APPLICATION NOTE

Atmel AVR116: Wear Leveling on DataFlash

32-bit Atmel Microcontrollers

Features

- Wear leveling
  - Average the program/erase operations in different blocks
- Write not need be preceded by an erase operation
- Redirect logical address from host system to physical address in flash memory
- Power loss recovery
- Support FAT file system

Description

Flash memory has a limited program/erase cycle, program and erase in a same block many times will result in bad blocks and decrease the flash memory life cycle dramatically.

Flash memory is not fit for sector-based file systems (fat, etc.). Flash memory has several characteristics that make difficult straightforward replacement of magnetic disks. First, a write in flash memory should be preceded by an erase operation, which takes an order of magnitude longer than a write operation. Second, erase operations can only be performed in a much larger unit than the write operation. This implies that, for an update of even a single byte, an erase operation as well as restoration of a large amount of data would be required. This not only degrades the potential performance significantly, but also gives rise to an integrity problem since data may be lost if the power goes down unexpectedly during the restoration process.

An intermediate software layer called flash translation layer (FTL) addresses the above mentioned issues. It enables the file systems access flash memory as access magnetic disks and prolong the flash memory life cycle.

This application note will help the user to use the FTL interface.

Table of contents

1. Abbreviations and definitions 3
2. Related parts 3
   - 2.1 Atmel AT25DFx series 3
   - 2.2 Atmel AT45DBx series 3
3. FTL library 3
   - 3.1 Introduction 3
   - 3.2 Architecture 4
   - 3.3 Interface 5
     - 3.3.1 Up layer 5
     - 3.3.2 FTL 6
     - 3.3.3 Hardware abstract layer 8
       - 3.3.3.1 Atmel AT25DFx series DataFlash 8
       - 3.3.3.2 Atmel AT45DBx series DataFlash 10
   - 3.4 Error control 12
4. Usage 13
   - 4.1 Integrate library to your project 13
   - 4.2 Example with Atmel EVK1100 14
   - 4.3 Using FTL through FAT 17
5. Performance 23

1. Abbreviations and definitions

- FTL: Flash Translation Layer
- HAL: Hardware Abstract Layer
- sector Logical unit, 512Bytes
- page Physical unit, actual page size of the Atmel® DataFlash®

2. Related parts

FTL library can be applied to the following parts:

2.1 Atmel AT25DFx series

- AT25DF641A
- AT25DF641
- AT25DF321A
- AT25DF321
- AT25DF161
- AT25DF081A
- AT25DF041A

2.2 Atmel AT45DBx series

- AT45DB642D
- AT45DB321D
- AT45DB161D

3. FTL library

3.1 Introduction

FTL redirects the logical address from uplayer to physical address in flash memory, and average the program/erase operation in different blocks.

FTL features the function of wear leveling, bad block management, garbage collection, defrag and power loss recovery. It fully supports FAT file system.

Due to different features of the AT25DFx series and AT45DBx series DataFlash, two libraries have been implemented respectively. Most of the two libraries are the same; the only difference for user is the hal layer interface. This will be detailed in Section 3.3.3.

The footprint of FTL library is about 6Kbyte. The RAM usage depends on the block number.

Atmel AT25DFx series:

Take AT25DF321A as an example. Its size is 4Mbyte and use 64K as a FTL blocks. The maximum RAM usage is 2Kbyte.

Approximately, the RAM usage is 1.1Kbyte + 14 × (block number - used block).

Atmel AT45DBx series:

Take AT45DB642D as an example. Its size is 8Mbyte and use 64K as a FTL block. The maximum RAM usage is 4.1Kbytes.

Approximately, the RAM usage is 2.1Kbyte + 16 × (block number - used block).

3.2 Architecture

FTL is divided into three parts, up layer, ftl and hal. Figure 3-1 illustrates the architecture. Figure 3-2 shows briefly the redirection of the logical address from uplayer to physical address in flash.

Figure 3-1. FTL architecture.

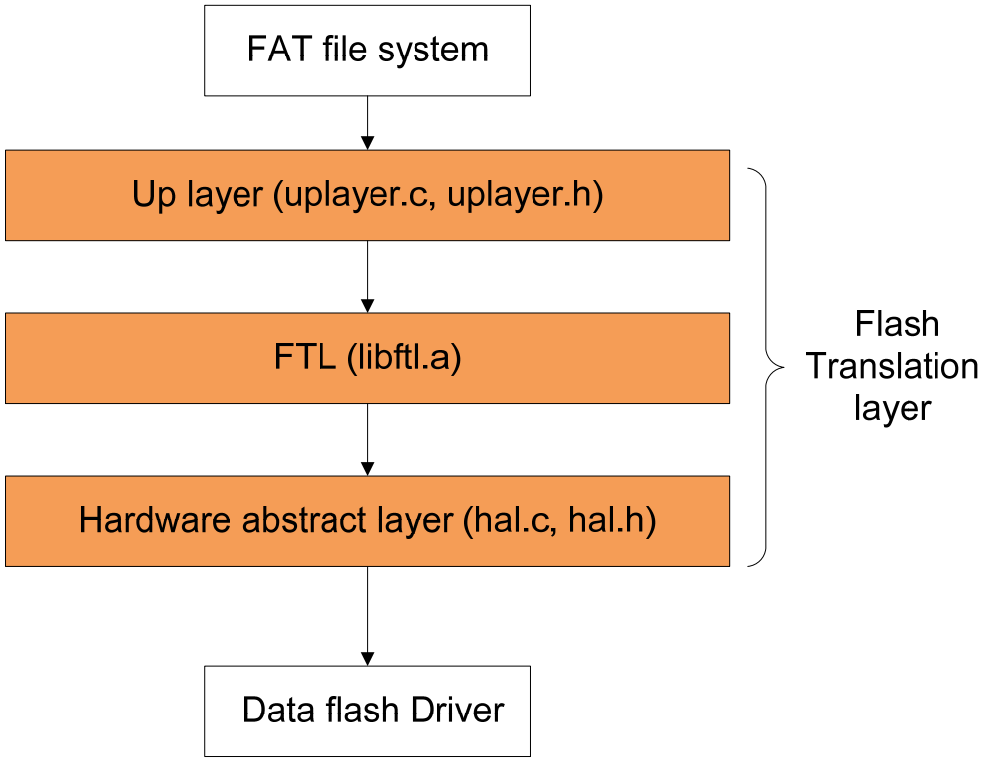
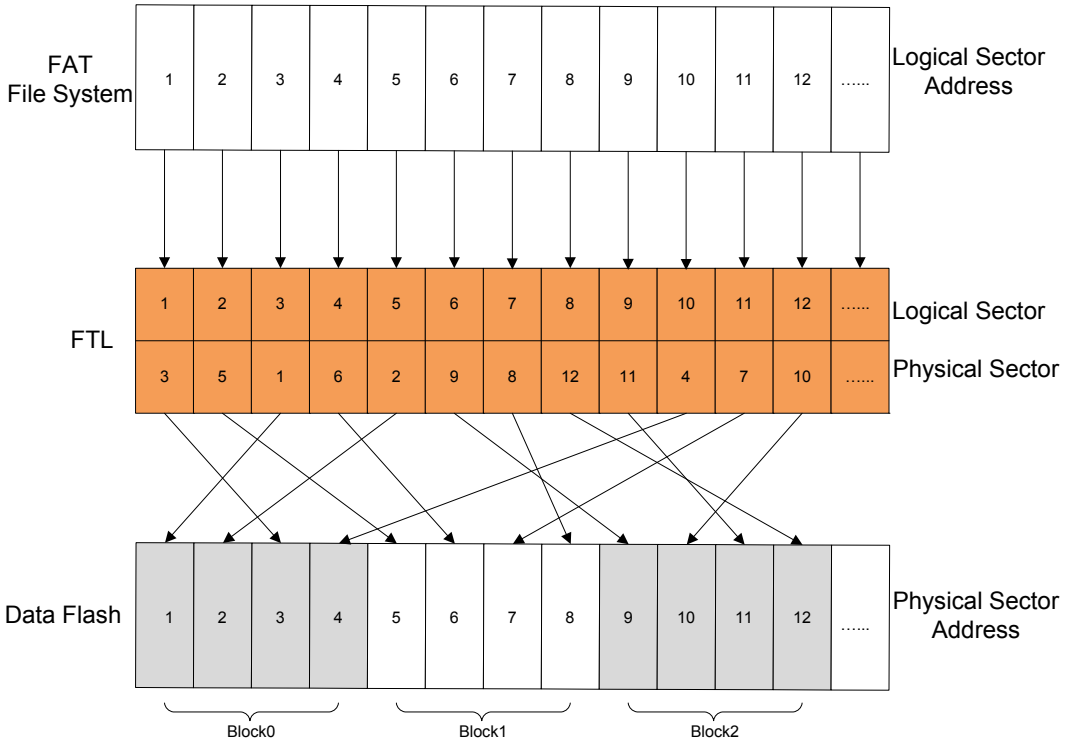


