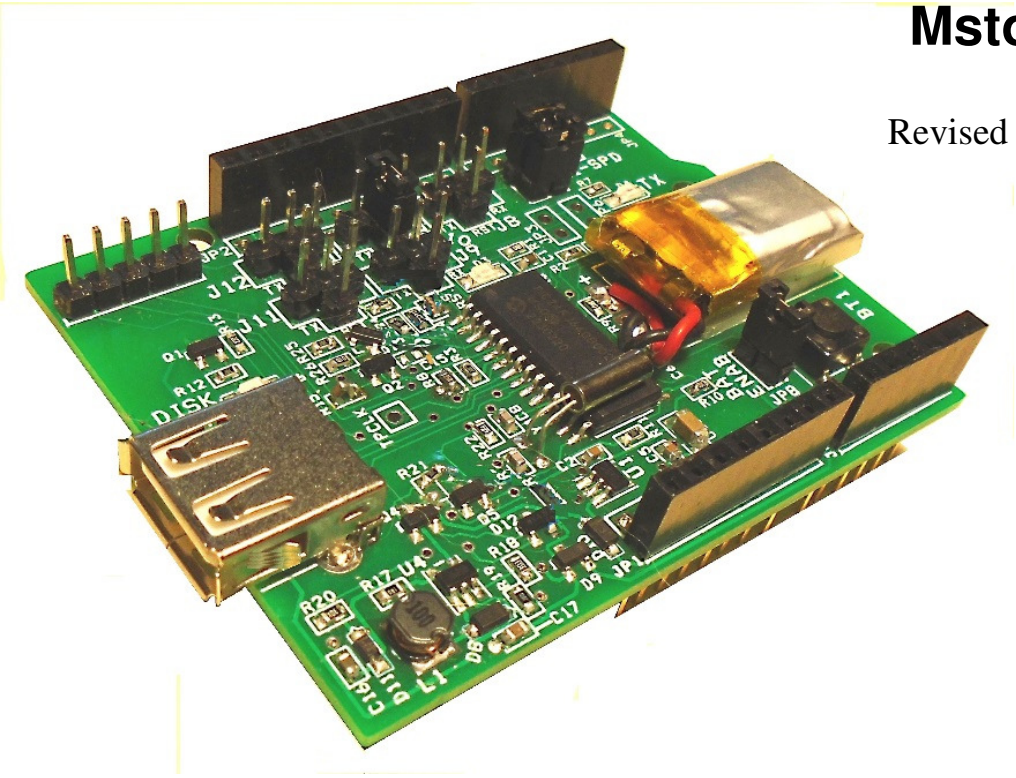
Versalent

www.versalent.biz

ARDUINO MASS STORAGE SHIELD



Mstor Manual

Version 1.04

Revised Feb 20, 2023

General Description :

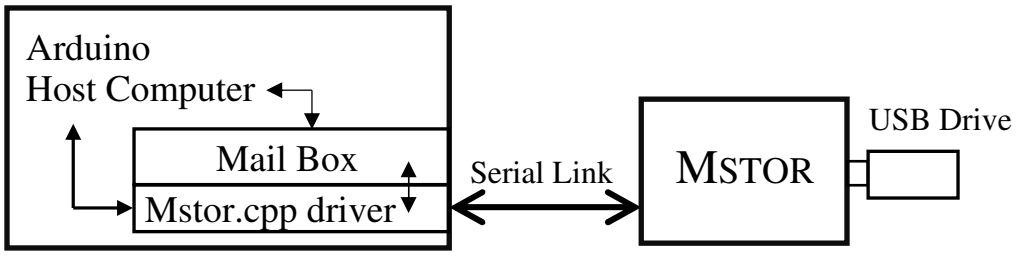
Mstor is as Arduino shield that provides access to a standard USB FAT-formatted drive of up to 16 gigabytes. It operates as a USB mini-host and offers all standard file access features like read/write/append/delete/rename/copy and concatenate files, as well as making, deleting and navigating directories. FAT formatting makes the drives compatible with PC's for data portability.

Communications is via a serial port to minimize the required I/O, and Mstor can transfer up to 18kbytes per second when using a built-in Serial Port[1] , or 3kbytes per sec when using a SoftwareSerial port. Commands and data are transferred to/from Mstor in framed, error-checked packets for high reliability.

This manual describes both Mstor SHIELD (with internal battery) and Mstor BASIC (no battery). Features described below that are provided by the battery do not function for model Mstor BASIC.

The Mstor Driver:

The driver's methods that are called to execute file operations do not return until the function is complete (synchronous operation). The host calls methods like fileOpen() , fileRead() etc to issue commands and a 1- byte result is returned indicating success or error. If data is also returned as with the fileRead() command, that data arrives in the (32-byte) mailbox along with a count of the number of bytes.



The methods which are available include all standard file functions. Refer to the methods-table. These methods are accessible once an Mstor object is created:

Mstor myMstor; //creates an Mstor object. Note no () parentheses.

A Few Common Methods	**DESCRIPTION**
myMstor.fileOpen(mode, fname)	Opens specified file in Read/Write/Append mode
myMstor.fileRead()	Reads file record, put data in Mstor.data[] array
myMstor.fileWrite(string) or myMstor. fileWrite()	Can write either strings, or binary data previously placed in the Mstor.data[] array
myMstor.fileDelete(fname)	Delete the specified file

The library of methods is extensive and includes methods for modifying/managing the directory structure, navigating that structure, managing the real-time clock's data/time settings, copying files etc. Refer to the methods-list later in this manual.

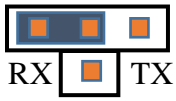
Mstor Configuration :

Mstor requires 3 of the Arduino's digital signals. JP8-JP12 jumper blocks allow you to select which 3 signals D8-D12 are used for the serial TX, RX and Reset. JP5 also allows selection of D0 & D1 for RX and TX which uses the high-speed HardwareSerial port (and therefore disables the Serial Monitor usage).

To use a SoftwareSerial port for Mstor (38.4kbaud maximum) :

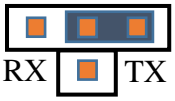
[This is the simplest and most convenient configuration since it retains the Arduino SerialMonitor for easy debugging]

1) Select only one of D8-D12 to be the RX signal using jumpers J8-J12 respectively. Place a jumper to the left in the selected J8-J12 block as shown



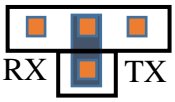
Default J8

2) Select only one of D8-D12 to be the TX signal using jumpers J8-J12 respectively. Place a jumper to the right in the selected J8-J12 block as shown



Default J9

3) Select only one of D8-D12 to be the Mstor Reset signal using jumpers J8-J12 respectively. Place a jumper centered vertically in the selected J8-J12 as shown.



