IMPLEMENT CIRCULAR BUFFER

1. Consider 2 pointers Head and Tail for Last Write Pointer and Last time Erase Pointer. Head Pointer has a precision of page and Tail Pointer has a precision of Block. While writing, head pointer is incremented from last page written to next writable block ‘s first page. When head meets tail, represents circular buffer as full. Tail pointer is incremented whenever there is a block erase.
2. A separate API to be formed to erase a prescribed number of blocks to minimize number of full erases. While erasing, get information of last page written from TOC block, and accordingly erase till block containing last page. Tail Pointer is incremented till the block which has been erased.
3. Two pages for storing Bad block information and Head, Tail pointers . Separate structures to store this information. Accordingly, 3 types of file information are stored in TOC: Configuration, Data and Table File. Total Configuration and Data files will be limited to 62.
4. While erasing memory, if bool is set to false, provision is to be made to store Table file as we need to remember the last time file write and file erase happened.
5. Another API is to be called to update whenever there is file erase along with file write.
6. To modify for more temp blocks for storing Table file whenever there is file erase is set to false and accordingly distinguish between odd / even as similar to Config block ? Accordingly File block starting index will change. Not necessary as same block can be used to back up.
7. Need to check Max file number from 64 to 62 ?
8. Implement TOC Block as Circular buffer?

Conditions check

1. What’s the file size limit ? Before writing file , if file size to write > = max capacity of flash , map it to max size of flash. Is this needed? Is it checked now.
2. Normally, each page in data has spare area which points out to Next page to write, but when Erased data is lost. This pointer can be continuous / discontinuous depending up on whether there are any intermediate Bad blocks.
3. In a loop, before writing initially both head and tail pointers are zero.
4. If file size to write <= Max flash capacity ,
5. Implemented -> Find first writable page ( this is got by check from TOC block first void page based on last page pointer and that page should not fall In bad block list. Increment to boundary of next block if page falls in between and is a bad block. This is done using Next page pointer.

Modification -> Read Head and Tail Pointers information from TOC. Head pointer will have information of last page written and is stored in TOC. Increment Head pointer to contain first page to write for current file. Head pointer is checked whether it falls in 0-4096\*64 pages else it is mapped to zero back and checked with Tail pointer if both coincide , then Head and Tail is checked whether both are zero , which indicates non erased memory or erased repeatedly and reset, in either case, memory is full. If Tail and Head pointer coincides, and not zero , provision given for writing only current block, then memory is full. If Head pointer doesn’t cross 4096\*64 pages , Current page is checked as to which block it is and accordingly, Head pointer is incremented to start writing from Next block.

1. Consequent pages to write

Implemented -> Here 2048 bytes buffer is filled prior to writing , and these are stored in buffer and written on to memory in multiples of page size. Before writing a page which is ( data + spare area which points out to next valid location), next page pointer where it should start writing the next page is checked ok or not ( Bad block check and void check).

Modification -> Here instead of Next page pointer head pointer is used for next valid loc. Same as described in section a.

This head pointer is copied in spare area as next valid page to search and accordingly page is written. This gives history of Head pointer required for file traversing in order to read back. This will solve problem of conversion of linear to circular buffer as history of Head pointers is stored in spare area , even if overflow in case of linear buffer we will get track of file reading for continuity.

After writing, if secure mode is used current page information for that file being written in TOC block is deleted and updated with new information of the same file and File handler for that particular file.

Remaining bytes are stored in temporary buffer which are not page multiples.

1. Implemented -> End of File. Here Next page in spare area is set to 0XFFFF to indicate discontinuity of file or end of file. Also, few buffered data less than page size is written and File handler is updated. Fast Mode file handler updated.

Modification -> When to update head pointer in TOC ? Like Fast Mode/ Secure Mode / Manual Mode.

Summary : Head pointer page writing of file and history of Head pointer is stored in each page , this is same way used for reading for continuity.

1. Erasing Data

To Implement API for prescribed number of blocks with Source Index = 0, and obtain from head pointer which has information about last page erase.

Calculate last page’s block index and successfully erase only those blocks.

Increment Tail pointer till last block which has been erased.

Implement an API to modify head pointer / Tail pointer in TOC.

1. While reading File,

Implemented:

Open File by name / Open File by number : this gives relevant file handler in TOC information where to read data / file size as depicted by file number.

If success, then read file in loop , next page pointer in current page will give continuity , how its written same way its read.

