Flash Mapper 1.0

MegaLib

Project Code:
Project Name: MegaLib

Revision History			
Rev.	**Date**	**Author**	**Description**
1.00	2008-11-25	HH	Initial version.

Table of Contents

List of Figures

1 Introduction

The Flash Mapper is a mapper function between a 512-byte block based logical device and a physical memory driver. Depending on the memory type the mapper may be anything between a null pass-through function to a complex wear-levelling function.

This document presents how VLSI Solution's version 1.0 of the Flash Mapper works. This mapper is available in the ROM of the VS1000a through VS1000d chips.

The basic operations and logical structures of the mapper are presented in Chapter 2.

Chapter 3 presents mapper C structures.

Finally, VLSI Solution's contact information is provided in Chapter 4.

2 Flash Mapper Basics

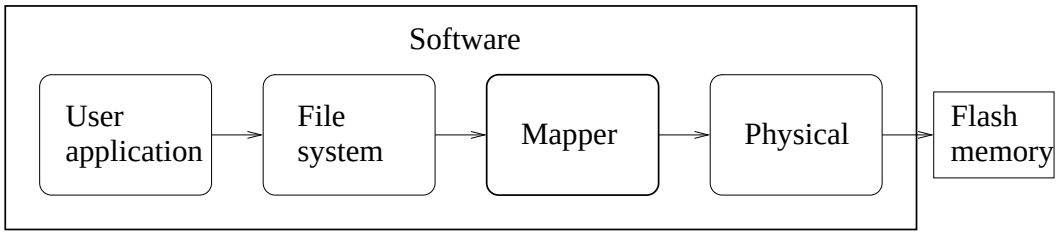

Figure 2.1: Device Chain from User Application to Flash Memory

The Flash Mapper is a software layer between the File system and Physical layers as shown in Figure 2.1. It hides the physical Flash Memory from the File System and performs automatic wear-levelling.

To make wear-levelling more efficient, the logical address as seen from the file system has been completely removed from the physical address that is used when writing a data block to the actual Flash Memory. Approximately 10 % of the available space of the Flash Memory is reserved for wear-levelling. If a read-only file system is to be implemented, less than 1 % of the file system is needed for bookkeeping, though.

2.1 Mapper Logical Structure

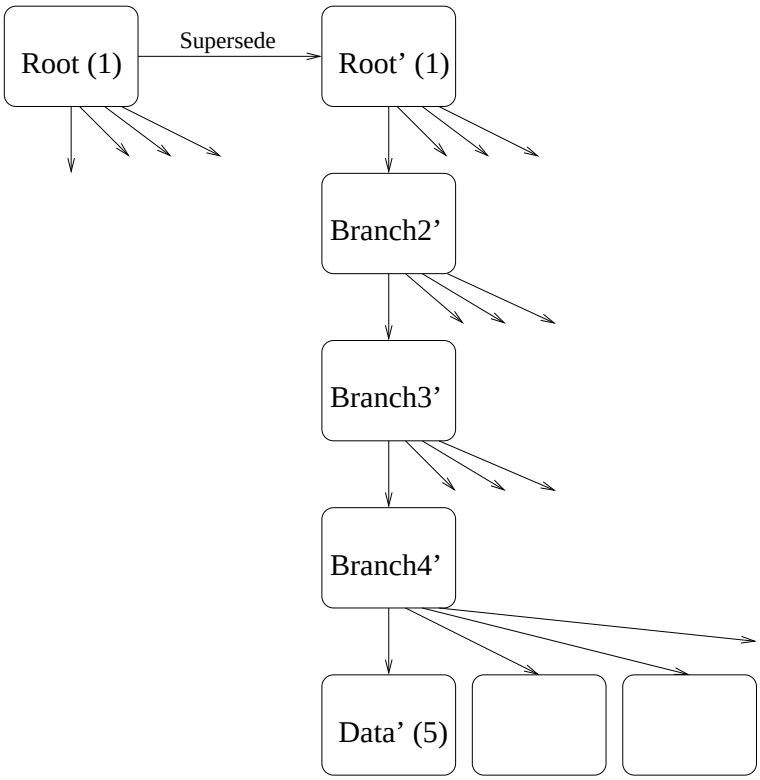

Figure 2.2: An Example Mapper B Tree

The Mapper has been implemented as a 5-level B Tree where each of the four uppermost node levels consist of 0...128 links to lower nodes. The lowest level is the actual data. All tree elements are 512 bytes. In addition to this, the 16-byte spare area of an underlying flash memory is used for bookkeeping functions.

Figure 2.2 shows the logical structure of an example mapper tree. It begins with a root node. To find a certain logical disk block, the tree is traversed to the node that contains the disk block data. The Root Node may be superseded in which case the superseding Node is used instead. In the example figure Root has been superseded with Root'.