Default J10

4) Remove all jumpers from JP5
5) Edit Mstorcfg.h to change the #define TX_PIN and RX_PIN and RESET_PIN to match the Dx Arduino signals of your jumper selections.
6) Edit Mstorcfg.h to insure that **#define DEBUG 0** exists.

The default factory configuration is shown above. RX = D8 (J8) , TX = D9 (J9) and Reset = D10 (J10) with Mstorcfg.h settings to match.

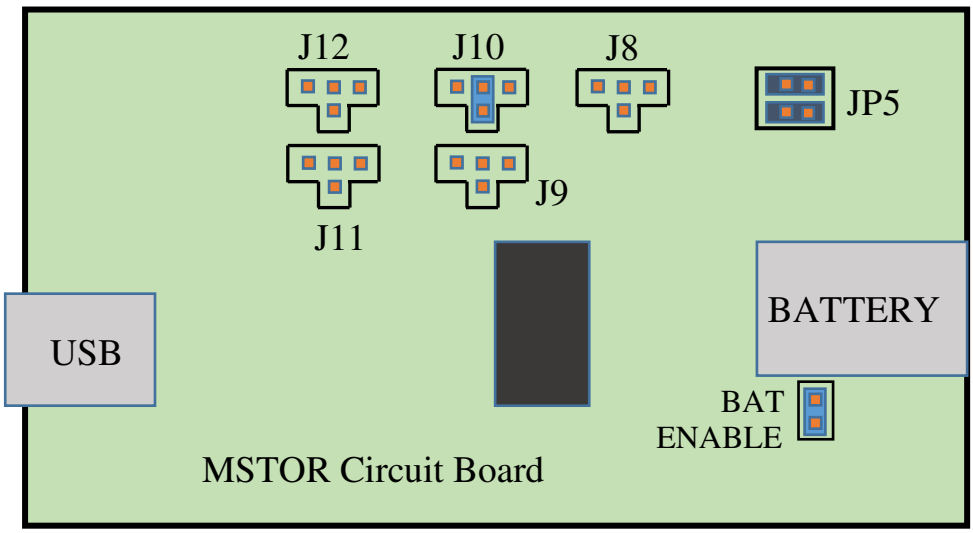
To use Arduino's HardwareSerial port (250k baud maximum) :

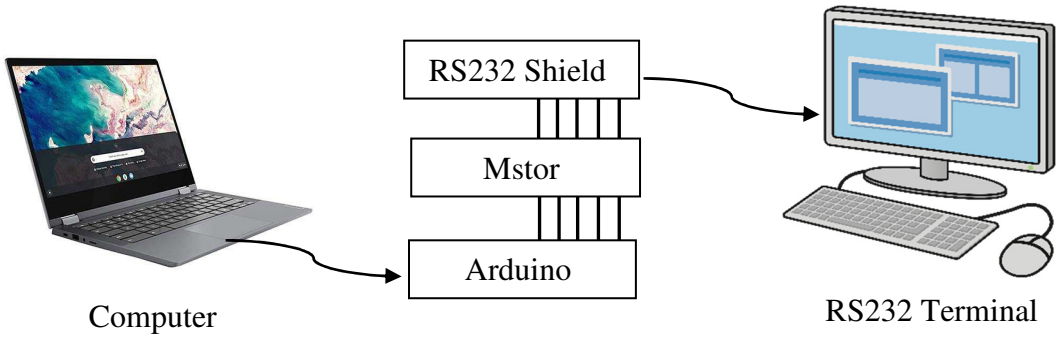
[A more complicated configuration, but provides higher disk speed]

This allows faster disk transfers by sacrificing the internal Serial Monitor feature making the (fast) HardwareSerial port available to Mstor.

1) FIRST you must change Mstor's baud rate using the setBaud(x) method… see below. [The Arduino Sample Application menu offers 'Ex' to do this: x = 0-8 to set the new baud rate.] Then power down to change the jumpers.
2) Remove the TX and RX (horizontal) jumpers from J8-J12 as shown.
3) Install one Reset jumper (center vertical) in one of J8-J12 as shown below.

4) Install two jumpers horizontally in jumper block JP5 as shown below to connect Arduino's D0 as signal RX & D1 as signal TX for the HardwareSerial port.
5) Edit Mstorcfg.h (in the Arduino Mstor library) .. comment out the line: //define DEBUG 0 near the top of the file, then compile/download the new Arduino code. If the application uses Serial Monitor output like the Arduino Sample App, you need to install and configure am RS232 shield connected to a serial terminal. See the High-Speed Connection Diagram below.





High-Speed Connection Diagram

Mstor uses Arduino's HardwareSerial port at up to 250k baud. If a Serial Monitor is needed, it must be external since the built-in Arduino Serial monitor is no longer available

Baud Rate Recovery:

If the baud rate is changed to a high-baud value (higher than 38.4k) , Arduino must then communicate with Mstor at that high baud rate which requires a HardwareSerial port. If one is not available, Mstor's baud-recovery procedure will restore it to the factory-default 38.4k:

1) Install Mstor Jumper JP6
2) Appy power to Mstor and press (Mstor's) reset button
3) Remove Jumper JP6

Operating Details :

Please refer to the sample application provided. It is a fully functional program that allows you to exercise all Mstor features and can be used as-is, as a base for your own applications, or as a cut/paste repository to include specific methods in your own apps. To start an app from scratch .. an Arduino Template is also provided which provides the basic structure of a disk application.

The required includes which should appear at the application top:

#include <SoftwareSerial.h>	can operate reliably up to 38.4kBaud
#include <MsTimer2.h>	2-25ms timer ISR to call myMstor.run_ms_timer()
#include <Mstor.h>	non-user-editable constants and method prototypes
#include <Mstorcfg.h>	a few editable constants
#include "Arduino.h"	provides access to Arduino features

Commands are sent using the methods provided. Some require one or more parameters which are typically strings however there are a few that use parameters of other data types. See the individual command descriptions. Commands like fileOpen() change the *mode* of Mstor. Some commands require Mstor to be in a particular mode. For instance, to successfully execute the fileRead() command, Mstor must be in the READ MODE – which it enters by executing fileOpen('R', filename). Other disk commands are dis-allowed in this mode Non-disk related commands (such as getTime() , getDate() and more) can operate with Mstor in any mode . These modes, the commands that initiate them, and the commands that can be execute in these modes are shown below:

Mode	Command to Enter Mode	Commands Allowed	Comments
COMMAND	Default	All	Power-up mode. Other modes exit to this mode.
READ	fileOpen('R',xx)	fileRead() fileClose() exitMode(), non-disk	Return to COMMAND MODE using fileClose() or exitMode()
SEEK	fileSeek(file,ofst)	fileRead() fileClose() exitMode(), non-disk	Return to COMMAND MODE using fileClose() or exitMode()
WRITE	fileOpen('W',xx)	fileWrite(), fileWrite(str), fileClose() exitMode(), non-disk	Return to COMMAND MODE using fileClose() or exitMode()
APPEND	fileOpen('A',xx)	fileWrite(), fileWrite(str), fileClose() exitMode(), non-disk	Return to COMMAND MODE using fileClose() or exitMode()
FILELIST	fileList(mask)	listRead() (To read filenames in a directory)	Exits to COMMAND MODE on reaching end-of-list, or exitMode() executed
DIRLIST	dirList(mask)	listRead() (To read subdir names in a directory)	Exits to COMMAND MODE on reaching end-of-list or exitMode() executed
COPY	copyFile()	none	Exits to COMMAND MODE on completion. Outputs a series of asterisks during copy operation.
CONCAT	concatFiles()	none	Exits to COMMAND MODE on completion. Outputs a series of asterisks during copy operation

Operating Modes:

Mstor is always in one of the 9 modes above. Disk-related modes are restrictive in that only appropriate commands execute when they are in effect. (i.e. fileWrite() will not be accepted within READ MODE). Non-disk type commands like getTime() will execute while in any mode.

1) **COMMAND MODE**: The Mstor command processor is idle, and it can receive and execute any of its available commands from the table below.

2) **WRITE MODE:** Mstor has received a fileOpen('W', filename) command and is expecting the host to send write commands. The host can use:

   a) fileWrite(string) commands to write ASCII strings (up to 45 characters max) to the file .. or
   b) fileWrite() with no parameter to write the (up to 32) binary bytes of data previously loaded into the Mstor.data[] array. Mstor.data_count must be set to the number of bytes loaded.

   When the host has no more data to send it sends a fileClose() command to close and return to COMMAND MODE

3) **APPEND MODE:** Mstor has received a fileOpen('A', filename) command. Same as write mode above except that an existing target file is not overwritten.

4) **READ MODE:** Mstor has received a fileOpen('R', filename) and is expecting readFile() command(s) to return up to 32-bytes of file data in Mstor.data[]. Successive calls continue to return file data until the host sends a fileClose() to end the mode. If /when end-of-file (EOF) is reached, Mstor will return 0-length data -- host should then issue a fileClose() to close and return to Command Mode**.**

5) **SEEK MODE**: Mstor has received a fileSeek(filename, offset) and is expecting readFile() command(s) to return up to 32-bytes of file data in Mstor.data[]. The initial read starts from the offset (bytes) specified. Successive calls continue to return file data until the host sends a fileClose() to end the mode. If /when end-of-file (EOF) is reached, Mstor will return 0-length data -- host should then issue a fileClose() to close and return to Command Mode**.**

6) **FILELIST MODE:** Mstor has received a fileList() command and built a list of files from the specified directory. Mstor then expects listRead() commands to retrieve file-items in the Mstor.data[] attay. Each file item describes one file. Once the last file item has been returned, Mstor returns a 0-length message. The host can end the mode at any time by executing exitMode().

   The file-item list contains all the matching files in the specified directory and each item is formatted as ASCII characters. The data is compressed since 32 characters is not enough to include all date-field and time-field separators.

Filename.extCreate-dateCreate-timeFileSize

Filename is 8.3 format and occupies exactly 12 characters (space-padded to 12)
Create-date is MMDDYY and occupies the next 6 characters
Create-time is HHMMSS[A or P] (12-hr) and occupies the next 7 characters
FileSize is up to 5 characters and uses 'k', 'M', and 'G' suffixes

7) **DIRLIST MODE:** Mstor has received a list dirList() command and is expecting readList() commands as above in List Files Mode to return directory list items. Item format is the same as for List Files Mode.

8) **COPY MODE:** Mstor has received a CopyFile() command and is in the process of making the copy. It first returns an ACK/NAK-message indicating command acceptance, then copies the file with no further host commands. The copy process occurs entirely within Mstor, The indicator of completion is data_ready = 1 and error_code = 0. The copy rate is approximately 8k bytes/sec so a 100k file would require about 13 seconds to complete.

9) **CONCAT MODE:** Mstor has received a ConcatFiles() command and is in the process of making the combination. Operation is identical to the CopyFile() command above.

Command Examples:

Simple command to get/display current time (executes in any mode):

```
error_code = myMstor.getTime();
if (error_code == 0)  Serial.print(myMstor.data, myMstor.data_count);
else
{
        Serial.write("Error – ");
        Serial.println(error_code);
}
```

```
//Code to read an entire file

error = myMstor.fileOpen('R', "file1.ext");     //open file in 'R'ead mode
if (error == 0)                                 //check for open success
{
    done = false;
    while (done == false)                       //loop to read records
    {
        error = myMstor.fileRead();             //read a record up to 32 bytes
        if (error == 0)                         //check for read error
        {
            for (i=0;i<myMstor.data_count;i++)  //access the record bytes
            {
               Serial.write(myMstor.data[i]);  //write each byte to screen
            }
            if (myMstor.data_count==0) done=true;  //when 0 count, no more data
        }
        else
        {
           process_error(error);                //on read error .. do something
           done = true;                         //and exit
        }
     }                                          //all data has been read
     closeFile();                               //close file which exits READ_MODE
}
else process_error(error);                      //file could not be opened
```

Some Restrictions:

- fileRead() and fileWrite() transfer a maximum of 32 bytes of data for each call since Mstor.data[] array is 32 bytes in size while fileWrite(string) can write strings up to 45 characters in length.
- Only **ONE FILE CAN BE OPEN** at a time.
- For Arduino's with only one HardwareSerial port that is required for the Serial Monitor .. a SoftwareSerial must be used for Mstor – software ports are low performance and cannot reliably operate above 38.4k baud
- Mstor's USB port provides limited power (less than a standard USB2.0 port) so cannot support rotating media. Mstor accepts ONLY FLASH-DRIVES that require 125mA maximum.
- File and Directory names MUST be in 8.3 format – although Windows allows the use of long-winded names, Mstor does not. Directory paths such as SubDir1/SubDir2/filename.ext are limited to 50 characters. File and directory names are not case sensitive.

Disk Drive Safety:

!DATA SAFETY!

As with PC's, random unplugging or power loss of an active USB drive can result in partial or complete loss of data and/or corrupted drive formatting.

