The Remote Disk Protocol Guide

Bob Applegate
Corsham Technologies, LLC
bob@corshamtech.com

Version 1.1
Last edit date: 04/30/2017
Last edit by: Bob Applegate

Table of Contents

Revision History

Revision	Date	Author	Changes
0.0	Early 2015	Bob Applegate	Initial beta release
0.1	July 2015	Bob Applegate	Added/refined the clock related commands and responses. Removed FORMAT and added new commands to open FAT file for writing.
0.2	Dec 2015	Bob Applegate	Added SAVE_CONFIG command and "command not implemented" error code.
1.0	July 8, 2016	Bob Applegate	Added SET_TIMER command.
1.1	April 30, 2017	Bob Applegate	Added READ_SECTOR_LONG and WRITE_SECTOR_LONG commands. Also reserved more command codes for future use.

Introduction

This project started out of necessity. I was in the process of building a 6800 based clone of the SWTPC computer and needed a way to store programs. Vintage disk controllers were hard to find on the internet and building an analog version was more work than I wanted to take on.

Having done a lot of work with Arduinos, the obvious solution was to use one with an SD shield, some software, and a simply parallel port on the 6800 side. This also made the drive portable so it could be ported to the new 6809 board and eventually to my KIM-1 and other vintage computer systems.

The original version was based on an Arduino UNO with a ATMega128 processor but the limited RAM made the code messier as more features were added, so I finally gave in and moved to an Arduino Mega.

While the current implementation is on a specific platform, there is nothing in the protocol that requires any specific computer on either side of the connection.

The protocol allows for FAT file access as well as mounting DSK format files to act as virtual disks for the host processor. There are very few commands implemented, so users can certainly extend this with their own additions for specific hardware. One of my goals is to add commands to read/write a real-time clock, as an example.

Copyrights

Terminology

Host	The main processor in the system which is running the disk operating system. This protocol is host agnostic.
Disk Command Processor (DCP)	The Arduino based system that is simulating disk drives. This includes the hardware and software.

Command/Response codes (C/R)	The commands sent from the Host to the DCP and returned from the DCP back to the Host.

Overview

The protocol is very simple. The first byte is a command or response, and then zero or more bytes follow with additional data needed by the command/response type. There are no checksums, no error recovery except to try again, and only one transaction may be in progress at any given time. That is, once a command is sent, no other commands can be sent until the response is received.

The protocol does not define the transport mechanism, so it is certainly possible to add error detection/retransmission into the wrapper protocol.

The host processor is in control of the bus; all transactions are initiated by the host, never the drive emulator.

Some may question why there is no error detection/recovery at the protocol level, but that's really a lower-level function. The initial implementation using a parallel interface works flawlessly so it was decided not to add more complication to the protocol.

If someone were to implement this protocol over a serial interface or over something more exotic like a UDP based system, then I would highly encourage error checking and recovery at the transport protocol layer.

Parallel Interface

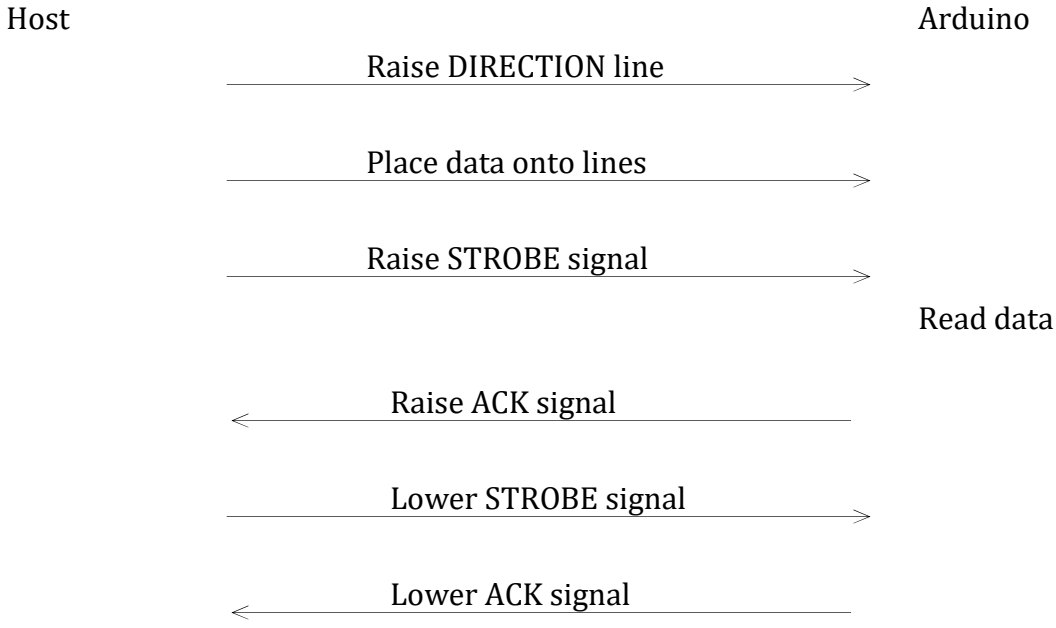
The protocol does not define the connection between the DCP and the host, but since the current implementation uses a parallel interface, this is the description of how it works.

