SafeFTL User Guide

Version 2.40

For use with SafeFTL versions 5.14 and above

Date: 05-Sep-2017 18:01

Table of Contents

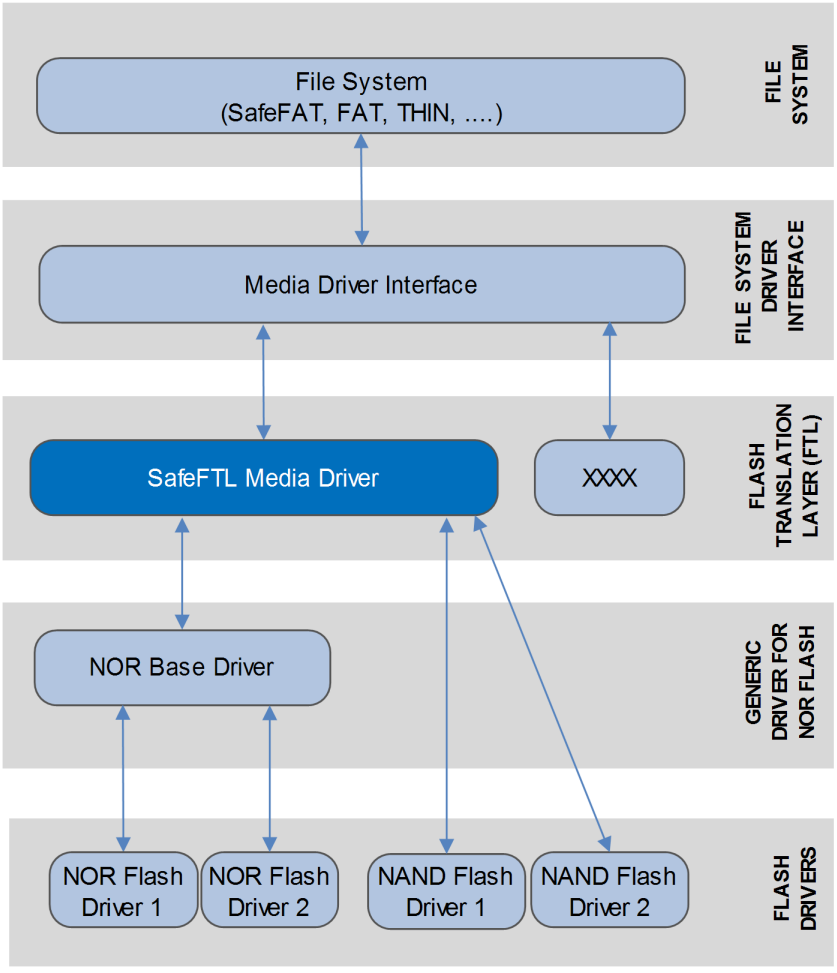
1 System Overview

1.1 Introduction

This manual is for those who want to implement a system using HCC's Safe Flash Translation Layer, SafeFTL. Read this guide thoroughly before implementing SafeFTL.

A Flash Translation Layer (FTL) is a system for attaching arrays of flash to a media driver. SafeFTL presents a simple logical sector interface to an application such as a file system, and manages the underlying complexity efficiently and safely. SafeFTL is a highly efficient FTL, designed for use with most standard types of NAND and NOR flash, in simple or complex configurations. SafeFTL is a media driver which fully conforms to the *HCC Media Driver Interface Specification*.

The system structure is shown in the diagram below:



Note the following:

- The file system can be any HCC file system that addresses logical sector arrays (including SafeFAT, FAT, and THIN).
- The SafeFTL media driver is the FTL. It manages a set of attached flash arrays. It can attach each array as a drive to any file system which uses the *HCC Media Driver Interface Specification*. SafeFTL can provide access to a wide variety of flash media including NAND flash, NOR flash, and serial flash. Many different SafeFTL drives can be attached simultaneously.
- When the flash media reaches its end of life, the drive becomes and remains read-only.
- The generic NOR base driver is responsible for handling the NOR flash drivers. For each NOR flash drive, you must add an entry to the SafeFTL drive list.
- The NOR flash drivers interface to the generic NOR base driver.
- The NAND flash driver for each array of NAND flash that is to be handled as a separate drive is attached to the SafeFTL media driver. To do this, you add an entry to the SafeFTL drive list.

