|  |
| --- |
| The Samraksh Company |
| Flash Based File-system |
| Design Document |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | **Modification by** | **Comment** | **Date** | **Review by** | **Comment** | **Date** | | Chethan and Ananth | Document creation | 2/27/12 |  |  |  | |  |  |  |  |  |  | |  |  |  |  |  |  | |

Contents

[1. File System: 3](#_Toc324756659)

[2. Log-based File System: 3](#_Toc324756660)

[3. Project statement: 3](#_Toc324756661)

[4. Project goals: 3](#_Toc324756662)

[5. Project milestones: 3](#_Toc324756663)

[6. Glossary: 3](#_Toc324756664)

[7. Technical Design: 4](#_Toc324756665)

[DataStore.h 4](#_Toc324756666)

[Address\_Table.h/Address\_Table.cpp 8](#_Toc324756667)

[Datastore\_int.h/datastore\_util.cpp 9](#_Toc324756668)

[Datastore\_reg.h/datastore\_reg.cpp 9](#_Toc324756669)

[d. Infrastructure: 16](#_Toc324756670)

[8. Performance Metrics: 16](#_Toc324756671)

[9. Project status: 16](#_Toc324756672)

[10. References: 16](#_Toc324756673)

### File System:

File System is a data structure by means of which data is stored in a storage media such that it is easy to perform operations (read, write, append, delete) on it. It also is a means of managing the storage media space intelligently such that the life-time of the storage device is extended as much as possible.

### Log-based File System:

Flash based file systems have criteria which may not apply to regular devices such as hard-disks. Some of them are:

1. Avoiding frequent writes to the same location so as to prevent damages to the device.
2. Efficient use of energy while performing memory operations.
3. Maintaining a contiguous set of data for easy and fast retrieval of information. This also helps in a fast erase operation of a block of data.

### Project statement:

To design and develop a log-based Flash File System for the eMote[[1]](#footnote-1) device.

### Project goals:

The goals of the project are as follows:

1. Design and implement a log-file system for a NOR based flash file system.
   1. The erase time of NOR flash (0.7 second) is 350 times longer than that of the NAND flash (2 ms). NOR is byte-addressable and is used in systems that need to store small amounts of data and a fast read operation.
2. Optimizing the read and write cycles so as to enhance the life-time of the Flash device.
   1. To allow out-of-place updates, which means that before data is written to a unit, the existing data either needs to be erased or written to a different block of memory.
   2. The size of the erase block is much larger than that of the write block. This could potentially mean that live data could be written next to a block earmarked for erasure. Therefore the garbage collection operation needs to be invoked which maintains the erase and write blocks in separate areas of memory.
   3. The typical life-time of a flash device is about 100,000 erase-cycles [3]. Therefore, to increase the life-time, this needs to be kept as minimal as possible.
3. Determining the usage of the end system and optimize the various operations of the file system accordingly.
4. Determining the optimum method for the migration of data from one block to another so that data is contiguous.
5. Implementing the file system on the Flash and benchmarking the performance against other similar file systems [4].

### Project milestones:

### Glossary:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | Log point |  | |  |  | | Indicates the current end of the log. The next set of data can be written to memory starting from this location. |
| |  |  | | --- | --- | | Erase point |  | | Indicates the last point that was erased. |
| |  |  | | --- | --- | | Clean point |  | | Indicates the point until which the log could be extended into the next erase block with new data without overriding the existing data. |
| |  |  | | --- | --- | | Scan point |  | | Used before executing the erase operation of a block. It is moved across an area of memory to check for the existence of active records. |
| |  |  | | --- | --- | | Erase block |  | | A contiguous section of memory that is ready to be erased in the erase cycle. |
| |  |  | | --- | --- | | Log headroom |  | | Space available in the log for writing new data without erasing the block of memory. |
| |  |  | | --- | --- | | Erasable area |  | | Represents the space in the next block of memory that can be erased to add extra space to the log headroom. |
| |  | | --- | | Hot Logs | | “Hot” logs refer to those areas of the memory which is allocated for frequent accesses and hence are volatile in nature. |
| |  |  | | --- | --- | | Cold logs |  | | “Cold” logs provide long-term storage and hence are relatively-nonvolatile. |