Figure 3-2. FTL redirect logical address to physical address.



3.3 Interface

3.3.1 Up layer

This layer provides the interface for FAT file system.

- uplayer_status_t **test_unit_ready**(void)
  - This function is used to check memory state

Argument	Type	Comment
None	-	-

Return value	Comment
UPLAYER_SUCCESS	Memory is ready
UPLAYER_UNIT_NO_PRESENT	Memory is not ready

- uplayer_status_t **read_capacity**(uint32_t *nb_sectors)
  - This function is used to read memory capacity

Argument	Type	Comment
nb_sector	uint32_t*	Pointer to the variable which store the number of sectors

Return value	Comment
UPLAYER_SUCCESS	Read memory capacity successful
UPLAYER_FAILURE	Read memory capacity fail

- bool **test_wr_protect**(void)
  - This function is used to check memory if write protect

Argument	Type	Comment
None	-	-

Return value	Comment
true	Memory is write protect
false	Memory is not write protect

- bool **test_unit_removal**(void)
  - This function is used to check memory if be removed

Argument	Type	Comment
None	-	-

Return value	Comment
true	Memory is removed
false	Memory is not removed

- uplayer_status_t **ram_2_df**(uint32_t sector, void *ram)
  - This function is used to write one page data to memory

Argument	Type	Comment
sector	uint32_t	The logical sector number
ram	void*	RAM to store one sector (512bytes) data write to memory

Return value	Comment
UPLAYER_SUCCESS	Write memory with one sector data successful
UPLAYER_FAILURE	Write memory with one sector data fail

- uplayer_status_t **df_2_ram**(uint32_t sector, void *ram)
  - This function is used to read one page data from memory

Argument	Type	Comment
sector	uint32_t	The logical sector number
ram	void*	RAM to store one sector (512bytes) data read from memory

Return value	Comment
UPLAYER_SUCCESS	Read memory with one sector data successful
UPLAYER_FAILURE	Read memory with one sector data fail

All the routines are described in uplayer.h file.

3.3.2 FTL

This layer provides interfaces for uplayer, and can also be called directly when you don't want to use FAT.

- ftl_status_t **ftl_init**(void)
  - This function is used to initialize ftl structure

Argument	Type	Comment
None	-	-

Return value	Comment
FTL_INIT_SUCCESS	FTL structure initialization successful
FTL_INIT_FAILURE	FTL structure initialization fail

Note: This routine will erase all the contents of the memory when FTL has not been used before on this memory.

- ftl_status_t **ftl_read**(uint32_t sector, uint8_t *buf)
  - This function is used to read one page data from memory

Argument	Type	Comment
sector	uint32_t	The logical sector number
buf	uint8_t*	Buf to store one sector (512bytes) data read from memory

Return value	Comment
FTL_READ_PAGE_SUCCESS	Read memory with one sector data successful
FTL_READ_PAGE_FAILURE	Read memory with one sector data fail

- ftl_status_t **ftl_write**(uint32_t sector, uint8_t *buf)
  - This function is used to write one page data to memory

Argument	Type	Comment
sector	uint32_t	The logical sector number
buf	uint8_t*	Buf to store one sector (512bytes) data write to memory

Return value	Comment
FTL_WRITE_PAGE_SUCCESS	Write memory with one sector data successful
FTL_WRITE_PAGE_FAILURE	Write memory with one sector data fail

- ftl_status_t **ftl_test_unit_ready**(void)
  - This function is used to test memory state

Argument	Type	Comment
None	-	-

Return value	Comment
FTL_UINT_READY	Memory is ready
FTL_UINT_NOT_READY	Memory is not ready

- ftl_status_t **ftl_read_capacity**(uint32_t *nb_sectors)
  - This function is used to read memory capacity

Argument	Type	Comment
nb_sectors	uint32_t*	Pointer to the variable which store the number of sectors

Return value	Comment
FTL_READ_CAPACITY_SUCCESS	Read memory capacity successful
FTL_READ_CAPACITY_FAILURE	Read memory capacity fail

- ftl_status_t **ftl_test_unit_wr_protect**(void)
  - This function is used to test memory write protect state

Argument	Type	Comment
None	-	-

Return value	Comment
FTL_UNIT_WR_PROTECT	Memory is write protect
FTL_UNIT_WR_NO_PROTECT	Memory is not write protect

- ftl_status_t **ftl_unit_unprotect**(void)
  - This function is used to unprotect memory

Argument	Type	Comment
None	-	-

Return value	Comment
FTL_UNIT_UNPROTECT_SUCCESS	Unprotect memory successful
FTL_UNIT_UNPROTECT_FAILURE	Unprotect memory fail

- ftl_status_t **ftl_test_unit_removal**(void)
  - This function is used to test memory if be removed

Argument	Type	Comment
None	-	-

Return value	Comment
FTL_UNIT_REMOVAL	Memory is removed
FTL_UNIT_NO_REMOVAL	Memory is not removed

Note: Use ftl_read_capacity(…) to get the available number of sectors.

All the routines are described in ftl.h file.