BEFORE REMOVING OR PLUGGING-IN A USB DRIVE, the drive power must be turned off using the setPower() command. Visually check that the red DRIVE-POWER LED is OFF before manipulating a USB drive. (Similar to the Windows 'Safety Remove' function.)

RED = STOP! do not plug/unplug the drive !

- **The Internal Battery (Model Mstor SHIELD only) :**

  Besides powering the real-time clock, Mstor's internal battery ***protects the USB drive from power-fail corruption.*** If drive-power is abruptly removed during a USB transaction, the format can be damaged rendering the entire disk unreadable. The battery operates like a miniature UPS[2] maintaining USB power until any USB-transactions-in-progress are completed before a controlled shutdown is executed.

  The battery can provide full operating power to Mstor for more than 5 minutes (orderly shutdown occurs in < 2 seconds). So even a partially charged battery can

protect the disk. Mstor shuts down to < 5uA of battery current so can be stored unused, with its clock running for about a year. If fully discharged when power is applied, Mstor can be operated immediately although battery-protection will not be available for the 1-hour required for minimum charging. There is a checkBattery() command which reports on the battery state.
Model Mstor BASIC is a reduced-cost model which does not provide a battery.

- **Data Integrity and Reliability:**

  All commands and data are transferred in CRC'd messages
  - Commands are executed by calling methods and a corrupted command will return an error code after a single attempt. Error code in Appendix 1.
  - Disk data is transferred using an error-correction protocol [except for fileWrite(string)] – if a data packet does not pass CRC validation, the receiver automatically requests a re-transmission (up to 3X before giving up and reporting an error). Re-transmission occurs without the caller's intervention so the correction is automatic.

  One of the Arduino signals required provides a Reset to Mstor. This reset is used to awaken Mstor from its Sleep state after power is applied. Resetting Mstor during a USB transaction could damage the disk format. To prevent this, Mstor internally blocks the reset signal during command execution.

- **Command Timeouts:**

  Since Mstor commands operate synchronously, they are 'blocking commands'. This means that if a command fails to return .. the Arduino application would be permanently locked up. To prevent this, each time a command method is invoked the driver automatically starts a command-timer. If the command timeout interval passes and the command has not completed, this timer forces the method to return – reporting an error but allowing the application to continue operating.

File System Limitations:

Mstor provides the filesystem which allows a host to read/write/edit/move/copy etc files throughout the USB drives directory structure. Wildcard masks are only allowed with the fileList(), and dirList() commands. See below. This means that Mstor cannot operate on batches of files .. for instance cannot delete or copy a list of files like a PC. Filepaths can use either a foreslash '/' or backslash '\' character between directories. Mstor provides a real-time battery-backed clock to maintain file time stamps. The battery recharges while power is applied, and lasts for 1+ years with power removed (sleeping with only its real-time clock running);

Data Transfer Speed:

Transfer speeds are limited by the serial port speed, Mstor.data[] array size, and the request/response interaction of the protocol used. At 9600 baud Mstor can transfer about 700 bytes per second and at 250k baud about 18k bps.

Directory and File Names:

All directory and filenames must be in 8.3 format. The top level (root) directory is identified by a single '/' or '\' character. Commands can be executed while in any directory, and can operate on files in any other directory using the appropriate path ahead of the filename. Paths starting with the '/' are absolute -- they start at the root directory and follow the specified subdirectory path. Paths not starting with '/' are relative to the current directory. So to delete Myfile which is two levels down under the root directory (while in any other directory), use :

result = deleteFile("/subdir1/subdir2/myfile"); //note filename NOT case sensitive

Or to delete Myfile which is two levels down below the current directory, use:

result = deleteFile("subdir1/subdir2/myFILE");

Notice that the case of myfile in the above examples is different in the examples. The file system is case-insensitive, so myfile matches with MYFILE and MyFile.

The double-dot syntax is available to specify a directory one level up from the current position in the directory tree. With Mstor currently in the botmdir directory:

(path from the root) /mydir/mydir1/nextdir.dir/botmdir

To move up one level : result=changeDir("..") //moves to /mydir/mydir1/nextdir.dir

Path/filename strings can be a total of 50 characters long. However command strings are limited to 80 characters, so paths may have to be shortened for commands that specify multiple files.

Directory Structure Complexity:

While there are no commands to return the entire directory tree, getCurrentDir() provides the complete path from the root directory to the current directory. Ex:

/mydir/mydir1/nextdir.dir/botmdir

If using a drive containing a complex, multi-branch directory structure, Mstor's host must have knowledge of that structure in order to navigate the tree. Because there is no visual representation of the directory tree, it is suggested that the tree structure be kept as simple as possible to keep navigation manageable.

Mstor Commands:

Commands are issued by calling the following methods of the Mstor object with parameters as described for each command. All methods return a value of 0 for success, or a 1-byte error. See Appendix 1 for error codes. Filenames as parameters can be preceded with path information such as "/dir1/dir2/filename.ext" (absolute) or "dir1/dir2/filename.ext" (relative). Any data that the command returns can be retrieved from Mstor.data[] with the number of bytes in Mstor.data_count (max =32 since that is the array size).

The MSTOR command list follows. Commands that are shaded In the table are those change the Mode, or are generally Mode-dependent, however there are exceptions. See the description of each command for details.

Mstor Driver Commands (Methods) List

Mstor Method (Type)	Command Description
fileOpen() **type – disk**	**result = fileOpen (char filemode, char * pathfile)** Must include two parameters. The first is 'R'ead, 'W'rite, or 'A'ppend and represents the mode the file will be opened in. The 2nd is the path/filename (path is optional). This method returns an unsigned char (result), On success result = 0. On fail a code from the error-code list is returned. Example: result = fileOpen('R', "dir1/dir2/filename") On success Mstor enters one of the following modes and will only execute commands appropriate to that mode. With Mstor in READ MODE it will execute fileRead() to read single records SEEK MODE it will execute fileRead() to read single records WRITE MODE it will execute fileWrite() to write single records APPEND MODE it will execute fileWrite() to write single records Only one file can be open at a time, and the host must issue the fileClose() command to exit this mode and return to COMMAND MODE. All file and directory names must confirm to 8P3 format. Result = 0 on success, or a short integer error code Can be executed in COMMAND MODE only.