### Technical Design:

##### Design overview:

##### Technical architecture:

###### DataStore APIs:

**Enums and Macros:**

|  |  |
| --- | --- |
| DATASTORE\_CLUSTER | uint32 blockSize |
| uint32 numBlocks |
| DATASTORE\_PROPERTIES | DATASTORE\_CLUSTER \*clusters |
| Int numClusters |
| LPVOID addressRangeStart |
| LPVOID addressRangeEnd |
| DATASTORE\_STATUS | DATASTORE\_STATUS\_OK |
| DATASTORE\_STATUS\_NOT\_OK |
| DATASTORE\_STATUS\_INVALID\_PARAM |
| DATASTORE\_STATUS\_INT\_ERROR |
| DATASTORE\_STATUS\_OUT\_OF\_MEM |
| DATASTORE\_STATUS\_OVERLAPPING\_ADDRESS\_SPACE |
| DATASTORE\_STATUS\_NOT\_FOUND |
| DATASTORE\_STATUS\_DUPLICATE\_RECORD\_ID |
| DATASTORE\_STATE | DATASTORE\_STATE\_UNINIT |
| DATASTORE\_STATE\_READY |
| DATASTORE\_STATE\_INT\_ERROR |
| RECORD\_HEADER | Uint32 zero (1 bit) |
| Uint32 version (2 bits) |
| Uint32 activeFlag (1 bit) |
| Uint32 size (28 bits) |
| Uint32 recordID |
| Void\* nextLink |
| DATASTORE\_REG\_ENTRY | LPVOID dataStoreHandle |
| LPVOID addrRangeStart |
| LPVOID addrRangeEnd |

###### #define MIN\_SPACE\_REQUIRED\_FOR\_COMPACTION (10\*1024)

#define DATA\_STORE\_OFFSET (0)

### DataStore.h

###### Private functions:

*DATASTORE\_STATUS initDataStore(char \*datastoreName, DATASTORE\_PROPERTIES \*property )*

**Tasks performed:**

1. Creates and initializes the flash emulator structures.
2. Initializes the variables that point to the beginning of the dataStore (lDataStoreStartBlockID, lDataStoreStartPtr, dataStoreStartByteOffset, dataStoreEndByteOffset)
3. Registers the data store with the registration table which is used for address translations.
4. Scans the flash device to get the log, erase and clean pointers.

*DATASTORE\_STATUS compactLog()*

**Tasks performed:**

1. Makes free space by moving around active records and reclaiming the space taken by inactive records.

*Void incrementLogPointer(uint32 amount)*

**Tasks performed:**

1. Helper function to increment log-pointer cyclically.
2. This helps ensure that the log pointer stays within the range of dataStoreStartByteOffset and dataStoreEndByteOffset.

*Void incrementErasePoint(int amount)*

**Tasks performed:**

1. Helper function to increment erase-point cyclically.
2. This helps ensure that the erase pointer stays within the range of dataStoreStartByteOffset and dataStoreEndByteOffset.

*Void incrementClearPoint(uint32 amount)*

**Tasks performed:**

1. Helper function to increment erase-point cyclically.
2. This helps ensure that the clear pointer stays within the range of dataStoreStartByteOffset and dataStoreEndByteOffset.

*LPVOID incrementPointer(LPVOID inputPtr, int amount)*

**Tasks performed:**

1. Helper function to increment pointers so that it doesn't run out of circular buffer.
2. This helps ensure that the pointer stays within the range of dataStoreStartByteOffset and dataStoreEndByteOffset.

*int calculateNumBytes(LPVOID fromAddr, LPVOID toAddr)*

**Tasks performed:**

1. Calculates the number of bytes between the given two locations in memory.
2. If fromAddr is less than toAddr, return the difference between the two (toAddr – fromAddr)
3. If fromAddr is greater than toAddr, calculate the sum of the below two values:
   1. Difference between end address of device and fromAddr
   2. Difference between toAddr and start address of device.

*uint32 calculateLogHeadRoom()*

**Tasks performed:**

1. Helper function to calculate the amount of free space available for immediate writing also called Log-head room.
2. If erasePoint offset is greater than the logPoint offset, return the difference between erasePoint and logPoint.
3. If erasePoint offset is less than the logPoint offset, return the sum of the below two values:
   1. Difference between the end of the dataStore and logPoint offset.
   2. Difference between the erasePoint offset and start of dataStore.