3.3.3 Hardware abstract layer

This layer provides interfaces for FTL. The routines in this layer will call data flash drivers which implemented by user. The address arguments are all physical address in this layer.

3.3.3.1 Atmel AT25DFx series DataFlash

- ftl_status_t **hal_block_erase**(uint32_t addr)
  - This function is used to erase memory block

Argument	Type	Comment
addr	uint32_t	The physical address of the memory block to erase

Return value	Comment
FTL_BLOCK_ERASE_SUCCESS	Memory block erase successful
FTL_BLOCK_ERASE_FAILURE	Memory block erase fail

Note: The block erase unit is 64Kbyte block.

- ftl_status_t **hal_read_id**(uint8_t *buf)
  - This function is used to read memory id

Argument	Type	Comment
buf	uint8_t*	Buf (4bytes) used to store memory ID

Return value	Comment
FTL_GET_CHIP_ID_SUCCESS	Get memory ID successful
FTL_GET_CHIP_ID_FAILURE	Get memory ID fail

- ftl_status_t **hal_set_block_status**(uint32_t addr, uint8_t *buf, uint16_t count)
  - This function is used to set block status

Argument	Type	Comment
addr	uint32_t	The physical address of the memory to write the bytes
buf	uint8_t*	Buf to store the data write to memory
count	uint16_t	Number of bytes need to write

Return value	Comment
FTL_SET_BLOCK_STATUS_SUCCESS	Write successfully
FTL_SET_BLOCK_STATUS_FAILURE	Write fail

- ftl_status_t **hal_get_block_status**(uint32_t addr, uint8_t *buf, uint16_t count)
  - This function is used to get the block status

Argument	Type	Comment
addr	uint32_t	The physical address of the memory to read the bytes
buf	uint8_t*	Buf to store the data read from memory
count	uint16_t	Number of bytes need to read

Return value	Comment
FTL_GET_BLOCK_STATUS_SUCCESS	Read successfully
FTL_GET_BLOCK_STATUS_FAILURE	Read fail

- ftl_status_t **hal_write_page**(uint32_t sector, uint8_t *buf)
  - This function is used to write one sector data to memory

Argument	Type	Comment
sector	uint32_t	The physical sector number
buf	uint8_t*	Buf to store one sector (512bytes) data write to memory

Return value	Comment
FTL_WRITE_PAGE_SUCCESS	Write memory with one sector data successful
FTL_WRITE_PAGE_FAILURE	Write memory with one sector data fail

- ftl_status_t **hal_read_page**(uint32_t sector, uint8_t *buf)
  - This function is used to read one sector data from memory

Argument	Type	Comment
sector	uint32_t	The physical sector number
buf	uint8_t*	Buf to store one page (512bytes) data read from memory

Return value	Comment
FTL_READ_PAGE_SUCCESS	Read memory with one sector data successful
FTL_READ_PAGE_FAILURE	Read memory with one sector data fail

- ftl_status_t **hal_test_unit_wr_protect**(void)
  - This function is used to test memory if write protect

Argument	Type	Comment
None	-	-

Return value	Comment
FTL_UNIT_WR_PROTECT	Memory is write protect
FTL_UNIT_WR_NO_PROTECT	Memory is not write protect

- ftl_status_t **hal_unit_unprotect**(void)
  - This function is used to unprotect memory

Argument	Type	Comment
None	-	-

Return value	Comment
FTL_UNIT_UNPROTECT_SUCCESS	Unprotect memory successful
FTL_UNIT_UNPROTECT_FAILURE	Unprotect memory fail

- ftl_status_t **hal_test_unit_removal**(void)
  - This function is used to test if memory be removed

Argument	Type	Comment
None	-	-

Return value	Comment
FTL_UNIT_REMOVAL	Memory is removed
FTL_UNIT_NO_REMOVAL	Memory is not removed

3.3.3.2 Atmel AT45DBx series DataFlash

- ftl_status_t **hal_read_id**(uint8_t *buf)
  - This function is used to read memory id

Argument	Type	Comment
buf	uint8_t*	Buf (4bytes) used to store memory ID

Return value	Comment
FTL_GET_CHIP_ID_SUCCESS	Get memory ID successful
FTL_GET_CHIP_ID_FAILURE	Get memory ID fail

- ftl_status_t **hal_set_block_status**(uint32_t addr, uint8_t *buf, uint16_t count)
  - This function is used to set block status

Argument	Type	Comment
addr	uint32_t	The physical address of the memory to write the bytes
buf	uint8_t*	Buf to store the data write to memory
count	uint16_t	Number of bytes need to write

Return value	Comment
FTL_SET_BLOCK_STATUS_SUCCESS	Write successfully
FTL_SET_BLOCK_STATUS_FAILURE	Write fail

- ftl_status_t **hal_get_block_status**(uint32_t addr, uint8_t *buf, uint16_t count)
  - This function is used to get the block status

Argument	Type	Comment
addr	uint32_t	The physical address of the memory to read the bytes
buf	uint8_t*	Buf to store the data read from memory
count	uint16_t	Number of bytes need to read

Return value	Comment
FTL_GET_BLOCK_STATUS_SUCCESS	Read successfully
FTL_GET_BLOCK_STATUS_FAILURE	Read fail

- ftl_status_t **hal_set_page_status**(uint32_t addr, uint8_t *buf, uint16_t count)
  - This function is used to set page status

Argument	Type	Comment
addr	uint32_t	The physical address of the memory to write the bytes
buf	uint8_t*	Buf to store the data write to memory
count	uint16_t	Number of bytes need to write

Return value	Comment
FTL_SET_BLOCK_STATUS_SUCCESS	Write successfully
FTL_SET_BLOCK_STATUS_FAILURE	Write fail

- ftl_status_t **hal_get_page_status**(uint32_t addr, uint8_t *buf, uint16_t count)
  - This function is used to get page status

Argument	Type	Comment
addr	uint32_t	The physical address of the memory to read the bytes
buf	uint8_t*	Buf to store the data read from memory
count	uint16_t	Number of bytes need to read

Return value	Comment
FTL_GET_BLOCK_STATUS_SUCCESS	Read successfully
FTL_GET_BLOCK_STATUS_FAILURE	Read fail

- ftl_status_t **hal_write_page**(uint32_t sector, uint8_t *buf, bool spare)
  - This function is used to write one sector data to memory

Argument	Type	Comment
sector	uint32_t	The physical sector number
buf	uint8_t*	Buf to store one sector (512bytes) data write to memory
spare	bool	Write sector with spare or not

Return value	Comment
FTL_WRITE_PAGE_SUCCESS	Write memory with one sector data successful
FTL_WRITE_PAGE_FAILURE	Write memory with one sector data fail

- ftl_status_t **hal_read_page**(uint32_t sector, uint8_t *buf, bool spare)
  - This function is used to read one sector data from memory

Argument	Type	Comment
sector	uint32_t	The physical sector number
buf	uint8_t*	Buf to store one sector (512bytes) data read from memory
spare	bool	Read sector with spare or not

Return value	Comment
FTL_READ_PAGE_SUCCESS	Read memory with one page data successful
FTL_READ_PAGE_FAILURE	Read memory with one page data fail

- ftl_status_t **hal_test_unit_wr_protect**(void)
  - This function is used to test memory if write protect

Argument	Type	Comment
None	-	-

Return value	Comment
FTL_UNIT_WR_PROTECT	Memory is write protect
FTL_UNIT_WR_NO_PROTECT	Memory is not write protect

- ftl_status_t **hal_unit_unprotect**(void)
  - This function is used to unprotect memory

Argument	Type	Comment
None	-	-

Return value	Comment
FTL_UNIT_UNPROTECT_SUCCESS	Unprotect memory successful
FTL_UNIT_UNPROTECT_FAILURE	Unprotect memory fail

- ftl_status_t **hal_test_unit_removal**(void)
  - This function is used to test if memory be removed

Argument	Type	Comment
None	-	-

Return value	Comment
FTL_UNIT_REMOVAL	Memory is removed
FTL_UNIT_NO_REMOVAL	Memory is not removed

All the routines are described in the hal.h file.