fileClose() **type – disk**	**result = fileClose()** No parameters are required. Since only one file can be open at a time, this method closes the file that is open. Mstor returns to COMMAND MODE which allows it to execute all commands. Result = 0 on success, or a short integer error code Can be executed in any mode.
fileSeek() **type – disk**	**result = fileSeek(ulong offset, char * pathfile)** Requires two parameters, The first is the offset into the file to start subsequent fileRead() commands. The 2nd is the filename preceded with an optional path. It is similar to fileOpen() but opens at an offset instead of the start-of-file. On success Mstor enters SEEK MODE and will respond to fileRead() commands. Example: result = fileSeek(450000, "dir1/bigfile.txt"); Only one file can be open at a time and the host must issue fileClose() to exit this mode and return to COMMAND MODE. All file and directory names must confirm to 8P3 format. Result = 0 on success, or a short integer error code Can be executed in COMMAND MODE only.
fileRead() **type – disk** **Used for files opened in READ or SEEK modes**.	**Result = fileRead()** No parameters. This method operates on files in READ MODE or SEEK MODE. This method returns an unsigned char (result), On success result = 0. On fail a code from the error-code list is returned. Example: result = fileRead(); On success the Mstor.data[] array contains up to 32 bytes of returned data. Mstor.data_count is the number of bytes available. If the open file is read to its end, it will return zero data bytes indicating end-of-data. The host must issue fileClose() to terminate the mode and return to COMMAND MODE. [The host may issue fileClose() at any time – an opened file does not have to be read to its end.] result = 0 on success, or a short integer error code Can be executed in READ or SEEK MODES only.
fileWrite() **type – disk** **Used for files opened in**	**Result = fileWrite()** **--** No parameter overload and/or **result = fileWrite(char * wstr)** One string parameter overload These methods operates on files opened in WRITE MODE or APPEND MODE. They return an unsigned char (result), On success result = 0. On fail a code from the error-code list is returned.

WRITE or APPEND modes.	Result = fileWrite() writes up to 32 (binary) bytes previously placed in the Mstor.data[] array with the number of bytes in Mstor.data_count. Result = fileWrite("string to write") writes a (null-terminated) string of up to 45 characters. This version cannot be used with binary data since that can include embedded nulls which would be interpreted as end-of-string. Host must issue fileClose() to close the file and return to COMMAND MODE. Result = 0 on success, or a short integer error code Can be executed in WRITE or APPEND MODES only.
fileRename() **type-disk**	**result = fileRename(str filename, str newname) –** 2 parameters Two parameters are required .. the first is the existing filename which can be preceded by a path. The 2nd parameter is the new name (no path allowed). The file is renamed and remains in its original location. Result = 0 on success, or a short integer error code Can be executed in COMMAND MODE only.
fileDelete() **type-disk**	**result = fileDelete(str filename)…** 1 parameter The filename may be preceded with an optional path. This command deletes the specified file. Result = 0 on success, or a short integer error code Can be executed in COMMAND MODE only.
fileList() **type-disk**	**result = fileList();** no parameter overload **result = fileList(str mask);** One parameter overload With no parameter this method creates a list of all files in the current directory (equivalent to a mask of *.*). If a string is provided it can include an optional path preceding the mask. It returns an unsigned char (result), On success result = 0. On fail a code from the error-code list is returned. Example: result = fileList("dir1/dir1/*.txt"); On success Mstor enters FILELIST MODE and will respond to listRead() commands .. each of which returns one file-item. File items are ~ 30 text characters which are returned in Mstor.data[], the exact count is returned in Mstor.data_count.

	Although no file is opened this mode is exited by either reading the list until it returns a zero-data-length file item, or by issuing the closeFile() command which will return Mstor to COMMAND MODE. If a path is specified it must end with a valid file mask. All file and directory names must confirm to 8P3 format. Longer filenames will be truncated. Result = 0 on success, or a short integer error code Can be executed in COMMAND MODE only.
dirList() **type-disk**	**result = dirList()** no parameter overload **result = dirList(str mask)** One parameter overload With no parameter this method creates a list of all sub directories in the current directory (equivalent to a mask of *.*). If a string is provided it can include an optional path preceding the mask. It returns an unsigned char (result), On success result = 0. On fail a code from the error-code list is returned. Example: result = dirList("dir1/dir1/*.dir"); On success Mstor enters DIRLIST MODE and will respond to listRead() commands .. each of which returns one file-item. File items are ~ 30 text characters which are returned in Mstor.data[], the exact count is returned in Mstor.data_count. Although no file is opened this mode is exited by either reading the list until it returns a zero-data-length file item, or by issuing the closeFile() command which will return Mstor to COMMAND MODE. If a path is specified it must end with a valid file mask. All file and directory names must confirm to 8P3 format. Result = 0 on success, or a short integer error code Can be executed in COMMAND MODE only.

listRead() **type – disk**	**result = listRead()** no parameters This method returns one file or directory item (up to 32 text characters) in the Mstor.data[] array with the count in Mstor.data_count. The format of a list item is: File/dir name [SP] Date [SP] Time{SP] size File/dir name is an 8P3 name [SP] is an ASCII space Date is MM/DD/YY Time is HH:MM:SSA/PM (12-hr format) Size is up to 4 characters using suffixes of k/M/G result = 0 on success, or a short integer error code Can be executed in FILELIST or DIRLIST MODES only.
getCurrentDir() **type-disk**	**result = getCurrentDir(unsigned char offset) --** 1 paramater Return up to 30 characters from the current directory string starting at the specified offset-position. The first call is made with offset = 0. If Mstor.data_count >= 30, another call is made with offset = 30 to retrieve more. When Mstor.data_count is < 30 the complete directory path has been returned. Assume a directory structure exists : /mydir1.doc/dir1/nextdir/**current.dir**/bottom.dir First call offset=0 returns to ----------^ 2nd call offset = 30 returns from -------^ to--^ The path can be referenced from the root by starting with a '/' character, ex: **/ mydir1.doc** or the path can be relative to the current directory by starting with a subdirectory name, ex: **bottom .** A single dot '.' Refers to the current directory, so an equivalent path is **. /bottom .** A double-dot '..' refers to one directory level up from the current directory. So if Mstor is in the nextdir directory, it can be moved to the root directory with: …. (two double-dots to move up 2 levels. Result = 0 on success, or a short integer error code Can be executed in COMMAND MODE only.