*LPVOID createAllocation( RECORD\_ID recordID, LPVOID givenPtr, uint32 numBytes)*

**Tasks performed:**

1. Create new record in the flash memory and gives out a virtual pointer/identifier for the same.
2. Check if logHeadRoom is less than the minimum space required for compaction. If yes, then free up some space by moving active records and clearing the inactive records.
3. If even after compaction, there is insufficient space, then there is insufficient space in flash.
4. After creating sufficient space, mark the old record as inactive and add the new entry to the address table. If it is a new entry, just add it directly to the address table.

*bool detectOverWrite( void \*addr, void \*data, int dataLen, uint32\* conflictStartLoc )*

**Tasks performed:**

1. Check if data has been over-written on the same flash location. The over-write is detected by logically ANDing the data being verified with that in the flash memory location.
2. If over-written, return TRUE.

*uint32 cyclicDataRead( LPVOID buff, LPVOID currentLoc, uint32 numBytes )*

**Tasks performed:**

1. Reads record header from the flash handling cyclic copy.
2. This function returns an error when trying to read from out of bound regions in memory.
3. Calculate the offset by subtracting the base address of the device from the current location.
4. If the sum of the offset and count of bytes to be read is less than the end of the data store, then copy the contents to a buffer and return the count of the bytes read.
5. If the sum of the offset and the count of bytes to be read is greater than the end of the data store, then perform the below operations:
   1. Calculate the number of bytes till the end, by subtracting the offset from the end of data store. Read this value into the buffer.
   2. Calculate the rest of the bytes to be read, but start reading from the beginning of the data store, as we have moved beyond the end of the data store.

*uint32 cyclicDataWrite( LPVOID buff, LPVOID currentLoc, uint32 numBytes )*

**Tasks performed:**

1. Writes record header from the flash handling cyclic copy.
2. Operation is exactly the same as cyclicDataRead.

*uint32 decideWhichOneisRecent(RECORD\_HEADER tempHeader, RECORD\_HEADER header)*

**Tasks performed:**

1. Helper function for the scanFlashDevice().
2. Returns the latest version of the two arguments passed to it.

*LPVOID scanFlashDevice()*

**Tasks performed:**

1. Scan the flash device and load all the records in the Flash Device. Used at bootup to load all the records in the flash.
2. The Flash Device stores the locations of all records written in the data store in 2 hidden sectors, which are defined right before the start of the data store.
3. Allocate the hiddenSectorPointer to the beginning of the first hidden sector. If the hiddenSector is all FFs (invalid data), then jump to the next hidden sector. Else, keep incrementing the hidden sector pointer until you reach a location which has valid data. Assign this value as the ClearLogPointByteOffset. Do similar operation in the second hidden sector if the first is invalid.
4. Next perform a scan on the data store to check for valid records. Do this based on the value obtained from the hidden sector.
5. When a valid sector is found, add that location to the addressTable. If the table already contains a similar record, check which one is recent and mark the older one as inactive.
6. Set the values for the logPointByteOffset, clearLogPointByOffset and erasePointByteOffset.

###### Public functions:

*Data\_Store( char \*flashDeviceName )*

**Tasks performed:**

1. Constructor that initializes the data store with the default settings.

*Data\_Store( char \*flashDeviceName, DATASTORE\_PROPERTIES \*property )*

**Tasks performed:**

1. Create a Datastore with the property passed as the argument - Only used with emulator.

*LPVOID createRecord( RECORD\_ID id, uint32 numBytes )*

**Tasks performed:**

1. Create a record in the data store.
2. Check if the recordID already exists in the address table or not. If it does not exist, then generate a unique pointer and register it in the address table by calling the function createAllocation.

*DATASTORE\_STATUS deleteRecord(RECORD\_ID id)*

**Tasks performed:**

1. Deletes a previously created record.
2. Get the current location of the recordID and mark its header as inactive. This will then be removed by the garbage collector.
3. Remove the recordID from the address table.

*LPVOID getAddress(RECORD\_ID id)*

**Tasks performed:**

1. Get base address for given Record\_id.
2. Get the pointer to the recordID from the address table and return it to the calling function.

*RECORD\_ID getRecordID(LPVOID givenPtr)*

**Tasks performed:**

1. Lookup the recordID in the address table corresponding to the virtual pointer previously given while creating a record.

*uint32 writeData(LPVOID dest, T \*data, int count)*

**Tasks performed:**

1. Write data to the store by calling the writeRawData function. Same as writeRawData but takes arguments in terms of words.