Note: The generic NOR base driver, NOR flash drivers, and NAND flash drivers all have their own manuals.

This manual describes two types of API function:

- Upper layer interface – use this API to set up SafeFTL.
- Lower layer interface – use this API to set up NAND flash drivers and the generic NOR base driver.

Reference Drivers

SafeFTL is a very flexible system for attaching arrays of flash to a media driver. There are many possible configurations of flash types, arrays, and controllers and it is not possible to provide a verified driver for each combination. However, HCC provides a wide variety of tested reference drivers in very diverse configurations.

When starting on a project, contact HCC with a description of your system configuration and we will provide the most appropriate reference driver. HCC can port drivers if required.

1.2 Read Disturb Monitoring

There are many types of NAND flash with a wide range of characteristics. Some of the larger devices, typically MLC flash, require high numbers of bits to be correctable. This correction is done at the low level driver to meet the requirements of the device.

These devices also have a feature whereby bit errors can develop for a several reasons, one of these being a read or write to a different page. For this reason, it is useful to monitor the level of correction required to a page and, based on this, decide if the page should be rewritten before it becomes irreparable.

Some NAND devices simply recommend when a page should be written, others return the number of bits that need correcting and leave the management software to make a judgement.

The low level driver for the NAND flash can signal to the FTL when a page rewrite is required. The FTL then schedules that page (actually that block, since that the whole block has to be replaced) for rewriting during normal flash maintenance.

It is thus up to the low level driver to decide, based on the algorithms used and the flash specification being managed, when to tell the FTL to do an update.

Two of the Configuration Options relate to this operation:

- MDRIVER_FTL_REWRITE enables this capability.
- MDRIVER_FTL_REWRITE_DURING_READ controls when this is done

The *rewrite_interval* element of the low level drivers *t_ftl_phy* physical description structure tells the FTL how often it should check for the need to rewrite.

1.3 Feature Check

The features of SafeFTL are as follows:

- Designed for integration with both RTOS and non-RTOS based systems.
- Can handle up to 4TB in a single wear-leveled array.
- Can handle arrays of flash greater than 4TB.
- Supports all common NAND/NOR devices.
- Can handle NAND devices with any page size, including 16896, 8448, 4224, 2112, and 528 bytes.
- Can handle file systems with any sector size, including 8KB, 4KB, 2KB, and 512 bytes.
- Can handle any array of NOR flash.
- Can handle multi-chip arrays.
- Supports data block logging.
- Dynamically configurable, based on available flash.

The system provides the following:

- Bad block management.
- Error Correction Code (ECC) algorithms.
- Fail-safety from unexpected reset.
- Atomic update (either the old entry is present or the new one).
- Zero copy block read/write.
- Effective wear leveling algorithms. Typically over 98% of blocks are available.
- Read disturb monitoring.
- A cache option.
- Single Level Cell (SLC), Multi-Level Cell (MLC), and multi-plane support.
- Optimized random read/write.
- Open NAND Flash Interface (ONFI) driver.
- Support for the Common Flash memory Interface (CFI) standard for NOR flash devices.
- Support for the Serial Flash Discoverable Parameter (SFDP) standard for serial NOR flash devices.
- Automatic garbage collection.
- A secure deletion of data option.
- Use of incremental writes only to pages in a block.

1.4 Packages and Documents

Packages

The table below lists the packages that you need in order to use SafeFTL:

Package	Description
hcc_base_doc	This contains the two guides that will help you get started.
media_drv_base	The base media driver package that includes the framework for all media drivers to use.
media_drv_ftl_base	The base SafeFTL package described in this document.
oal_base	The base OS Abstraction Layer (OAL) package.
psp_template_base	The base Platform Support Package (PSP).
media_drv_ftl_nand_ram	A sample flash driver for NAND flash simulated in RAM. This is a template for building a NAND flash driver. NAND flash drivers are written specifically for a particular NAND flash controller (normally integrated with the target microcontroller) and the specific NAND flash array used. HCC can provide proven NAND flash drivers for many different configurations.

Documents

For an overview of HCC file systems and flash management technologies, see Product Information on the main HCC website.

Readers should note the points in the HCC Documentation Guidelines on the HCC documentation website.

HCC Firmware Quick Start Guide

This document describes how to install packages provided by HCC to the target development environment. Also follow the *Quick Start Guide* when HCC provides package updates.