changeDir() **type-disk**	**result = changeDir(string dirstr) –** 1 parameter Change the current directory. The 'dirstr' may be a full path referenced from the root by starting with a '/' character, or may be a relative path referenced to the current directory by starting with a sub-directory name. The '..' double-dot syntax moves up one level from the current level, and the '.' Single-dot refers to the current level. result = 0 on success, or a short integer error code dirstr Examples: Assume a directory structure exists : /mydir1.doc/dir1/nextdir/current.dir/bottom.dir **"/"** sets directory to the root from any directory **"/mydir1.doc/dir1"** sets to dir1 sub-dir from any directory **"dir1"** sets directory to dir1 *only from* /mydir1.doc sub-dir **"…."** 4-dots moves up 2 levels so from /bottom.dir would move up to /mydir1.doc/dir1/nextdir result = 0 on success, or a short integer error code Can be executed in COMMAND MODE only.
fileCopy() **type-disk**	**result = fileCopy(str filename, str newfile)** Requires two string parameters. The first is the existing file and the 2nd is the new file to be created. (new file will be overwritten if it exists). **This is an Asynchronous Command.** This means that the result is returned very quickly, however the command continues to execute for as long as it takes copying about 8k bytes per second. Mstor is unavailable for new commands until this command completes. During execution, Mstor returns a series of asterisks, one for every 3.5k bytes of data moved, so ~2-per-second – these asterisks arrive as single ASCII characters (no data framing) and can be read directly from the serial port – or ignored. When the copy is complete the series of asterisks is ended with one final ACK character. Result = 0 on success, or a short integer error code Can be executed in COMMAND MODE only.

fileConcat() **type-disk**	**result = fileConcat(str file1, str file2, str target) ..** Requires 3 string parameters. The 1st file to combine, the 2nd file to combine, and the target file which will hold the concatenation of these files. If the target file exists it will be overwritten. Each filename may include an optional path, however the sum of the lengths of all the strings must be <= 45 characters. **This is an Asynchronous Command.** This means that the result is returned very quickly, however the command continues to execute for as long as it takes copying about 8k bytes per second. Mstor is unavailable for new commands until this command completes. During execution, Mstor returns a series of asterisks, one for every 3.5k bytes of data moved, so ~ 2-per-second – these asterisks arrive as single ASCII characters (no data framing) and can be read directly from the serial port – or ignored. When the copy is complete the series of asterisks is ended with one final ACK character. Result = 0 on success, or a short integer error code Can be executed in COMMAND MODE only.
exitMode() **type-disk**	**result = exitMode()** .. no parameters This command can be used to terminate ANY mode and return to COMMAND MODE. If a file is open and Mstor is in READ/WRITE/APPEND/SEEK modes, this will close any file currently open, then return to COMMAND MODE which is identical to the fileClose() command. However it also terminates the FILELIST, DIRLIST, COPY and CONCAT modes, closing any open file, and returning to COMMAND MODE. This is the only safe way to prematurely teriminate the asynchronous commands fileCopy() and fileConcat() once they are started result = 0 on success, or a short integer error code Can be executed in any mode.
getFileSize() **type-disk**	**result = getFileSize(str filename) ..** one parameter The single parameter is a filename preceded with an optional path. Mstor returns series of ASCII decimal digits representing the full number of bytes in the file with no suffixes result = 0 on success, or a short integer error code Can be executed in COMMAND MODE only.

makeDir() **type-disk**	**result = makeDir(str dirname) –** 1 parameter Creates a new sub-directory under the current directory with the specified name. Name must confirm to 8.3 format – and cannot contain unusual characters such as period, comma, backslash, parenthesis, asterisk etc – those that typical operating systems do not allow. Result = 0 on success, or a short integer error code Can be executed in COMMAND MODE only.
deleteDir() **type-disk**	**result = deleteDir(str dirname) …** 1 parameter Deletes the specified directory. Can include a path to the directory, or just the directory name if it resides in the current directory. Result = 0 on success, or a short integer error code Can be executed in COMMAND MODE only.
testBattery() **type-nondisk**	**result = testBattery()** -- no parameters Applies a load to the internal battery to check its readiness to provide power-fail protection. Returns 0 on success (good battery) or an error code if the battery is unable to provide sufficient power to provide protection.as discussed in Reliability Features below. Although not a disk command, it can be executed in COMMAND MODE only.
getMode() **type-nondisk**	**result = getMode() – no parameters** This command can be issued at any time except during fileCopy() and fileConcat() to report the current Mstor mode. It can be used to identify if Mstor was left if an unexpected mode. Result = 0 on success, or a short integer error code Can be executed in any mode.
setBaud() **type-nondisk**	**Result = setBaud(byte rate) ..** one parameter This command takes one parameter (0 – 8) representing the baud rate to change to: 0 = 1200, 1 = 2400, 2 = 4800, 3 = 9600, 4 = 19.2k 5 = 38.4k *, 6 = 57.6k, 7 = 115.2k ,8 = 250k result = 0 on success, or a short integer error code Can be executed in COMMAND MODE only.

setIdleTimeout() **type-nondisk**	**result = setIdleTimeout(byte timeout) –** 1 parameter, 0 to 60 minutes . Default = 5 . Setting a value of 0 disables the feature. Idle Timeout is the value (in minutes) used by the Idle Timer. If Mstor is in a mode other than COMMAND MODE, and is idle (no disk access has occurred, and no messages have been received) for more than this time, Mstor closes any open file and returns itself to COMMAND MODE . No messages are issued if a timeout occurs. Result = 0 on success, or a short integer error code Can be executed in any mode.
getCmdTimeout() **type-nondisk**	**result = getCmdTimeout() -** no parameters Returns the current command timeout in milliseconds as a series of ASCII digits. The data is returned in Mstor.data[] with the number of bytes in Mstor.data_count. Result = 0 on success, or a short integer error code Can be executed in any mode.
setCmdTimeout() **type-nondisk**	**result = setCmdTimeout(str timeout) -** 1 parameter Sets the command timeout in milliseconds. The timeout value is a string of ASCII digits ranging from “250” to “10000” representing 250-1000ms .Default = timeout is 1000ms. Result = 0 on success, or a short integer error code Can be executed in any mode.
getTime() **type-nondisk**	**result = getTime() –** no parameter returns current time (in 12 Hr format) Time value is formatted as a series of bytes HH:MM:SSxM (x=’A’ or ‘P’) in the Mstor.data[] array with the number of bytes in Mstor.data_count result = 0 on success, or a short integer error code Can be executed in any mode.
setTime() **type-nondisk**	**result = setTime(str newtime) –** 1 parameter sets current time (in 12 Hr format) newtime value formatted as “HH:MM:SSxM” (x=’A’ or ‘P’) result = 0 on success, or a short integer error code Can be executed in any mode.