*uint32 writeRawData(LPVOID dest, void\* data, uint32 numBytes)*

**Tasks performed:**

1. Write raw data - Can be of any word-size.
2. It checks for overwrites and moves records in the flash if required.

*uint32 readData(LPVOID src, T\* data, uint32 count)*

**Tasks performed:**

1. Read data from the store by calling the readRawData function. Same as readRawData but takes arguments in terms of number of words.

*uint32 readRawData(LPVOID src, void \*data, uint32 numBytes)*

**Tasks performed:**

1. Reads the record stored in the record corresponding to the address passed (src).
2. User may offset the virtual pointer with in the record size to read the data from corresponding location.

*DATASTORE\_ERROR getLastError()*

**Tasks performed:**

1. Returns the error code of any error in the previous call.

### Address\_Table.h/Address\_Table.cpp

**Class *DATASTORE\_AddrTable***

###### Private functions:

*DATASTORE\_STATUS search (LPVOID givenAddr, DATASTORE\_ADDR\_TBL\_ENTRY \*entry)*

1. Searches for given address in the address table using binary search.
2. After the entry is found, return “DATASTORE\_STATUS\_OK” and the location of the entry found in the address table.

###### Public functions:

*DATASTORE\_AddrTable()*

1. Constructor for class *DATASTORE\_AddrTable.*

*DATASTORE\_STATUS addEntry(DATASTORE\_ADDR\_TBL\_ENTRY \*entry)*

1. Adds given entry into the address table for future lookup making sure that its not duplicated.
2. If duplicate enty found in the table, call fails with status DATASTORE\_STATUS\_RECORD\_ALREADY\_EXISTS.

*LPVOID getCurrentLoc(RECORD\_ID recordID)*

1. Return the current location of the recordID in the addressTable.

*LPVOID getCurrentLoc(LPVOID givenPtr, LPVOID startPtr, LPVOID endPtr)*

1. Return the location of the givenPtr between the start and the endPtr.

*LPVOID getGivenAddress(RECORD\_ID recordID)*

1. Return the address of the recordID from the addressTable.

*RECORD\_ID getRecordID(LPVOID givenAddr)*

1. Search for the givenAddr from the addressTable and return the corresponding recordID.

*uint32 getAllocationSize(RECORD\_ID recordID)*

1. Return the allocation size for the recordID from the addressTable.

*uint32 getMaxWriteSize(LPVOID givenAddr)*

1. Returns the number of bytes that can be written from the given point.

*DATASTORE\_STATUS updateCurrentLocation(RECORD\_ID recordID, LPVOID newLoc)*

1. Search for the recordID in the addressTable and update its location to the “newLoc” argument.

*DATASTORE\_STATUS removeEntry(RECORD\_ID recordID)*

1. Search for the recordID and remove its entry from the addressTable.

*DATASTORE\_STATUS removeEntry(LPVOID givenPtr)*

1. Search for the givenPtr and remove its entry from the addressTable.

*int comparePtrRange(DATASTORE\_ADDR\_TBL\_ENTRY \*entry, LPVOID addr)*

1. Check if the “addr” variable is less than the givenPtr of the addressTable entry. If yes, return -1.
2. If the “addr” variable is greater than the givenPtr, return +1.
3. If the above two conditions fail, return 0.

### Datastore\_int.h/datastore\_util.cpp

**Class *uniquePtr***

**Public functions:**

*uniquePtr(LPVOID start, LPVOID end)*

1. Initializes the class with given address range.

*LPVOID getUniquePtr(uint32 numBytes)*

1. This function returns a unique pointer within the given range.

*int datastore\_abs(int val)*

1. Helper function to calculate absolute value of value being passed.

### Datastore\_reg.h/datastore\_reg.cpp

*DATASTORE\_STATUS DATASTORE\_REG\_CreateEntry(DATASTORE\_REG\_ENTRY \*entry)*

1. Check if there is an overlap in the dataStore. If there isn’t any, add the entry to the dataStore.

*LPVOID DATASTORE\_REG\_LookupDataStoreHandle(LPVOID addr)*

1. After checking for an overlap for the “addr” value passed as an argument, return the corresponding handle to the address.