There are eight data lines which change direction based on the direction of the message. While sending a command the data lines are output from the host and input from the DCP, and the direction changes for responses. As soon as a response is done, the direction changes back.

There are also three flow control lines which never change direction. Two are outputs from the host to the DCP and one is from the DCP to the host.

One of the control lines is a direction line. It is high when the host is in control of the data lines and low when the DCP should be controlling them. The DCP should never be transmitting if the direction line is high.

The other two control lines are identical except one is from the host and the other is from the DCP. The sender of a byte must place the data onto the 8 data lines, then raise the control line to indicate valid data is present. The other side then reads the data and raises its control line to acknowledge the data. The sender pulls its control line low, and the receiver then pulls its control line low. That completes the transfer of one byte of data.



Commands/Responses Sorted by Numeric Value

While most values are either a Command or a Response, It is certainly possible they can be both. Since the protocol is not symmetrical, the host expects only responses while the DCP expects only commands.

Some values are reserved for Corsham Tech's use, but any other values are free for others to use. Entries link to the detailed protocol specification for that command/response. If no link is provided, the command is not documented yet.

<table>
<tr><th>Value (hex)</th><th>C/R</th><th>Name</th></tr>
<tr><td>01</td><td>C</td><td>GET_VERSION</td></tr>
<tr><td>02-04</td><td></td><td>Reserved (future time/date functions)</td></tr>
<tr><td>05</td><td>C</td><td>PING</td></tr>
<tr><td>06</td><td>C</td><td>LED_CONTROL</td></tr>
<tr><td>07</td><td>C</td><td>GET_CLOCK</td></tr>
<tr><td>08</td><td>C</td><td>SET_CLOCK</td></tr>
<tr><td>09-0F</td><td></td><td>Reserved</td></tr>
<tr><td>10</td><td>C</td><td>GET_DIRECTORY</td></tr>
<tr><td>11</td><td>C</td><td>GET_MOUNTED LIST</td></tr>
<tr><td>12</td><td>C</td><td>FILE_MOUNT</td></tr>
<tr><td>13</td><td>C</td><td>FILE_UNMOUNT</td></tr>
<tr><td>14</td><td>C</td><td>WRITE_FILE
Command code: 1B
Valid responses: ACK, NAK
Frame:
<table><tr><td>Offset:</td><td>0</td><td>1-X</td><td>X+1</td></tr><tr><td>Contents:</td><td>1B</td><td>Filename</td><td>00</td></tr></table>This will cause a new file to be created for writing. This is followed by a sequence of WRITE_BYTES commands to write the contents of the file. The filename must meet the requirements of the DCP and is not defined here. At the end of the filename is a null.