getDate() **type-nondisk**	**result = getDate() --** no parameter returns current date formatted as YYYY-MM-DD as a series of bytes in the Mstor.data[] array with the number of bytes in Mstor.data_count result = 0 on success, or a short integer error code Can be executed in any mode.
setDate() **type-nondisk**	**result = setDate(str newdate) --** 1 parameter sets a new date with newdate formatted as "YYYY-MM-DD" result = 0 on success, or a short integer error code Can be executed in any mode.
getClkCal() **type-nondisk**	**result = getClkCal() –** no parameters Retrieves the current clock calibration value as a signed char type (-127 to 127). This is the calibration adjustment that is applied to the real-time clock to speed it up (positive values) or slow it (negative values) with 0 being the no-adjustment value. The returned value can be read from Mstor.data[] as a 1-byte signed char. Mstor.data_count = 1. Allows for calibration of the real-time clock. Accepts from -127 to +127). Positive numbers make the clock run faster. Each increment increases the clock's speed by 2ppm (5sec per month). The value is saved in non-volatile memory. Result = 0 on success, or a short integer error code Can be executed in any mode.
setClkCal() **type-nondisk**	**result = setClkCal(char cal_value) –** 1 parameters Allows for calibration of the real-time clock. Accepts a value from -127 to +127). Positive numbers make the clock run faster. Each increment increases the clock's speed by 2ppm (5sec per month). The value is saved in non-volatile memory. The clock continues to run when power is removed from Mstor (sleeping). Result = 0 on success, or a short integer error code Can be executed in any mode.

getVersion() **type-nondisk**	**result = getVersion() –** no parameters Mstor returns its firmware version as a series of characters. Data returns in the Mstor.data[] array with the number of bytes in Mstor.data_count .. example MSTORv1.05a result = 0 on success, or a short integer error code Can be executed in any mode.
getPower() **type-disk**	**result = getPower()…** no parameters Returns the current power state of the Mstor disk as two characters dd: dd = N- (USB power ON, Drive NOT ready) dd = NR (USB Power ON, Drive READY) dd = F- (USB power OFF – drive commands not available) Data returns in the Mstor.data[] array with the number of bytes in Mstor.data_count . result = 0 on success, or a short integer error code Can be executed in any mode.
setPower() **type-disk**	**result = setPower(char cpwr)…** 1 parameter Sets the power state of the Mstor disk. A single character is supplied as the parameter: 'N' = turn USB power ON (attached disk will enumerate) 'F' = turn USB Power OFF 'S' = put Mstor to SLEEP (turns off disk power, then sends Mstor into sleep state. Battery continues to charge if power is applied. If power is removed, Mstor draws < 5uA battery current. Applies to Mstor SHIELD model , not Mstor BASIC model. Result = 0 on success, or a short integer error code. When put to SLEEP, no result code is returned. Can be executed in any mode. While sleeping**,** Mstor no longer responds to commands. Its clock continues to run. If external power is removed the internal rechargeable battery will last 1-2 years. To return to operating mode 5VDC power must be applied and Mstor *must be reset* to be awakened. See Hardware Resets for further description.

mstorReset() **type-nondisk**	**mstorReset()** **--** no parameters This method returns nothing. It applies a momentary hard reset to Mstor which will wake it up if sleeping, or recover from any unexpected condition or states.

Mstor Hardware Resets:

Mstor requires a 'hard reset' to wake up from its low power SLEEP state. This reset can be applied using either of the two methods below. Mstor restarts operation in COMMAND MODE with all files closed, disk power OFF, and all other settings at their pre-power-down values. ***NOTE: The Reset Button should NOT be pressed while the RED LED is lit indicating that the USB drive is powered.*** Method #1 below provides a safer shutdown because it will not be generated until the USB connection is suspended (idle).

1) Apply a HIGH level to the Arduino signal designated as the Mstor reset. Jumpers J8-J12 allow one Arduino digital signal D8-D12 respectively to be assigned as the Mstor reset signal. Use the mstorReset() method above to reset Mstor. [Note that Mstor protects itself from being reset during USB transactions since that could corrupt disk formatting].

2) Press/ release the reset button (next the silvery battery).

Mstor Power States:

Mstor has 3 power-states in which it can operate:

1) **Fully Powered.** External 5VDC power is applied, the USB port and the connected drive is fully powered, the battery tests good, and it is ready to execute disk access or system configuration commands. Mstor's green LED flash rate* is slow indicating that the USB drive is initialized and ready. (Mstor requires approximately 35mA, and a USB drive typically requires an additional 30-60mA).

2) **USB Powered-Down.** This is the state that Mstor enters when awakened from sleep. Mstor is operational for non-disk access but the USB drive power is off (red LED is off), and power consumption is about 50% of full power. It is safe to plug/unplug a drive.. Mstor's green power LED flashes* very quickly indicating

that the USB drive is not detected. The setPower() command (see above) is available to apply power the drive and return to the fully powered state in the few seconds required to enumerate the drive (re-initialize USB communications).

3) **Sleep State (Mstor SHIELD model only).** Mstor is completely shut down except for the very low power real-time clock . Mstor will not respond to serial commands, no LEDs are lit and the USB drive is powered down. With external power applied, the battery will charge at 0.1-20mA depending on the battery state. Mstor requires a reset to exit sleep state and return to USB Powered Down state.

*If the battery level is detected as LOW, the LED flash rate changes to two quick flashes and a longer pause to indicate the battery condition.
.

Timekeeping and File TimeStamps:

Mstor provides a battery-backed real-time clock for time- stamping files and directories when created and updated. During manufacture and testing each Mstor internal clock is set to Pacific Standard Time. It is calibrated to +/- a few seconds per month and the internal battery will keep this clock active for 1 year or more in the sleep state. The battery recharges while power is applied. If powered down for an extended period and the battery discharges, the clock's date and time should be reset to maintain meaningful timestamps. (See setTime() and setDate() commands above) .

Reliability Features:

Mstor implements power-level monitoring:
For model Mstor SHIELD if the applied 5V power drops to 4.5V, Mstor aborts any disk operation in progress and enters sleep mode – everything shuts down except the real-time clock. During this transition the on-board battery and booster maintain 5V to the USB disk while an orderly shutdown is executed. When valid power returns Mstor can be reset to wake-up from sleep and resume normal operation. If a reset is applied and external power has not been restored, Mstor wakes momentarily, the quickly re-enters sleep mode. This safety feature protects the disk from formatting damage.
For model Mstor BASIC if the applied 5V power drops to 4.65V, Mstor aborts any disk operation in progress, but it is not guaranteed that the operation can be halted while the flash drive has sufficient power to operate correctly.

USB Power Considerations:

USB ports are sometimes used for battery chargers, lighting power sources or powering external rotating-disk USB drives. The Mstor USB port should ***not be used for any of these devices***. Typical solid-state USB 'thumb' drives require 30-60mA, and while

standard USB 2.0 ports can provide up to 500ma for device operation or charging, Mstor is limited to USB drives that require no more than 125ma.

Baud Rate Control and Configuration:

Mstor ships with its baud rate set to the factory default 38.4k.