IMPLEMENT BAD BLOCK MANAGEMENT

1. Create an array of all block indexes of max 4096 and mark as ‘1’ for bad or ‘0’ for good. This will take 8 bit ( character size )\*4096 / 8 i.e. 4096 bytes to store list of bad block. Each page size is 2048 bytes …………… So we will dynamically create an array for bad block as no of bad blocks grows.?
2. Initially after production, a test case is run to calculate block indexes which are bad and accordingly marked and updated in structure.
3. If during page write / block erase there is a failure, whether any block has gone bad is checked and accordingly structure is updated which stores list of bad blocks in TOC.
4. Another API is to be called to update block list if any bad block is found.
5. No need to check every time in loop while writing whether the current block is bad / not. Those indexes which are marked bad are checked and skipped while writing default.

**List of API’s and Strcutures**

1. #define MAX\_NO\_OF\_WORDS 128 ( 4096/32) 32 bits in each word , each bit represents a block.

Typedef BlockFileHandler / TableFileHandler ( Structure for Table File )

{

Uint32\_t Head pointer;// in terms of page

Uint32\_t Tail Pointer;// in terms of block

Uint8\_t WriteOnlyCurBlk;// this bit is ‘1’ current block is also written

Uint32\_t BadBlockMarker[MAX\_NO\_OF\_WORDS];

} TableFileHandler;

1. Enum LFS\_FILE\_TYPE

{ LFS\_CONFIG\_FILE, LFS\_DATA\_FILE }

1. Enum LFS\_UPDATE\_TYPE

{ LFS\_BAD\_BLOCK\_MARKER\_UPDATE,LFS\_HEAD\_POINTER\_UPDATE,LFS\_TAIL\_POINTER\_UPDATE }

If there is a version mis match erase memory, after write of reserved block,

Call API 4. FS Version mismatch.

1. LFS\_RESULT InitializeCircularBuffer()

Call API 7 for reading Table File and Initialize Head and Tail pointers.

Initialize Head and Tail pointers.

Instead of zero, head is initialized to FILEBLOCK’s first page and Tail to FILEBLOCK.

1. LFS\_RESULT LFS\_PartialErase();

Call API 4 to read Tail / Head Pointers from TOC.

TOC block and Data Blocks are erased for those files as specified by Tail Pointer from TOC.

Here, also Tail Pointer to be updated and written after Erase. This will internally call API 4 for Tail pointer up dation in TOC.

Tail Pointer is calculated based on Head Pointer till where it is written.

Default : Erase starts from last Tail Pointer as source and Destination Till Head Pointer.

1. LFS\_RESULT LFS\_UpdateTableFileInTOC(TableFileHandler \* tablefilehandler , LFS\_UPDATE\_TYPE)

This will be called when any Bad block list update/Tail pointer / Head pointer update is to be done i.e. basically updating Table file in TOC.

This will internally call API 9 if bad block is found to write at a stretch Table file in TOC.

1. LFS\_RESULT LFS\_ReadTableFileInTOC(TableFileHandler \* tablefilehandler)

This will read Table file from TOC and update in defined structure as defined by parameter page.

1. LFS\_RESULT LFS\_GetValidHeadPointer(TableFileHandler \* tablefilehandler);

This handler will have Bad Block Indexes and Last written Head and Tail Pointers. Here, Tail and Head are independent pointers. Head Pointer is incremented to skip Bad blocks. Here, as prior implementation each page occupancy check is required since we are storing Last Head pointer using which page was written?

1. LFS\_RESULT LFS\_CheckCurrentBlockIsBad(uint32\_t BlockIndex , TableFileHandler \* tablefilehandler, ,bool\_t \*isBad)

This will check while traversing current block is bad or not. This is taken from TableFileHandler BadBlockMarker by parsing which word to access among 128 words and which bit to access in word. If particular bit is ‘1’ , block is bad else good.

1. LFS\_RESULT LFS\_CreateBadBlockList (TableFileHandler \* tablefilehandler)

This will create Bad block list at beginning by parsing all bad blocks and updating in words.

1. LFS\_RESULT LFS\_WriteTableFileInTOC(TableFileHandler \* tablefilehandler)

Write Table file contents

1. LFS\_RESULT LFS\_DeletePages(uint32\_t SrcPage,uint32\_t DstPage)

Delete Pages between Src and Dst Page Index including Src and Dst Page Index.