Note that only one file can be open at a time. If another is opened while one is already open, the results are undefined.</td></tr>
</table>

<table>
<tr><td></td><td></td><td>

WRITE_BYTES

Command code: 1C
Valid responses: ACK, NAK
Frame:

<table>
<tr><td>Offset:</td><td>0</td><td>1</td><td>2-X</td></tr>
<tr><td>Contents:</td><td>1C</td><td>Length</td><td>Bytes</td></tr>
</table>

This is used to write data to a file that has been opened with the WRITE_FILE command. The length can be from 0 to 255; a value of 0 indicates 256 bytes. Ie, it is not possible to write zero bytes. The number of bytes specified are then sent. The data is raw; whatever is included in the message is exactly what will be written to the disk file.

After all bytes have been sent, a DONE/ABORT command is sent to close the file.

SAVE_CONFIG

Command code: 1D
Valid responses: ACK, NAK
Frame:

<table>
<tr><td>Offset:</td><td>0</td></tr>
<tr><td>Contents:</td><td>1D</td></tr>
</table>

This is used to save the current mounted drives to the default configuration file. At the next restart, the new configuration will be used.

GET_DRIVE_STATUS

</td></tr>
<tr><td>15</td><td>C</td><td>DONE/ABORT</td></tr>
<tr><td>16</td><td>C</td><td>READ_FILE</td></tr>
<tr><td>17</td><td>C</td><td>READ_BYTES</td></tr>
<tr><td>18</td><td>C</td><td>READ_SECTOR</td></tr>
<tr><td>19</td><td>C</td><td>WRITE_SECTOR</td></tr>
<tr><td>1A</td><td>C</td><td>GET_MAX_DRIVES</td></tr>
<tr><td>1B</td><td>C</td><td>WRITE_FILE</td></tr>
</table>

1C	C	WRITE_BYTES
1D	C	SAVE_CONFIG
1E	C	SET_TIMER
1F	C	READ_SECTOR_LONG
20	C	WRITE_SECTOR_LONG
21-3F		Reserved
40-7F		Available
80		Reserved
81	R	VERSION_INFO
82	R	ACK
83	R	NAK
84		Reserved
85	R	PONG
86		Reserved
87	R	CLOCK_DATA
88-8F		Reserved
90	R	DIRECTORY_ENTRY
91	R	DIRECTORY_END
92	R	FILE_DATA
93	R	DRIVE_STATUS
94	R	SECTOR_DATA
95	R	MOUNT_INFO
96	R	MAX_DRIVES – NOT DONE YET
97-AF		Reserved
B0-FF		Available

Commands/Responses Sorted by Function

Misc

Get version information
Ping/Pong
Setting LEDs

Mounting/Unmounting DSK Format Files

Mount
Unmounts
Status

List of mounted drives

Working with FAT Files

Request Directory
Next Directory Entry
Open File
Get Bytes from File

Working with Mounted Files

Sector Read
Sector Write
Sector Read Long
Sector Write Long
Status

Time/Date

Get current time/date
Set time/date

Common Values

Unless otherwise specified, numeric values are in hexadecimal.

Drive

The disk drive number is zero based. The highest value supported by a given version of DCP is implementation dependent, but at least four drives (0 to 3) must be supported.

Sector Size

Many commands allow the specification of sector size. This is always a one byte field and only the following values are valid:

Value	Actual size (decimal)
1	128
2	256
3	512
4	1024

Table 1

Another other value is undefined and the implementation of the DCP might choose a default value.

Error Codes

A standard set of common error codes is specified:

Value (decimal)	Error
00	No error
10	Drive not mounted
11	Drive already mounted
12	File not found
13	Read only
14	Illegal drive number
15	Illegal track number
16	Illegal sector number
17	Read error
18	Write error
19	Device not present

20	Command not implemented. Usually the result of a command sent to a shield that does not support the most recent command.

Table 2

Details of Commands/Responses

These are the more detailed explanations of the messages. The order is random; mostly the order that I entered the data, cut and pasted, etc.

GET_VERSION

Command code: 01
Valid responses: VERSION_INFO
Frame:

Offset:	0
Contents:	01

The DCP must be able to provide some information about itself.