2.2 A Mapper Node

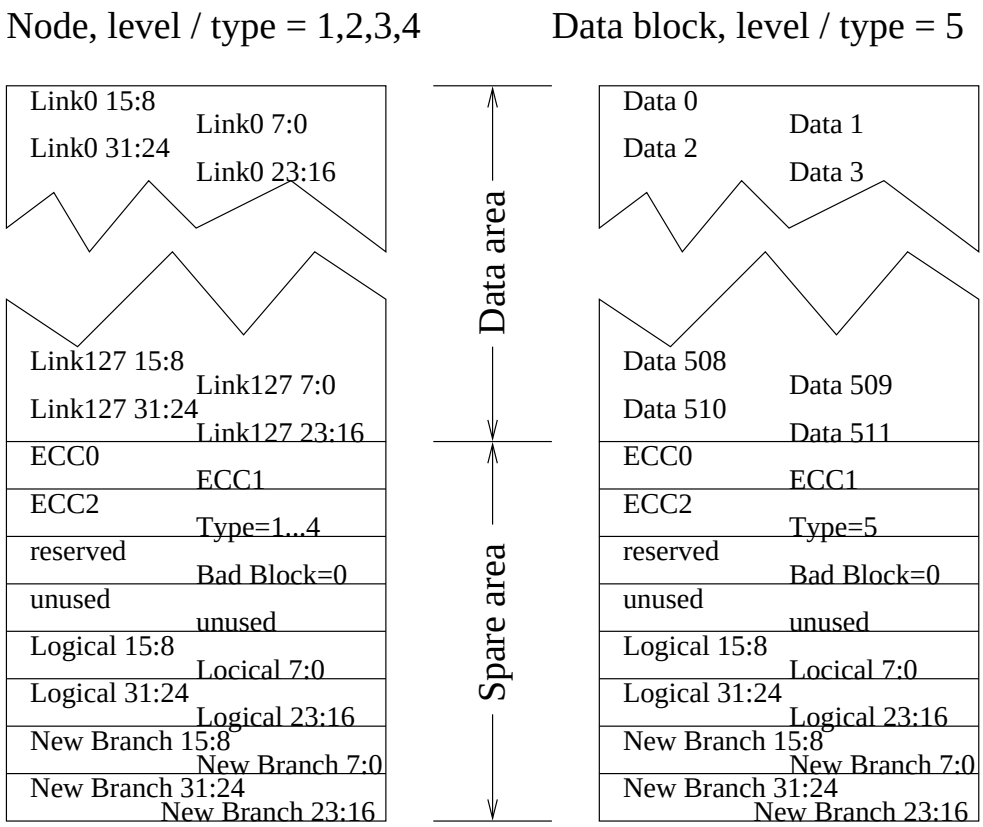

Figure 2.3: Two Kinds of a Mapper Node

The internal structure of a Mapper Node or data block is presented in Figure 2.3.

Link is a link to a next-level node.

Data is the data of a data block.

ECC0...2 are 24 error correction bits.

Type is the type, or rather the level of the branch. It is 1 for a Root Node, 2...4 for a Branch Node, 5 for a Data Node and 0xFF for an unused node. 0 is an error condition.

BadBlock should always be 0xFF. Anything else is an error condition.

If the node is a Root Node (Type = 1), Logical contains the last physical page that has been written to. The mapper uses this information to be able to continue wear-levelling where it was left off the last time. If the node is a Data Node (Type = 5), Logical is the logical block number. Contents of this field doesn't matter for other Node Types.

If New Branch is 0xFFFFFFFFU, then this node is a current node. If it is anything else, it is the physical address for the supersede node. However, the physical address should only be used for the Root Node.

2.3 Traversing the Mapper Tree

Let's assume we want to read a disk block, say logical disk block number 0x01234567. The validity of the address needs to be checked: it has to fit into 28 bits. The mapper splits the address needs in 7-bit chunks, as shown in Figure 2.4. These chunks (0x09, 0x0D, 0x0A and 0x67) will be later used as node indices.

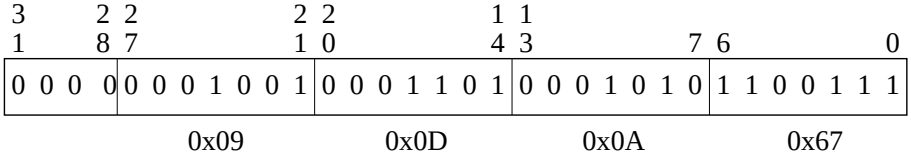

Figure 2.4: Splitting a 32-bit Disk Block Number

To traverse a Mapper Tree, the Root Node must first be found. To find it the following things need to be known:

- The Flash Mapper always skips the first Physical Erase Block. This area has been left for user applications that write directly to the memory without going through the file system.
- A copy of the Root Node is written to each of the first 10 Physical Erase Blocks. This will make it sure that a Root Node can be found even if wear-levelling has erased some Erase Blocks at the beginning of memory.
- When a Root Node has been found, it must be checked that the New Branch field is 0xFFFFFFFFU. If it is not, the field contains the physical address of a newer Root Node. If this is the case, the new Root Node is read and the New Branch field of the new Node is chacked until the latest Root Node has been found.