Mstorcfg.h defines the baud rate for both the Mstor_Port and the Monitor_Port – the two serial ports which the Sample Arduino Application uses. With DEBUG defined in this file,

- Mstor_Port is connected to a SoftwareSerial port set to 38.4k baud (to match default Mstor). SoftwareSerial ports do not operate reliably above this rate. The default factory jumpers define D8 as TX, D9 as Rx, D12 as mstor-reset – see above, change I/O usage as needed.
- Monitor_Port is connected to a HardwareSerial port set to 115.2k baud to be used as the normal Arduino Serial Monitor port.
- Using this default configuration you can use the SerialMonitor for debugging and developing your own Mstor appications. The Arduino-Mstor-Template will get you started. If the 3.5 k byte/sec disk transfer rate is fast enough, this is the most convenient configuration to continue to use.

If you need faster disk transfer speeds after debugging is complete, you can alter the port configurations.

1) ***First, Mstor must be set to its new operating baud rate.*** Use the Arduino Sample Application to issue the "E8{CR} [3] command which sets Mstor's baud rate to 250k. The application will no longer talk to Mstor.
2) Undefine DEBUG by commenting that line in Mstorcfg.h -- the ports are now reversed: Mstor_Port is connected to the HardwareSerial port and set to 250k baud as defined by the .h file. This port is no longer available as a SerialMonitor port. If a SerialMonitor port is needed (Arduino Sample Application requires one) the monitor port is now defined as a SoftwareSerial port which must be provided by an RS232-shield connected to an RS232 terminal.
3) If the application does NOT use an RS232 Shield as a SerialMonitor, de-allocate D8 & D9 by removing the 'SoftwareSerial SSserial(RX_PIN, TX_PIN) ' statement from the Arduino Template Application.
4) Re-compile the Arduino application to set the new serial configurations in effect and download to the Arduino.
5) If your application needs a Monitor port, configure the RS232 shield to use D8 & D9 as TX, RX, and install the two JP5 jumpers horizontally as shown previously to connect D0 & D1 HardwareSerial signals to Mstor.
6) The Arduino is now configured to talk to Mstor at 250k baud, and Mstor was reconfigured in (1) to talk at 250k. If you installed an RS232 shield it is now the SerialMonitor port.
7) Don't forget that you are occupying the Arduino programming/Monitor port. If you have to reload the Arduino code, you must temporarily remove the two JP5 jumpers for programming to succeed, then re-install the jumpers for Mstor communications to resume.

Physical, Electrical, Environmental Specifications:

Physical:	
Size:	(2.7" x 2.1" x 0.85" not including USB drive
Weight:	2oz
USB Connector:	Standard Type-A
Electrical:	
Abs Max Voltage Input	5.5VDC
Max USB Disk Current	125mA
Nominal Voltage Input:	+5.0V +/ 0.25V
Input Current	*Full operating:* Mstor alone: 35mA *Full operating:* Mstor plus typical 16 gig disk: 80-100mA *Mstor Sleeping external power applied:* .1-20mA charging *Mstor Sleeping no external power:* < 5uA battery current
Internal Battery (Mstor SHIELD only)	Battery Type: 3.6V @ 60mAh LIPO Powers internal clock: minimum 1 year in sleep mode Recharge time from full discharge to operational: 1 hour Recharge time from full discharge to full charge: 15 hours
Serial Port Baud Rate:	1200 to 250k programmable
Input Signal Level HIGH Input Signal Level LOW	2.6V minimum, 5.25V maximum 0.0V minimum, 0.60V maximum
Battery Life Expectancy	3-5 years
Environmental:	
Max Operating Temp:	+70°C
Min Operating Temp:	10°C
Max Storage Temp:	+70°C
Min Storage Temp:	-10°C
Humidity:	Non-condensing at all temperatures

Appendix 1
Mstor Error Codes

NO_ERROR	0
BAD_COMMAND	1
FILE_NOT_FOUND	2
DIRECTORY_NOT_FOUND	3
FAILED_TO_OPEN_FILE	4
FAILED_TO_CLOSE_FILE	5
INVALID_PARAMETER	6
INVALID_COMMAND_LENGTH	7
FILE_WRITE_ERROR	8
DIR_PATH_ERROR	9
UNKNOWN_COMMAND_STATE	10
FAILED_TO_CREATE_DIRECTORY	11
FAILED_TO_DELETE	12
INVALID_COMMAND_MODE	13
TOO_MANY_CONSECUTIVE_RETRIES	14
INVALID_8P3NAME	15
WILDCARDS_NOT_ALLOWED	16
COMMAND_CHECK_ERROR	17
FAILED_TO_RENAME_FILE	18
FILE_READ_ERROR	19
STREAM_TIMEOUT	20
COMMAND_TOO_LONG	21
INVALID_DECIMAL_POSITION	22
INVALID_8P3_CHARACTER	23
FNAME_TOO_LONG	24
DISK_NOT_READY	25
INVALID_PACKET	26
GENERAL_ERROR	27
SOURCE_NOT_FOUND	28
USER_ABORT	29
COMMAND_TIMEOUT	30
INVALID_RETRY_REQUEST	31
UNRECOVERABLE_PACKET_FAIL	32
MSTOR_NOT_FOUND	33
POWER_FAILURE	34
BLOCK_SIZES_DO_NOT_MATCH	35
INVALID_TIMEOUT	36
INVALID_FILE_MODE	37
FILEISZE_ERROR	38
BAD_BATTERY	39
END_OF_ERRORS	40

Glossary of Unique Terms:

{ } ASCII non-printable control character control character codes such as {ACK}, {NAK}, {CR}

[1] Arduino has at least one built-in Hardware Serial port which can operate at high speed. The Uno has only 1 of these, other Arduino's have more.

[2] UPS -- Uninterruptable Power Supply, a battery-backed power delivery system that continues to provide power after the main power source has failed.

[3] Baud Rate settings: 'Ex{CR}' to set Mstor Baudrate

x = '0' (1200 baud)	x = '5' (38.4k baud)
x = '1' (2400 baud)	x = '6' (57.6k baud)
x = '2' (4800 baud)	x = '7' (115.2k baud)
x = '7' (115.2k baud)	x = '8' (250k baud)
x = '4' (19.2k baud)	

Document Revision Record:

Revision	Revision Date	Description
V1.00	Jan 7, 2023	Initial Release
V1.01	Feb 2,2023	Change default Reset signal to D10, grammar corrections
V1.02	Feb 8, 2023	Add High-Speed Connection Diagram
V1.03	Feb 12, 2023	Add description of reduced function for Mstor BASIC, Add Baud Rate Recovery description
V1.04	Feb 20, 2023	Reverse RX/TX labels on J8-J12 jumper blocks