VERSION_INFO

Response code: 81
Frame:

Offset:	0	1-X
Contents:	81	Version

This is the response to a request for version information. It comes back as an ASCII string. The first part is the manufacturer/developer of the DCP, ending with a CR/LF sequence, followed by an ASCII version number, terminated with a 0 byte.

PING

Command code: 05
Valid responses: PONG
Frame:

Offset:	0
Contents:	05

Ping can be sent to the DCP which will reply with a Pong. This is a test message that verifies communication to/from the DCP works and that the DCP is functioning.

PONG

Response code: 85
Frame:

Offset:	0
Contents:	85

This is the response to a Ping command and indicates the DCP got and processed the Ping command.

LED_CONTROL

Command code: 06
Valid responses: None
Frame:

Offset:	0	1	2	3
Contents:	06	Bitmap	Bitmap	Bitmap

This command controls which LEDs on the DCP can be remotely controlled and can also set them on or off.

The bitmap at offset 1 has a bit set for each LED that is controllable via the protocol. If a bit is not set, then the LED is controlled by the DCP software.

The bitmap at offset 2 has a 1 bit for every bit to be modified by this command. If only one LED is to be changed, only one bit will be set.

The bitmap at offset controls the setting of the LED(s) which have their bits set in offset 2. If a bit is set, the LED is turned on.

Note that the map of LEDs in this command to actual LEDs on the board is undefined, and there are no indications if the user attempts to control non-existent LEDs. By default, no LEDs are under user control, so setting bits in offset 2 and 3 without corresponding bits in offset 1 will make no changes to the LEDs.

DONE/ABORT

Command code: 15
Valid responses: No response sent
Frame:

Offset:	0
Contents:	15

This is a way to stop some commands from continuing to send data. It can be used instead of READ_BYTES to abort the transfer of a long file.

READ_SECTOR

Command code: 18
Valid responses: SECTOR_DATA, NAK
Frame:

Offset:	0	1	2	3	4	5
Contents:	18	Drive	Sector size	Track	Sector	Sectors per track

Offset 1 is the zero-based drive number.

Offset 2 is the sector size. See Table 1.

Offset 3 is the zero based track number, or the high part of the sector number if using sector-only numbering.

Offset 4 is the zero based sector number, or the low part of the sector number if using sector-only numbering.

Offset 5 specifies the number of sectors per track. If the value is zero, then the track and sector are treated as a zero-based 16 bit sector number with the high part in offset 3 and the low part in offset 4.

The value at offset 5 can change between commands; some operating systems get this value from a sector in track 0, then have it for future operations. For example, FLEX has this at offset 27 (hex) in the System Information Record.

WRITE_SECTOR

Command code: 19
Valid responses: ACK, NAK
Frame:

Offset:	0	1	2	3	4	5	6-X
Contents:	19	Drive	Sector size	Track	Sector	Sectors per track	Data

Offset 1 is the zero-based drive number.

Offset 2 is the sector size. See Table 1.

Offset 3 is the zero based track number, or the high part of the sector number if using sector-only numbering.

Offset 4 is the zero based sector number, or the low part of the sector number if using sector-only numbering.

Offset 5 specifies the number of sectors per track. If the value is zero, then the track and sector are treated as a zero-based 16 bit sector number with the high part in offset 3 and the low part in offset 4.

The value at offset 5 can change between commands; some operating systems get this value from a sector in track 0, then have it for future operations. For example, FLEX has this at offset 27 (hex) in the System Information Record.

Starting at offset 6 is the data to be written. The number of bytes must match the sector size.

ACK

Response code: 82
Frame:

Offset:	0
Contents:	82

This is a common response to indicate the last command was completed without error.

NAK

Response code: 83
Frame:

Offset:	0	1
Contents:	83	Error Code

This response indicates the last operation encountered an error and may not have completed. The error code values are in Table 2.

SECTOR_DATA

Response code: 94
Frame:

Offset:	0	1-X

Contents:	94	Data

This is in response to a sector read command. The data for the requested sector is returned. There must be exactly the same number of bytes as are in the sector, as the host will be expecting that exact number.