*DATASTORE\_STATUS DATASTORE\_REG\_RemoveEntry(LPVOID addr)*

1. Remove the entry for the address from the dataStore.

*static bool DATASTORE\_REG\_checkOverLap(DATASTORE\_REG\_ENTRY \*entry)*

1. Check for any overlap in the address range.

###### Flash\_Emulator APIs:

**Enums and Macros:**

|  |  |
| --- | --- |
| EMULATOR\_STATUS | EMULATOR\_STATUS\_OK |
| EMULATOR\_STATUS\_NOT\_OK |
| BLOCK\_CLUSTER | int blockSize |
| int numBlocks |
| FLASH\_PROPERTIES | BLOCK\_CLUSTER \*clusters |
| Int numClusters |

#define EMULATOR\_DBG(messageStr) emulatorDebugLogs(\_\_FILE\_\_, \_\_LINE\_\_, \_\_FUNCTION\_\_, EMULATOR\_SEVERITY\_DEBUG, messageStr)

#define EMULATOR\_ASSERT( condition, messageString ) do{if(!(condition)){EMULATOR\_DBG(messageString);}}while(0);

###### Private functions:

*bool createNewFlashDeviceFile( char \*flashName, FLASH\_PROPERTIES \*properties )*

To create a new Flash device

*bool loadExistingFlashDevice ( char \*flashName, FLASH\_PROPERTIES \*properties )*

To load an existing flash device

*EMULATOR\_STATUS updateBlockStatistics(int blockID, FLASH\_BLOCK\_STATISTICS \*newVal)*

Updates the block statistics on each erase, write events on given block

*void initFlashEmulator( char \*flashName, FLASH\_PROPERTIES \*properties )*

Iinitializes everything for Constructor

*void buildConversionTbl( void )*

Function that build the conversion table that is used in the conversion between blockID and Address

*bool createMemoryMapping()*

Function that creates memory mapping of file, one that is exposed to user with READ-ONLY permission and internal mapping with READ-WRITE permission

###### Public functions:

*flash\_emulator( char \*flashName, FLASH\_PROPERTIES \*properties )*

Constructor - Opens an existing flash by name flashName or creates one with given FLASH\_PROPERTIES if it doesn't exist

*flash\_emulator<T>( char \*flashName, char \*configFileName )*

Constructor - Opens an existing flash by name flashName or creates one with given properties given in configFIleName

*int writeData( T \*dataIn, int dataCount, void\* addr )*

Writes data into flash at given offset.

Return Value: Number of bytes actually written. Return value ZERO means error

*int writeRawData( void \*dataIn, int dataSize, void\* addr )*

Writes any datatype to flash - Flash with WordSize=Int wants to write some string.

Return Value : Number of bytes actually written. Return value ZERO means error

*Flash\_Memory<T>\* getDeviceBaseAddress()*

Returns the base address of the device from which user can read the data

*int getBlockIDFromAddress(LPVOID address)*

Converts the given absolute address into the blockID

*Flash\_Memory<T>\* getBlockAddressFromBlockID(int blockID)*

Converts the given blockID into the block start address

*EMULATOR\_STATUS eraseBlock(int eraseBlockID)*

Erase a block given by eraseBlockID.

Return Value : Appropriate error status defined EMULATOR\_STATUS

*int getNumberofClusters()*

Get number of clusters in the device

*EMULATOR\_STATUS getDeviceProperties(FLASH\_PROPERTIES \*properties)*

Read the device properties - Especially useful when already existing flash image is opened.

Return value : Appropriate error status defined EMULATOR\_STATUS

*EMULATOR\_STATUS getBlockStatistic(int blockID, FLASH\_BLOCK\_STATISTICS \*statistic)*

Reads the statistics for given blockID.

Return value : Appropriate error status defined EMULATOR\_STATUS

*int getBlockSize( int blockID )*

Get block size of any block

*unsigned int getDeviceSize(){return flashSize;}*

Get device size

*EMULATOR\_STATE getDeviceState(){return state;}*

Gives the current state of the Emulator

*EMULATOR\_STATUS close()*

Close the device handle

*EMULATOR\_STATUS deleteDevice()*

Delete the emulator. Closes the device if it is open, and deletes the emulation file

##### Design Diagrams and Data Flow:

**General overview:**

Datastore uses the flash memory as a circular buffer i.e., it keeps writing data from the beginning till the end and when it reaches the end, it moves to the beginning again. For this implementation, the datastore assumes or defines few requirements as described below.

* 1. **Log head room:** Datastore requires to have a reserved amount for free space available all the time for it to operate. This space is crucial for garbage collection operation where the system moves the active data elements to the end of log and reclaims the inactive records.