This layer should be implemented by the user. Users need to use this layer to perform a full test of their DataFlash driver.

3.4 Error control

All the routines return a status which is described in ftl_status.h. Using these statuses we can locate where the error happen easily.

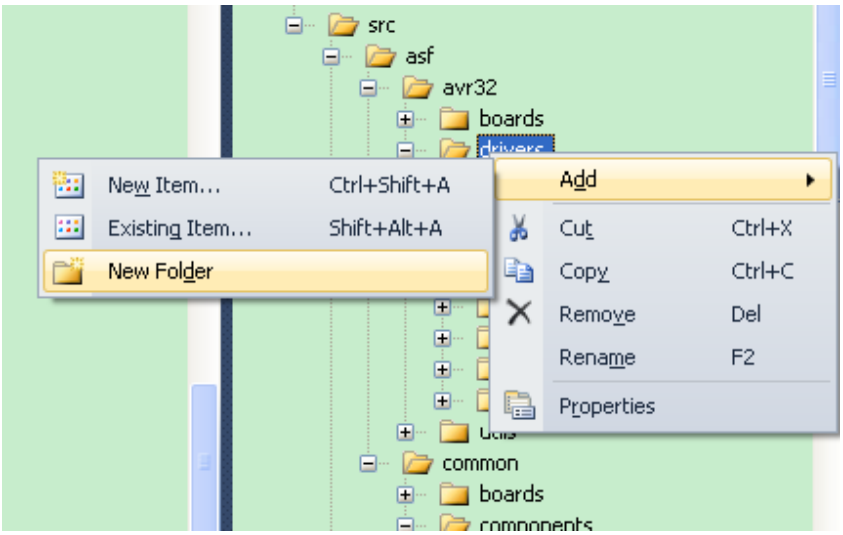
The routines in uplayer.c return a status which is known by FAT.

4. Usage

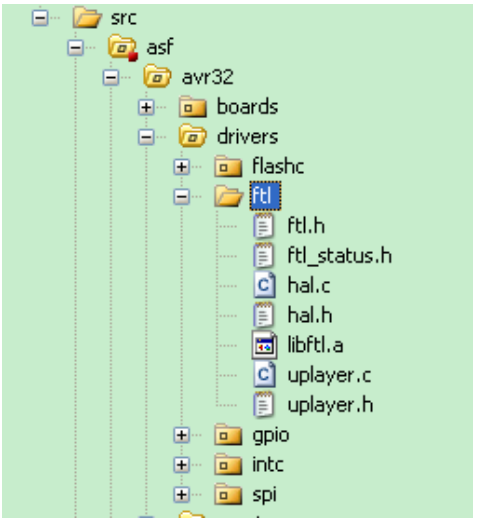
All examples and test cases are tested in Atmel Studio 6.0.

4.1 Integrate library to your project

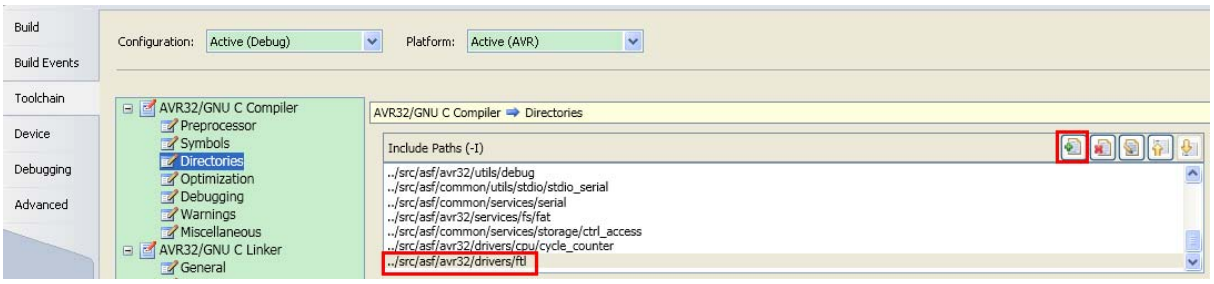
1. Create a new folder in your project; you can create this folder under drivers.

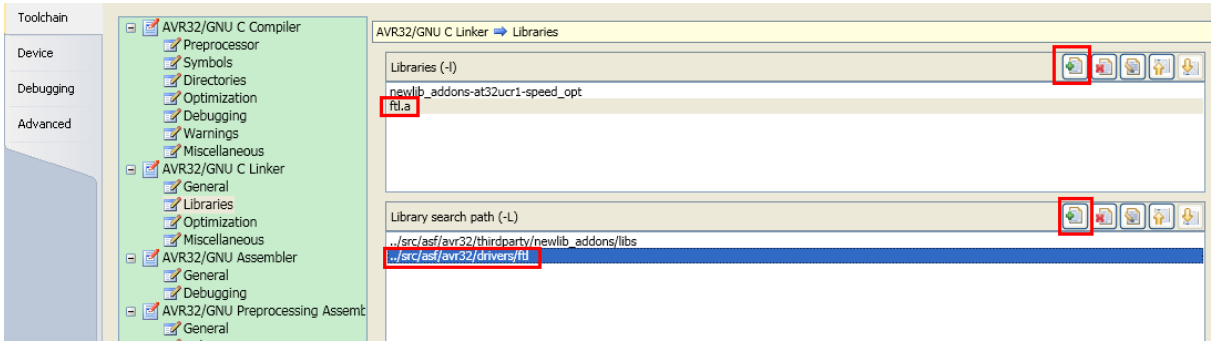


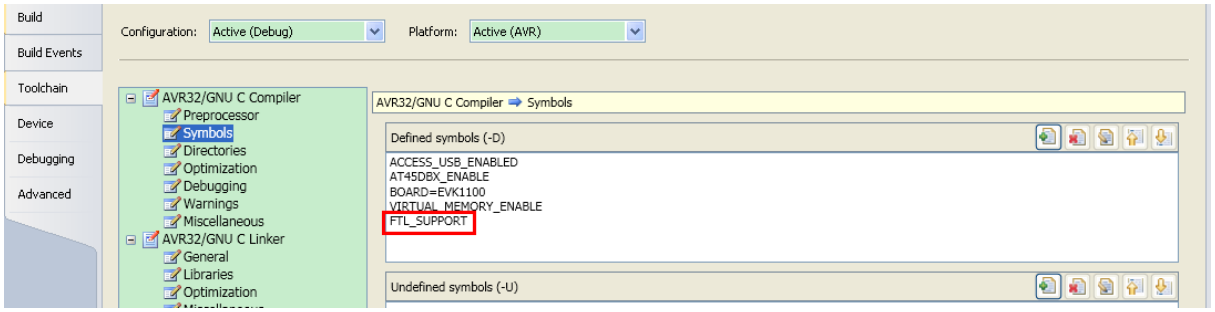
2. Add the FTL library files to this folder. Here we create a new folder ftl under drivers.