When the Root Node has been located, the B Tree is traversed. The 7 MSb's of the 28-bit Logical Block Number are used (in our example 0x09) to read the 32-bit LinkN from the data area (byte offset $4 \times n$ through $4 \times n + 3$). In the example, Link9 is read at bytes 0x09*4...0x09*4+3 = 36...39 and interpreted as shown in Figure 2.3. If the number reads 0xFFFFFFFFU, the Logical block has never been written to and thus doesn't physically exist. If the block is to be written to, it is created. If it is read from, the current Flash Mapper returns a block with all-zeros. If LinkN isn't 0xFFFFFFFFU, that physical page is read as the next-level Node.

The Branch Nodes are followed similarly to how the Root Node was, except that the mapper should never run to a superseded Node (example: Node number 0x0D is read from Branch 2, Node number 0x0A from Branch 3 and Node number 0x67 from Branch 4). This way the Mapper will end up with the physical page address for the actual data. The data is read from the physical page and returned to the caller.

2.4 Wear-Levelling Cleaning

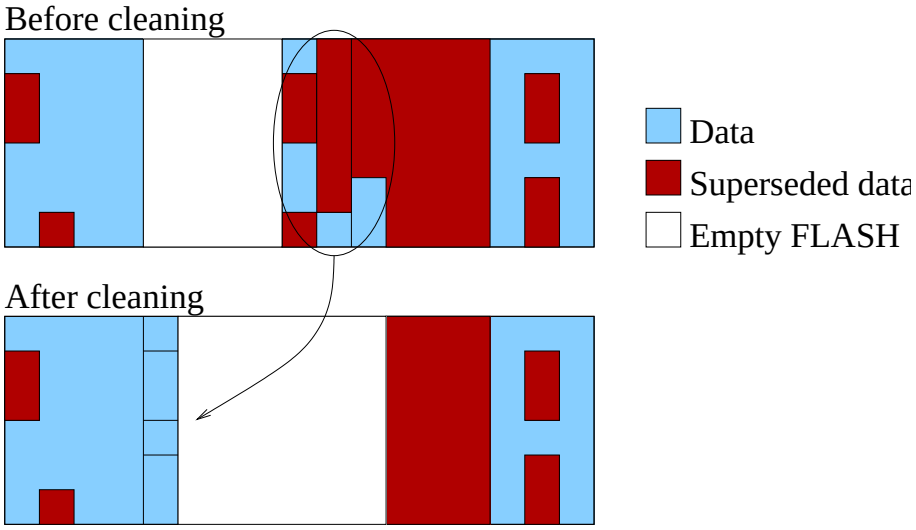

Figure 2.5: Wear-Levelling Cleaning

Regardless of what logical disk block addresses the file system writes into, the Mapper uses the Flash Memory in a cyclic manner. Wear-levelling is performed whenever the cycle has gone through the whole memory so that there would be a danger of running out of space.

Wear-Level Cleaning copies Nodes and Data Blocks from the erasable area, removes unused ones, and writes the ones in use to the current write area. This is continued until there exists enough of free space for the Mapper to operate. Depending on the fullness of the disk and how files are located on it, an automatic clean operation can take anything from a few milliseconds to several seconds.

If the power of a unit is cut or program execution is stopped in another way while performing a Wear-Level Cleaning operation, the Mapper system may end up in an inconsistent state.

2.5 Mapper Inconsistencies

Running into an non-existing Node (LinkN at any stage is 0xFFFFFFFF) is usually not an error. It only indicates that this Node hasn't been written to before. The Flash Mapper returns a block full of zeros in this case.

The following conditions all indicate a problem in the Mapper system:

- Root Node cannot be found in Physical Erase Blocks 1...10.
- Root Node New Branch points to a Node with different Type.
- Link points to a Node that doesn't have a Type one larger than the current Node.
- A Data Block doesn't have the correct Logical address (should be the same that was being looked for).
- When traversing a tree, you should never run into a superseded Node at any Node level.
- Orphan Nodes or Data Blocks exist. An Orphan Node or Data Block is a data page that doesn't have a New Branch (New Branch = 0xFFFFFFFFU) but no other valid Node points to it.

3 Mapper C Structures

This Chapter presents the Mapper C structures.