This value is controlled by the compile time **MIN\_SPACE\_REQUIRED\_FOR\_COMPACTION**.

Minimum value recommended for this is, little more than maximum record size that is expected to be allocated at any time or a sector size, whichever is maximum.

* 1. **Hidden Sectors for book keeping:** Datastore requires two sectors before the beginning of datastore for book keeping. It stores the scan start point here, which will be used by the datastore to scan and discover all the data elements at the boot up. Two sectors are required for reliable persistence model explained later.
  2. **Three pointers:** Datastore internally maintains three pointers for its operation as explained below.
     + ***Log Point:*** The end of the log where next record will be written.
     + ***erasePoint:*** The Point till which erase is done and the space between Log Point and erase Point is the free space that is readily available for write. System always try to keep it as big as atleast **MIN\_SPACE\_REQUIRED\_FOR\_COMPACTION.**
     + ***clearPoint:*** Point from where the clear/compaction/garbage collection starts later. Points to the oldest entry in the flash. During compaction, clear-pointer moves from entry to entry and discards if its inactive, or moves to the end of log if its active.
  3. **Record Header:** Each record in the flash has a overhead of record header which is currently defined by the structure, accounting for 12 bytes currently.

typedef struct \_record\_header

{

uint32 zero:1;

uint32 version:2;

uint32 activeFlag:1;

uint32 size:28; /\* sizeof(version+isActive+size) =32bits \*/

uint32 recordID;

void\* nextLink;

}RECORD\_HEADER;

If we can add one more field to point to the previous data elements, hence having a logical doubly linked list, the number of erases in the hidden sector can be reduced by many folds. This is left for further discussions and future implementation.

* 1. **Address Table:**

Data store gives out virtual address to the application when it creates new records which remains same even when the physical records are moved in the flash changing the physical address. To translate between the virtual to current physical address, data store maintains a look-up table sorted on the virtual address **in the RAM**.

Currently, each entry takes 16 bytes, the structure is given below.

typedef struct

{

RECORD\_ID recordID;

LPVOID givenPtr;

DWORD allocationSize;

LPVOID currentLoc;

}DATASTORE\_ADDR\_TBL\_ENTRY;

<This can be reduced to 12 bytes with slight hit to performance in one of the API>

**Operations:**

***Boot-up scan:***

<This is to be completely implemented in the next version.>

When the system boots, data store as a step in initialization does a complete scan of the flash media to identify already stored records from the previous operation. To do this, the data store finds the latest entry of the start point from the hidden sector, starting from this point, it does the following scan operations.

* 1. Go through all the records starting from the start point till the end and populating the address table with all the discovered active records.
  2. Initialize logPointByteOffset,erasePointByteOffset,clearLogPointByOffset.

***Record Creation:***

The application calls data store to create new records using createRecord() providing a unique record id and size of the record. Data store will make sure that the record ids are unique, if not, the call fails.

New records are always added to the end of the log, before adding, data store checks if the head room has atleast minimum required amount of memory, if not it goes for garbage collection till it can make enough room in the flash. Once record is created, a unique virtual address (address range) is assigned to this record and returned to the caller. Also, the data store makes an entry for this record in the address table.

**Write Operation:**

When application requests to write data into a record, the data store makes sure to check bounds and will write only within the record size. If there is no overwrite is detected, the data can be written directly at the current location, if there is a overwrite detected with the previously written data, because of nature of flash memory, changes can’t be done inplace, in these situation, write will invalidate the current location, move the record to the end of log and updates the address table accordingly. Since the flash is used as a circular buffer, the write takes care of wrapping at the end of the flash.

If the write fails to migrate the record when overwrite is detected, the call will fail.

**Read Operation:**

When application requests for data from a record, the data store will use the address table for the translation of application’s virtual memory to the current location in flash and reads the data into the application provided buffer. Read does boundary checks and determines the number of bytes to be read.

**Delete Operation:**

When application requests to delete a record, data store will invalidate its entry on the flash, hence marking it for garbage collection later, and removes its entry from the address table. The freed memory because of delete will only be available after garbage collector reclaims this memory.