3. Set the search path for toolchains and enable FTL_SUPPORT. Right click on the project, select Properties.







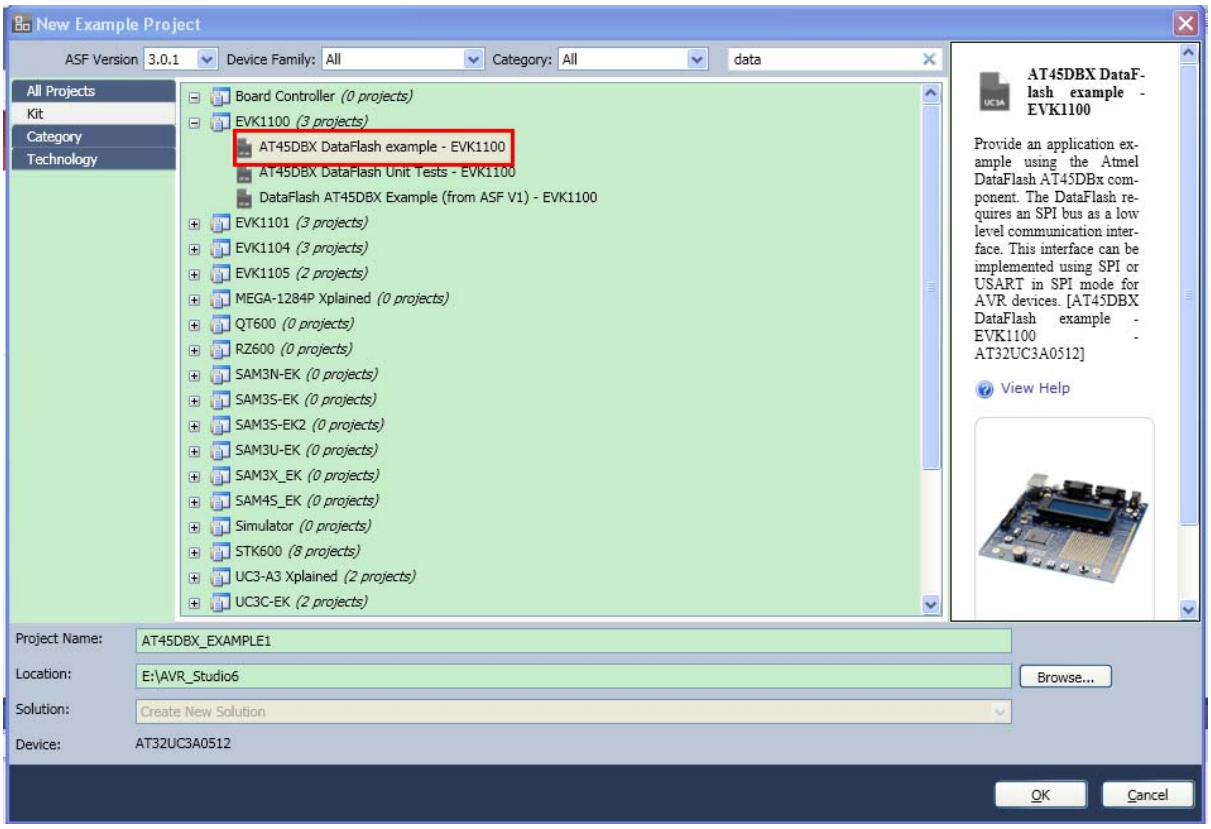
4. Add low level data flash driver (at25dfx.c/h or at45dbx.c/h) to your project. If these files have already existed in your project, just replace them.

Note: Make sure the data flash can work (etc. read/write ok) before using FTL library.

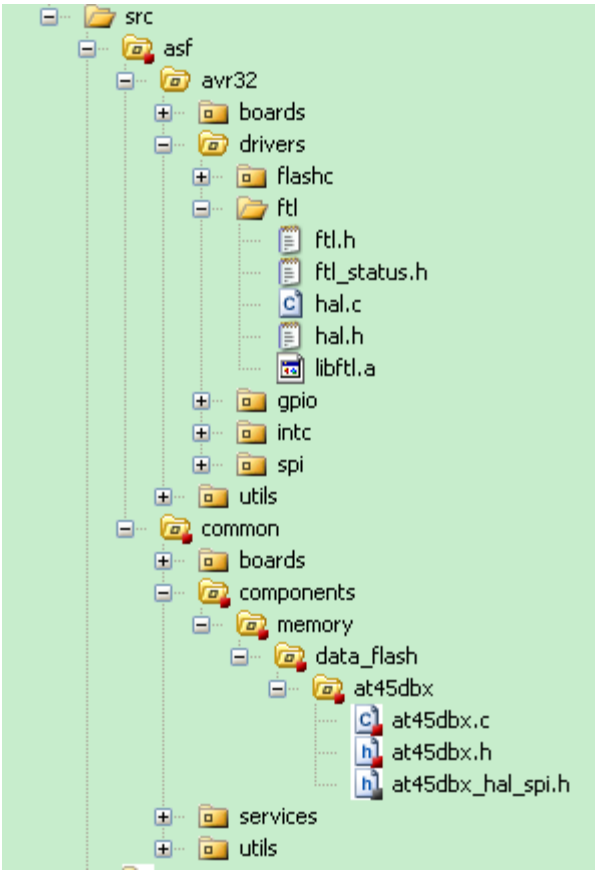
4.2 Example with Atmel EVK1100

This section details a quick way to test the FTL functionality. The step-by-step procedure is as follow:

1. Create a new AT45DBX DataFlash example.



2. Add FTL library to project. This step has been detailed in Section 4.1.



3. Add FTL functionality test routine in file at45dbx_example.c.

```
#include "ftl.h"
int main(void)
{
    uint16_t i;
    sysclk_init();

    // Initialize the board.
    // The board-specific conf_board.h file contains the configuration of the board
    // initialization.
    board_init();
    at45dbx_init();
    if(at45dbx_mem_check()==true)    {
        gpio_set_pin_low(DATA_FLASH_LED_EXAMPLE_0);
    } else
    {
        test_ko();
    }
//ADD FTL functionality test here
    test_case_1();

    // Prepare half a data flash sector to 0xAA
    for(i=0;i<AT45DBX_SECTOR_SIZE/2;i++) {
        ram_buf[i]=0xAA;
    }
    // And the remaining half to 0x55
    for(;i<AT45DBX_SECTOR_SIZE;i++) {
        ram_buf[i]=0x55;
    }
```

```
bool test_case_1(void)
{
        uint32_t i;
        ftl_status_t status;
        uint8_t page_buf[512];

        //We need call ftl_init() before we use FTL
        ftl_init();

        //Here we write 400Mbytes to page 0 of dataflash
        //to test ftl read/write and wear leveling
        for(i = 0; i < 819200; i++){ // 400MByte data

                memset(page_buf,0xaa,512);
                //Write one page data to sector number 0
                status = ftl_write(0, page_buf);
                if(status != FTL_WRITE_PAGE_SUCCESS) {
                        print_dbg("Write fail!\r\n");
                        return false;
                }

                memset(page_buf,0,512);
                //Read one page data from sector number 0
                status = ftl_read(0, page_buf);
                if(status != FTL_READ_PAGE_SUCCESS) {
                        print_dbg("Read fail!\r\n");
                        return false;
                }
                //Verify the data write and read
                if(compare_buf(0xaa, page_buf)){
                        print_dbg("Verify fail!\r\n");
                        return false;
                }
        }
        return true;
}
```

Note: Routine ftl_init() should be called firstly when the FTL is used.

If you use the FTL the first time, the routine ftl_init() will erase the entire chip. You should be careful if there are some important data in the chip.

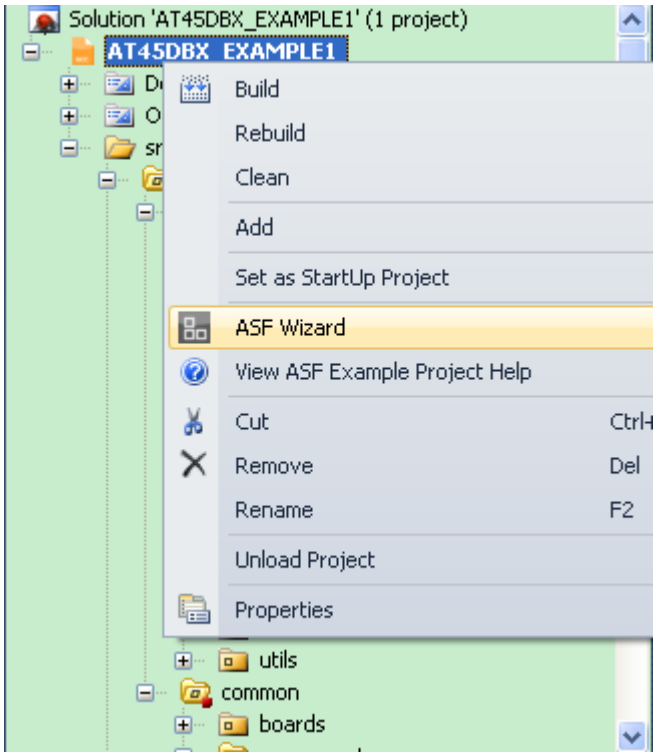
4. All things are done now. The only thing left is to compile the source code.

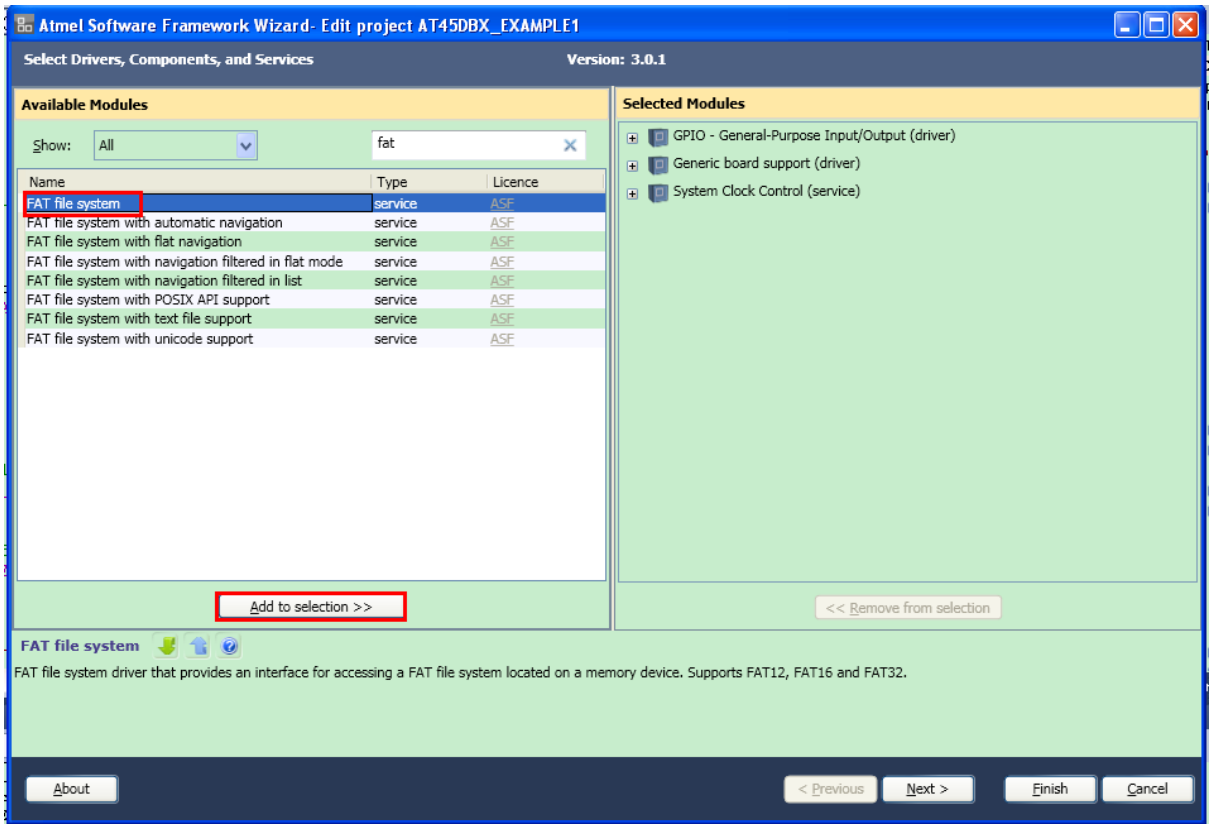
4.3 Using FTL through FAT

This section details a quick way to test the FTL functionality through FAT interface.

