM_DATA,
          AM_OFFSET);
345 }
346
347 void queue_data_field(uint8_t* data) {
348   generate_data_field(data);
349   queue_data((uint8_t*)&data_field, sizeof(struct DATA_FIELD), DATA_AM_CLOCK, DATA_AM_DATA,
          AM_OFFSET);
350 }
351
352 void queue_sync() {
353   queue_gap(0x00, 0xFF, 6);
354 }
355
356 void queue_track(uint8_t track) {
357   int i;
358   disable_output();
359   reset_queue();
360   //Add our pre-index gap
361   queue_gap(0xFF, 0xFF, 80);
362   //Set physical index pulse boundry
363   set_index_pulse_boundry(OUTPUT_TRACK_SIZE - INDEX_PULSE_SIZE, INDEX_PULSE_SIZE);
364   //Add sync bytes
365   queue_sync();
366   //Add index mark
367   queue_gap(INDEX_AM_DATA, INDEX_AM_CLOCK, 1);
368   //Add Gap1
369   queue_gap(0xFF, 0xFF, 26);
370   queue_sync();
371   for (i = 0; i < SECTORS_PER_TRACK; i++) {
372     queue_id_field(i + 1, track);
373     queue_gap(0xFF, 0xFF, 11);
374     queue_sync();
375     queue_data_field(get_sector(i + track * SECTORS_PER_TRACK));
376     if (i != NUM_TRACKS - 1) {
377       queue_gap(0xFF, 0xFF, 120);
378       queue_sync();
379     }
380   }
381   //Final Gap
382   queue_gap(0xFF, 0xFF, 128);
383   enable_output();
384 }
385
386
387
388 //This routine is called on every update to the PWM timer
389 void TIM1_UP_IRQHandler() {
390     uint8_t local_bit = 15 - current_output_bit % 16; //Grab the current local bit
391     uint16_t local_byte = (current_output_bit /16); //Grab the current local byte
392     TIM1->CCR1 = ((output_track[local_byte] >> local_bit) & 1) ? PULSE_WIDTH : 0;//Set the
          PWM output for the next cycle
393     current_output_bit = (current_output_bit + 1) % (OUTPUT_TRACK_SIZE * 16); //Go to next
          output and stay within
394     NVIC_ClearPendingIRQ(25); //Clear Pending
395   TIM1->SR &= ~0x1; //Clear interrupt flag
396 }
397
398
```

```
399 uint32_t bps;
400 uint32_t target_bps;
401 int32_t bps_percent_error;
402 void generate_statistics() {
403     static uint32_t last_bits = 0;
404     if ((current_output_bit > last_bits) && output_enabled) {
405         bps = (current_output_bit - last_bits) * 10;
406         target_bps = TARGET_BPS;
407         bps_percent_error = (bps < target_bps) ? ((target_bps - bps) * 100)/target_bps : ((
                bps - target_bps) * 100)/target_bps;
408     }
409     last_bits = current_output_bit;
410 }
411
412 void TIM4_IRQHandler() {
413   current_time++;
414     if ((current_time % 100) == 0)  {
415         generate_statistics();
416     }
417   NVIC_ClearPendingIRQ(30);
418   TIM4->SR &= ~0x1; //Clear pending bit
419 }
420
421
422 void EXTI9_5_IRQHandler() {
423   if(!get_pin(DIRECTION_SEL) && current_track > 0) {
424     current_track--;
425   } else if (get_pin(DIRECTION_SEL) && current_track < NUM_TRACKS) {
426     current_track++;
427   }
428
429   EXTI->PR = 0x200; //Mark as handled
430   NVIC_ClearPendingIRQ(23);
431 }
432
433
434 void main()
435 {
436   int first = 1;
437   uint32_t start_time;
438     uint32_t start_head_step_time;
439   uint16_t last_track = current_track;
440     uint8_t index = FALSE;
441
442   set_false(DISK_CHANGE);
443   for(;;) {
444     if (get_pin(MOTOR_ON) && get_pin(DRIVE_SELECT)) {
445             if (last_track != current_track) {
446                 if ((current_time - start_head_step_time) > 20) {
447                     set_false(READY);
448                     queue_track(current_track);
449                     set_true(READY);
450                     start_head_step_time = current_time;
451                     last_track = current_track;
452                 }
453                 start_head_step_time = current_time;
454             }
455       if (first) {
456         start_time = current_time;
457         queue_track(current_track);
458         while(current_time - start_time < 480); //We have to wait 480 ms for "spinup" time
459         set_true(READY);
460         enable_output();
461         first = 0;
462       }
463       if (current_track == 0)
464         set_true(TRACK00);
465       else
```

```
466             set_false(TRACK00);
467                 if (index == TRUE &&
468                   current_output_bit > index_pulse_end &&
469                   current_output_bit < index_pulse_start) {
470                     set_false(INDEX);
471                     index = FALSE;
472                 } else if(index == FALSE &&
473                   current_output_bit > index_pulse_start) {
474                     set_true(INDEX);
475                     index = TRUE;
476                 }
477         }
478         else {
479           disable_output();
480           first = 1;
481           set_false(READY);
482         }
483       }
484 }
```