GET_DIRECTORY

Command code: 10
Valid responses: DIRECTORY_ENTRY, DIRECTORY_END
Frame:

Offset:	0
Contents:	10

This requests the directory for the current disk drive. Effectively, get a directory of the FAT filesystem that is currently being used as a drive. This might be too closely tied to FAT; someone might have a better idea for this. Also note that there is no option to get a directory of any other drive… that logic seemed way too complicated.

DIRECTORY_ENTRY

Response code: 90
Frame:

Offset:	0	1-X	X+1
Contents:	90	ASCII	00

This is a response to a request for a disk directory. Each file is returned one at a time in a DIRECTORY_ENTRY response. The filename is in ASCII and is terminated by a null.

Note that what is returned and in what format depends on the DCP. For the Arduino code, only files in the current directory are returned, and they are all shortened to 8.3 format and all upper case.

DIRECTORY_END

Response code: 91
Frame:

Offset:	0
Contents:	91

This indicates the end of the directory entries. Note that this does not contain an entry. It is also used to mark the end of a list of mounted drives.

READ_FILE

Command code: 16
Valid responses: ACK, NAK
Frame:

Offset:	0	1-X	X+1
Contents:	16	Filename	00

This requests a specific file to be opened and made ready to read. This is followed by a sequence of READ_BYTES commands to get the contents of the file. The filename must meet the requirements of the DCP and is not defined here. At the end of the filename is a null.

Note that only one file can be open at a time. If another is opened while one is already open, the results are undefined.

READ_BYTES

Command code: 17
Valid responses: FILE_DATA
Frame:

Offset:	0	1
Contents:	17	Length

This requests bytes from an open file. The sender must specify the maximum number of bytes that can be sent at a time.

FILE_DATA

Response code: 92
Frame:

Offset:	0	1	2-X
Contents:	92	Length	data

This is sent with data from an open file. The length field specifies the number of bytes in this message, or 0 if end of file has been reached.

WRITE_FILE

Command code: 1B
Valid responses: ACK, NAK
Frame:

Offset:	0	1-X	X+1
Contents:	1B	Filename	00

This will cause a new file to be created for writing. This is followed by a sequence of WRITE_BYTES commands to write the contents of the file. The filename must meet the requirements of the DCP and is not defined here. At the end of the filename is a null.

Note that only one file can be open at a time. If another is opened while one is already open, the results are undefined.

WRITE_BYTES

Command code: 1C
Valid responses: ACK, NAK
Frame:

Offset:	0	1	2-X
Contents:	1C	Length	Bytes

This is used to write data to a file that has been opened with the WRITE_FILE command. The length can be from 0 to 255; a value of 0 indicates 256 bytes. Ie, it is not possible to write zero bytes. The number of bytes specified are then sent. The data is raw; whatever is included in the message is exactly what will be written to the disk file.

After all bytes have been sent, a DONE/ABORT command is sent to close the file.

SAVE_CONFIG

Command code: 1D
Valid responses: ACK, NAK
Frame:

Offset:	0
Contents:	1D

This is used to save the current mounted drives to the default configuration file. At the next restart, the new configuration will be used.

GET_DRIVE_STATUS

Command code: 14
Valid responses: DRIVE_STATUS
Frame:

Offset:	0	1
Contents:	14	Drive

This is meant as a way to simulate a DOS requesting the status of a disk drive as if it were getting status from a disk controller chip.

DRIVE_STATUS

Response code: 93
Frame:

Offset:	0	1
Contents:	93	Status

The status field is a bitmap:

Bit	Meaning when set
0	Drive is mounted (disk is present)
1	Read-only (clear indicates read/write)

GET_MOUNTED LIST

Command code: 11
Valid responses: MOUNT_INFO, DIRECTORY_END
Frame:

Offset:	0
Contents:	11

This requests a list of mounted drives. Each entry will be sent back in a MOUNT_INFO message and after the last one, a DIR_END will be sent. Note that the MOUNT_INFO messages will arrive one after another without the host needing to request the next.

MOUNT_INFO

Response code: 95
Frame:

Offset:	0	1	2	3-X	X+1
Contents:	95	Drive	Read-only	Filename	0

This record indicates a single mounted drive. The DCP should send a status for all supported drives, not just mounted ones. In other words, in the DCP supports four drives but none are mounted, four of these records should be returned, one per drive, with no filename.