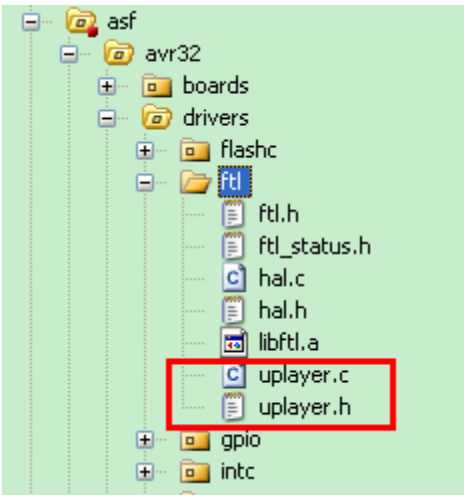
Based on the Atmel EVK1100 AT45DBX Example, which is detailed in Section 4.2, the step by step procedure is as follow:

1. Add FAT service to the project.

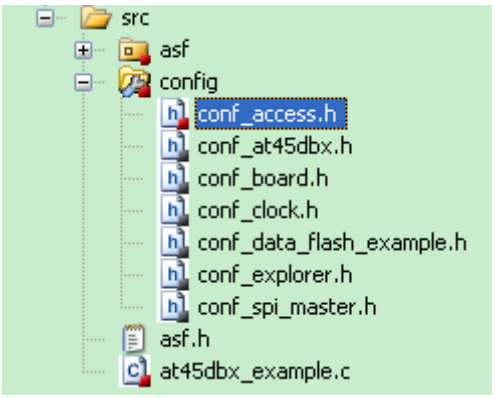




2. Add FTL uplayer.c and uplayer.h to the ftl folder.



3. Add FTL uplayer interface for FAT in config/conf_access.h.



```
/*!  name LUN 1 Definitions
 */
//! @{
#ifdef FTL_SUPPORT
#define AT45DBX_MEM                         LUN_1
#define LUN_ID_AT45DBX_MEM                  LUN_ID_1
#define LUN_1_INCLUDE                       "uplayer.h"
#define Lun_1_test_unit_ready               test_unit_ready
#define Lun_1_read_capacity                 read_capacity
#define Lun_1_wr_protect                    test_wr_protect
#define Lun_1_removal                       test_unit_removal
#define Lun_1_usb_read_10                   usb_read_10
#define Lun_1_usb_write_10                  usb_write_10
#define Lun_1_mem_2_ram                     df_2_ram
#define Lun_1_ram_2_mem                     ram_2_df
#define LUN_1_NAME                          "\"AT45DBX Data Flash\""
#else
#define AT45DBX_MEM                         LUN_1
#define LUN_ID_AT45DBX_MEM                  LUN_ID_1
#define LUN_1_INCLUDE                       "at45dbx_mem.h"
#define Lun_1_test_unit_ready               at45dbx_test_unit_ready
#define Lun_1_read_capacity                 at45dbx_read_capacity
#define Lun_1_wr_protect                    at45dbx_wr_protect
#define Lun_1_removal                       at45dbx_removal
#define Lun_1_usb_read_10                   at45dbx_usb_read_10
#define Lun_1_usb_write_10                  at45dbx_usb_write_10
#define Lun_1_mem_2_ram                     at45dbx_df_2_ram
#define Lun_1_ram_2_mem                     at45dbx_ram_2_df
#define LUN_1_NAME                          "\"AT45DBX Data Flash\""
#endif
```

4. Change FAT feature level in file config/conf_explorer.h.

```
//! Level of features.
//! Select among:
//!  -  c FSFEATURE_READ:           All read functions.
//!  -  c FSFEATURE_WRITE:          nav_file_copy(), nav_file_paste(), nav_file_
//!  -  c FSFEATURE_WRITE_COMPLET:  FSFEATURE_WRITE functions and nav_drive_form
//!  -  c FSFEATURE_ALL:            All functions.
#define FS_LEVEL_FEATURES     (FSFEATURE_READ | FSFEATURE_WRITE_COMPLET)
#warning "FAT Level of Features set to None by default: edit the conf_explorer.h
```

5. Enable stream memory to memory interface to make compile pass.

```
#define ACCESS_STREAM          true //!< Streaming MEM <-> MEM interface.
#define ACCESS_STREAM_RECORD false //!< Streaming MEM <-> MEM interface in record mode.
#define ACCESS_MEM_TO_MEM      true //!< MEM <-> MEM interface.
#define ACCESS_CODEC          false //!< Codec interface.
//! @}
```

6. Add test routine in the at45dbx_example.c file and include asf.h in this file.

```
//Each file 1MByte size
#define NB_WRITE        2000L
#define BUF_SIZE        512L
bool test_case_2(void) //test ftl under fat
{
        uint16_t i, cnt;
        uint32_t index = 100;
        uint8_t page_buf[BUF_SIZE]; //Buf used to write
        uint8_t read_buf[BUF_SIZE]; //Buf used to read

        uint8_t *name[3] = {"data1","data2","data3};

        if(!mount_fat()) { //format Dataflash and mount fat
                printf("Mount fat fail!\r\n");
                return false;
        }
        memset(page_buf,0xa5,512);

        while(index--) {
                //Write file data1, data2, data3, each file 1Mbytes
                for(i = 0; i < 3; i++) {
                        printf("File: %s created\r\n", name[i]);
                        if(!nav_file_create((const FS_STRING)name[i])) {
                                printf("Create file: %s fail\r\n", name[i]);
                                return false;
                        }
                        if(!file_open(FOPEN_MODE_W)) {
                                printf("Open file: %s for write fail\r\n", name[i]);
                                return false;
                        }
                        for(cnt = 0; cnt < NB_WRITE; cnt++) {
                                if(!file_write_buf(page_buf, BUF_SIZE)) {
                                        file_close();
                                        printf("File write fail: %s\r\n", name[i]);
                                        return false;
                                }
                        }
                        file_close();
                        //Read back to verify it
                        if(!file_open(FOPEN_MODE_R)) {
                                printf("Open file: %s for read fail\r\n", name[i]);
                                return false;
                        }
                        while (file_eof()==false) {
                                file_read_buf(read_buf, BUF_SIZE);
                                if(compare_buf(0xa5, read_buf)) {
                                        printf("Verify file %s fail!\r\n", name[i]);
                                        file_close();
                                        return false;
                                }
                        }
                        file_close();
                        printf("File: %s write OK, file size: %dMByte\r\n\n", name[i],1);
                }
                // Delete file data1, data2, data3, then we can write them in next loop
                if(!file_delete()) {
                        printf("File delete fail!\r\n");
                        return false;
                }
        }
        return true;
}
```

```
//Format dataflash and mount fat
void mount_fat()
{
	nav_reset();
	//Select navigator 0
	if(!nav_select(0)) {
		printf("Nav select fail \r\n");
		return false;
	}
	//Navigator select the dataflash driver
	if(!nav_drive_set(0)) {
		printf("Nav drive set fail!\r\n");
		return false;
	}
	//Format dataflash
	if( !nav_drive_format(FS_FORMAT_DEFAULT) ) {
		printf("Format fail!\r\n");
		return false;
	}
	//Mount dataflash
	if(!nav_partition_mount()) {
		printf("Partition mount fail\r\n");
		return false;
	}
}
```

```
//Delete files
bool file_delete()
{
	uint8_t name[3] = {“data1”,”data2”,”data3”};
	uint8_t i;
	for(i = 0; i < 3; i++) {
		//Select the file to be deleted
		if(!nav_setcwd( (FS_STRING)name[i], true, false)) {
			printf("Nav setcwd: %s fail\r\n", name[i]);
			return false;
		}
		//File delete
		if(!nav_file_del( false )) {
			printf("File: %s delete fail\r\n", name[i]);
			return false;
		}

		printf("File: %s deleted \r\n", name[i]);
	}
}
```

5. Performance

FTL wear leveling performance is tested on the Atmel DataFlash AT45DB642D. This performance can also be applied to the Atmel AT25DFx series DataFlash.