3.1 Physical Structure

Physical is the lowest layer that actually converses with a hardware device. Read and write operations are based on pages. An erase block may contain one or more pages. The erase block size should be a power of two.

```
struct FsPhysical {
  /** Version number. 8 MSBs contain version number, 8 LSBs size of
      the structure in words. */
  u_int16 version;
  /** In 16-bit words */
  u_int16 pageSize;
  /** In pages */
  u_int16 eraseBlockSize;
  /** The size of the memory unit in erasable blocks */
  u_int16 eraseBlocks;
  /** Creates a physical layer. param is a device-specific parameter,
      usually 0. */
  struct FsPhysical *(*Create)(u_int16 param);
  /** Delete a physical layer */
  s_int16 (*Delete)(struct FsPhysical *p);
  /** Read pages. meta is physical-specific data and not necessarily
      used. If either data or meta is NULL, that information is not
      returned. Setting both pointers to NULL is an error condition. */
  s_int16 (*Read)(struct FsPhysical *p, s_int32 firstPage, u_int16 pages,
  u_int16 *data, u_int16 *meta);
  /** Write pages. meta is physical-specific data and not necessarily
      used. If either data or meta is NULL, that information is not
      written. Setting both pointers to NULL is an error condition. */
  s_int16 (*Write)(struct FsPhysical *p, s_int32 firstPage, u_int16 pages,
   u_int16 *data, u_int16 *meta);
  /** Erase block. \e firsPage is the page number of the first page in the
      block. */
  s_int16 (*Erase)(struct FsPhysical *p, s_int32 page);
  /** Frees the bus for other devices */
  s_int16 (*FreeBus)(struct FsPhysical *p);
  /** Re-initializes bus after a fatal error (eg memory card removal) */
  s_int16 (*Reinitialize)(struct FsPhysical *p);
};
```

3.2 Generic Mapper Structure

The generic mapper structure is a structure common to all different mappers.

```
struct FsMapper {
  /** Version number. 8 MSBs contain version number, 8 LSBs size of
      the structure in words. */
  u_int16 version;
  /** How many 16-bit words in a block */
  u_int16 blockSize;
  /** How many usable blocks in the whole system */
  u_int32 blocks;
  /** How many blocks can be cached by the mapper */
  u_int16 cacheBlocks;
  /** Create a mapper. */
  struct FsMapper *(*Create)(struct FsPhysical *physical, u_int16 cacheSize);
  /** Delete a mapper */
  s_int16 (*Delete)(struct FsMapper *map);
  /** Read blocks */
  s_int16 (*Read)(struct FsMapper *map, u_int32 firstBlock, u_int16 blocks,
  u_int16 *data);
  /** Write blocks */
  s_int16 (*Write)(struct FsMapper *map, u_int32 firstBlock, u_int16 blocks,
   u_int16 *data);
  /** Free blocks (implementation must be able to go fastly through
      large free areas. */
  s_int16 (*Free)(struct FsMapper *map, u_int32 firstBlock, u_int32 blocks);
  /** Flush all data. if \e hard is non-zero, all potential journals are
also flushed. */
  s_int16 (*Flush)(struct FsMapper *map, u_int16 hard);
  /** Pointer to this Mapper's Physical layer. */
  struct FsPhysical *physical;
};
```

3.3 Flash Mapper Structure

These are the structures needed by the flash mapper.

```
/** Meta data */
struct FmfMeta {
  u_int16 ecc01;
  u_int16 ecc2AndType;
  u_int16 reservedAndBadBlock;
  u_int16 unused;
  u_int32 logicalPageNo; /* For root node, this is last used */
  s_int32 newBranch; /* Page # for root node, non- -1 for others */
};


#define FS_MAP_NON_FULL 4

/**
   A Flash Mapper specific structure that contains required
   extensions to the basic Mapper structure.
*/
struct FsMapperFlash {
  /** Public structure that is common to all mappers. */
  struct FsMapper m;
  /** Root node physical address. */
  u_int32 root;
  /** Logical blocks in erase unit. */
  s_int16 blocksPerErase;
  /** Last new physical address. */
  s_int32 lastUsed;
  /** Internal cache. */
  struct FmfCache *cache;
  /** Total of physical pages. */
  s_int32 physPages;
  /** Blocks that are not (almost) completely full with FMF_TYPE_DATA */
  s_int32 emptyBlock[FS_MAP_NON_FULL];
  /** How many pages in a block must be free for the block to be non-full */
  s_int16 nonFullLimit;
  /** How many blocks have been skipped while cleaning. */
  s_int32 skipped;
  /** How many blocks have been cleaned */
  s_int32 freed;
};
```

4 Contact Information

VLSI Solution Oy
Entrance G, 2nd floor
Hermiankatu 8
FIN-33720 Tampere
FINLAND

Fax: +358-3-3140-8288
Phone: +358-3-3140-8200
Email: sales@vlsi.fi
URL: http://www.vlsi.fi/

For technical questions or suggestions regarding this application, please contact support@vlsi.fi.