If Read-only is 0 then the drive can be written to. If non-zero then it is a read-only drive.

FILE_MOUNT

Command code: 12
Valid responses: ACK, NAK
Frame:

Offset:	0	1	2	3-X	X+1
Contents:	12	Drive	Read-only	Filename	0

This instructs the DCP to mount a specified file to a specified drive. If the drive is already mounted or if the file does not exist, this will return an error.

Read-only can be either a 0 (drive can be written) or non-zero (drive cannot be written to).

FILE_UNMOUNT

Command code: 13
Valid responses: ACK, NAK
Frame:

Offset:	0	1
Contents:	13	Drive

This tells the DCP to unmounts the filesystem immediately. If there are pending writes, they must be performed first. If the drive is not mounted, then no error is indicated.

GET_CLOCK

Command code: 07
Valid responses: CLOCK_DATA, NAK
Frame:

Offset:	0
Contents:	07

This requests the current time/date from the hardware clock.

CLOCK_DATA

Response code: 87
Frame:

Offset:	0	1	2	3	4	5	6	7	8
Contents:	87	Month	Day	Year high	Year Low	Hour	Min	Sec	DOW

This returns real-time clock data if a clock is available. Each field is a BCD (binary coded decimal) version.

Month: 01-12. This is the true month, one based, in binary.

Day: 01-31, in binary.

Year: This is a four digit year. While most uses will never go back in time, a demo system might want to pretend to be in the 1970s so the full year can be specified. If the clock hardware does not support centuries, then the upper byte should be zero.

The hour is from 0 to 23, in binary.

Minutes and seconds are from 00 to 59, in binary.

The DOW (Day Of Week) value ranges from 0 (Sunday) to 6 (Saturday).

Note that if the clock hardware does not support any of these fields, a value of $FF should be placed in that field to indicate it is not supported. Applications receiving the clock data must allow for this.

SET_CLOCK

Command code: 08

Valid responses: ACK, NAK
Frame:

Offset:	0	1	2	3	4	5	6	7	8
Contents:	08	Month	Day	Year high	Year Low	Hour	Min	Sec	DOW

This sets the clock. The data format is exactly as described for the CLOCK_DATA message.

SET_TIMER

Command code: 1E
Valid responses: ACK, NAK
Frame:

Offset:	0	1
Contents:	13	Timer value

This tells the DCP to begin generating interrupts at the specified interval. The interval values are somewhat arbitrary, based on what I wanted and what the Arduino code could easily do. More values can be added.

Timer Value (decimal)	Time
0	0 – disable interrupts
1	10 ms
2	20 ms
3	30 ms
4	40 ms
5	50 ms
6	100 ms
7	250 ms
8	500 ms
9	1 second
10-31	Reserved
32-255	Available

READ_SECTOR_LONG

Command code: 1F
Valid responses: SECTOR_DATA, NAK
Frame:

Offset:	0	1	2	3	4	5	6
Contents:	1F	Drive	Sector size	Sector MSB	Sector	Sector	Sector LSB

Offset 1 is the zero-based drive number.

Offset 2 is the sector size. See Table 1.

Offsets 3 to 6 are a large sector number, MSB first. There are no tracks for this command, only sectors. This is a zero-based value. Note that the Arduino code might have limitations on how large of a sector number it can support.

WRITE_SECTOR_LONG

Command code: 20
Valid responses: ACK, NAK
Frame:

Offset:	0	1	2	3	4	5	6	7-X
Contents:	20	Drive	Sector size	Sector MSB	Sector	Sector	Sector LSB	Data

Offset 1 is the zero-based drive number.

Offset 2 is the sector size. See Table 1.

Offsets 3 to 6 are a large sector number, MSB first. There are no tracks for this command, only sectors. This is a zero-based value. Note that the Arduino code might have limitations on how large of a sector number it can support.

Starting at offset 7 is the data to be written. The number of bytes must match the sector size.