Figure 5-1 shows the wear leveling on the DataFlash AT45DB642D. This result was got through 400Mbytes data writing in the same page of the DataFlash using FTL interface.

X-axis value is block number and Y-axis value is erase count in different blocks.

Figure 5-1. Wear leveling on the Atmel AT45DB642D using FTL interface.

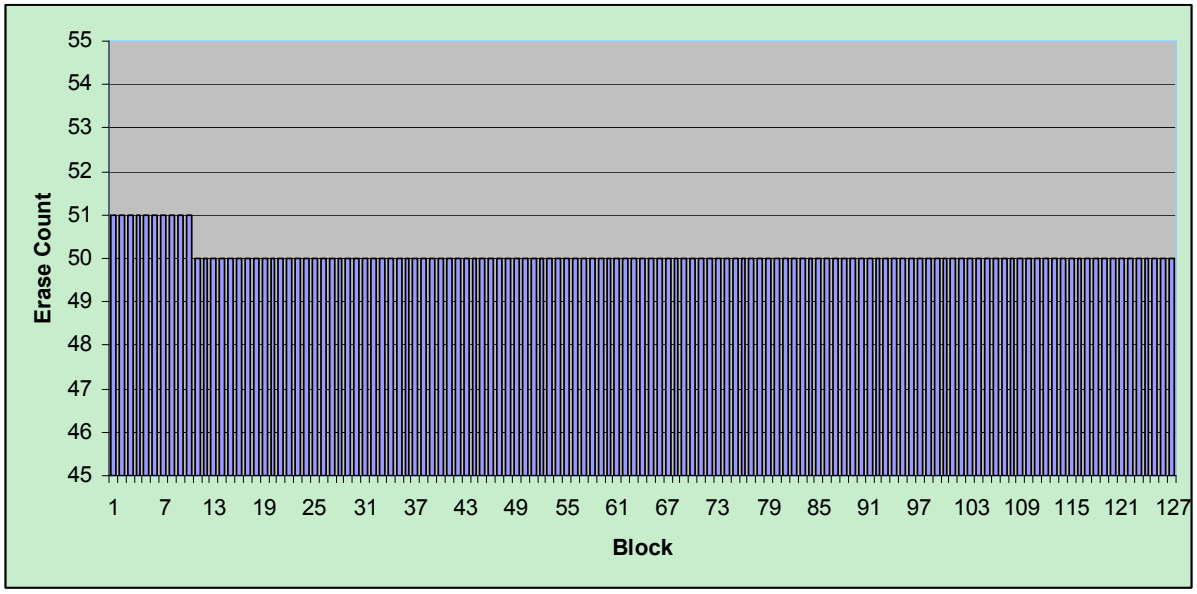
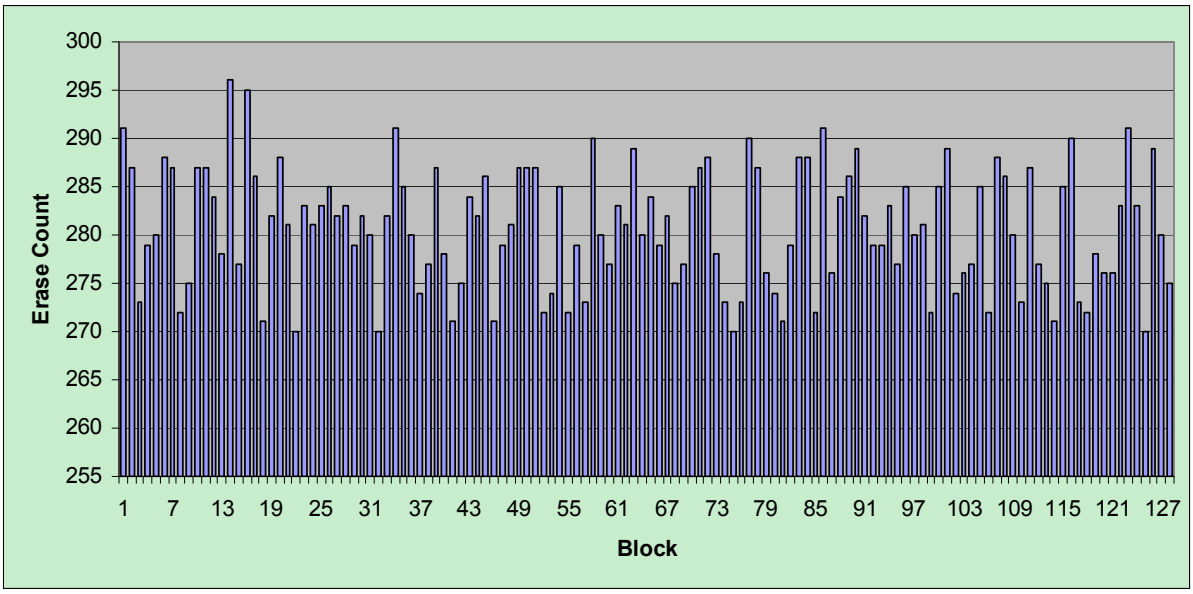


Figure 5-2 shows the wear leveling on the DataFlash AT45DB642D after about 400MByte data write through FAT interfaces.

X-axis value is block number and Y-axis value is erase count in different blocks.

Figure 5-2. Wear leveling on the Atmel AT45DB624D using FAT interfaces.



FTL divides DataFlash lifecycle to four levels (for example, AT45DB642D 100,000 times program/erase life cycles, each level has 25,000 program/erase cycles). The block will not be used when its erase count gets up to the first level while other blocks' erase counts are still below 25,000. Free blocks are allocated by round robin scheduling again when all the blocks get up to the first level. This process is going on till the blocks get its end life cycle.

Atmel Corporation
2325 Orchard Parkway
San Jose, CA 95131
USA
Tel: (+1)(408) 441-0311
Fax: (+1)(408) 487-2600
www.atmel.com

Atmel Asia Limited
Unit 01-5 & 16, 19F
BEA Tower, Millennium City 5
418 Kwun Tong Road
Kwun Tong, Kowloon
HONG KONG
Tel: (+852) 2245-6100
Fax: (+852) 2722-1369

Atmel Munich GmbH
Business Campus
Parkring 4
D-85748 Garching b. Munich
GERMANY
Tel: (+49) 89-31970-0
Fax: (+49) 89-3194621

Atmel Japan G.K.
16F Shin-Osaki Kangyo Building
1-6-4 Osaki
Shinagawa-ku, Tokyo 141-0032
JAPAN
Tel: (+81)(3) 6417-0300
Fax: (+81)(3) 6417-0370