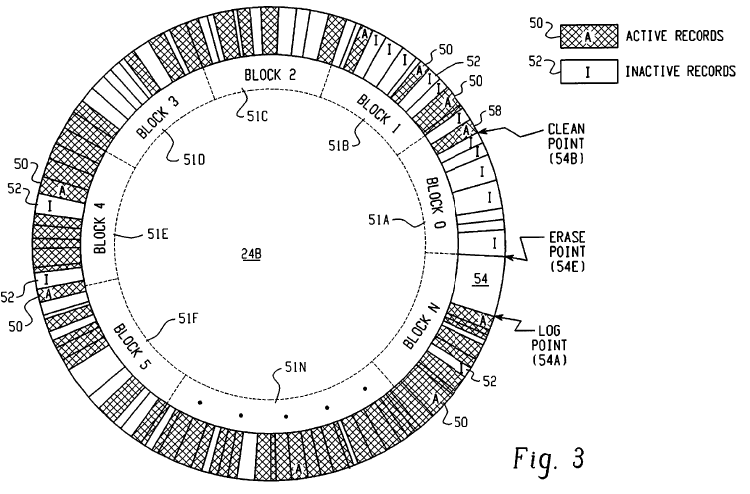
**Compaction of log/Garbage Collection:**

When the log head room (space immediately available for writing) goes below a threshold value, the system goes for Garbage collection. The system checks record pointed to by clearPtr and does one of the two things,

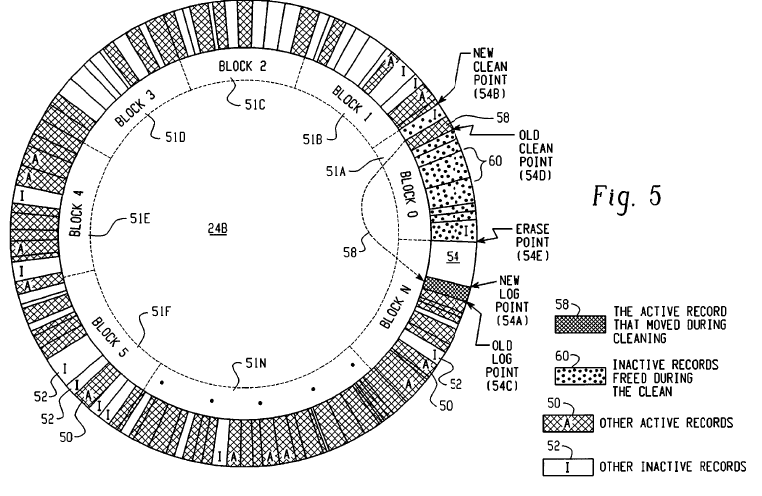
1. If the record is active, it is migrated to the end of the log.
2. If the record is inactive, it is skipped hence reclaiming the memory once the erase is done at the end.

After this, the clearPtr moves to the next record and the same process repeats till the clearPtr moves to the next flash sector/block leaving the previous one to be erased.

**Diagram showing flash as a circular buffer with logPointer, erasePointer and CleanPointer (From Patent)**

****

**Diagram showing flash during compaction, showing how active records are moved to end of log as the clearPoint scans through the flash (From Patent)**

****

### Infrastructure:

### Performance Metrics:

### Project status:

|  |  |  |  |
| --- | --- | --- | --- |
| **Date** | **Task** | **Status** | **Owner** |
| Feb 24, 2012 | Installation and setup of system | Complete | Mukundan, Chethan, Ananth. |
| Feb 24, 2012 | Project overview | Complete | Mukundan, Chethan, Ananth. |
|  | Design Document |  | Ananth, Chethan. |
|  |  |  |  |

### References:

1. Portable Electronic Device having a log-structured file system in Flash memory – Kenneth W.Parker – United States Patent No.: US 6,535,949 B1.
2. ELF: an efficient log-structured flash file system for micro sensor nodes - Hui Dai, Michael Neufeld, Richard Han; *Conference On Embedded Networked Sensor Systems - SenSys , pp. 176-187, 2004*.
3. Algorithms and Data Structures for Flash Memories - Eran Gal, Sivan Toledo; *ACM Computing Surveys - CSUR , vol. 37, no. 2, pp. 138-163, 2005.*
4. <https://en.wikipedia.org/wiki/Flash_file_system>.
5. <http://msdn.microsoft.com/en-us/library/ee435793.aspx>

1. eMote from Samraksh belongs to the wireless Sensor Networks family and features the .Net MicroFramework v4.0 on a 32-bit ARM microcontroller. [↑](#footnote-ref-1)