HCC Source Tree Guide

This document describes the HCC source tree. It gives an overview of the system to make clear the logic behind its organization.

HCC Media Driver Interface Guide

This document describes the specification for the upper layer interface that SafeFTL uses. This means that SafeFTL can be used as a set of drives by any file system using this Media Driver Interface standard.

HCC SafeFTL User Guide

This is this document. Each physical driver has its own user guide.

HCC FTL NOR Base Flash Driver User Guide,
HCC FTL NOR RAM Flash Driver User Guide,
HCC FTL NAND Flash Driver User Guide

These documents are provided for each flash driver that can be managed by SafeFTL.

1.5 Change History

This section describes past changes to this manual.

- To view or download earlier manuals, see Archive: SafeFTL User Guide.
- For the history of changes made to the package code itself, see History: media_drv_ftl_base.

The current version of this manual is 2.40. The full list of versions is as follows:

Manual version	Date	Software version	Reason for change
2.40	2017-09-05	5.14	Corrected *Packages* list.
2.30	2017-06-20	5.13	New *Change History* format.
2.20	2016-04-18	5.06	Added three configuration options.
2.10	2015-12-10	5.04	Added *Read Disturb Monitoring*, new *Upper Layer Functions* and new *Types*.
2.00	2015-04-30	4.04	First online version.

2 Source File List

This section describes all the source code files included in the system. These files follow the HCC Embedded standard source tree system, described in the *HCC Source Tree Guide*. All references to file pathnames refer to locations within this standard source tree, not within the package you initially receive.

> **Note:** Do not modify any files except the configuration files and **ftldrv.c**.

2.1 API Header File

The file **src/api/api_mdriver_ftl.h** is the only file that should be included by an application using this module. For details of the API functions, see Application Programming Interface.

2.2 Configuration Files

These files in the directory **src/config** contain all the configurable parameters of the system. Configure these as required. For more details, see Configuration.

File	Description
config_mdriver_ftl.h	Header file containing the Configuration Options.
config_mdriver_ftl.c	Flash drive structure file.

2.3 Version File

The file **src/version/ver_mdriver_ftl.h** contains the version number of this module. This version number is checked by all modules that use this module to ensure system consistency over upgrades.

2.4 Source Code Files

These files in the directory **src/media-drv/ftl/common** are the source files provided with the system. Apart from **ftldrv.c**, **these files should only be modified by HCC**.

File	Description
ftldrv.c	Media driver interface handler source code sample for SafeFTL. You can modify this file.
ftldrv.h	Header file for **ftldrv.c**.
hlayer.c	Wrapper layer source code. (This is required only if 2KB page flash is used with a 512 byte sector interface).
hlayer.h	Wrapper layer header file.
ml_types.h	Internal type definitions.
mlayer.c	Intermediate layer source code.
mlayer.h	Intermediate layer header file.
wear.c	Wear leveling module source code.
wear.h	Wear leveling module header file.

3 Configuration

SafeFTL is a flexible system for managing arrays of flash in many different configurations. In general, it is best to use the default settings for configurable parameters, but you may need to change some of them to meet your needs.

3.1 Configuration Options

Set the system configuration options in the file **src/config/config_mdriver_ftl.h**. This section lists the available options and their default values.

MDRIVER_FTL_MAX_DRIVE

The maximum number of flash media in the system that use FTL. This is the maximum number of flash drives that SafeFTL can manage. The default is 3.

MDRIVER_FTL_USE_HLAYER

Keep this at the default of 1 to include the intermediate HLAYER. HLAYER performs sector size translation from the actual page size of the media to 512 bytes. Without HLAYER, SafeFTL reports the page size of the flash media as *F_PHY.bytes_per_sector*.

MDRIVER_FTL_MAX_PAGE_PER_BLOCK

The maximum number of pages in a block on all the media in the system. Changing this value may change the layout of pages, which affects compatibility with existing SafeFTL volumes. The default is 1024.

MDRIVER_FTL_MAX_BLOCK_AVAILABLE

The maximum number of blocks on each SafeFTL drive in the system. Changing this value may change the layout of pages, which affects compatibility with existing SafeFTL volumes. The default is 65536.

MDRIVER_FTL_MAX_LOG_BLOCK_AVAIL

The maximum number of LOG blocks on any flash drive. The maximum is 254, which is the default.

MDRIVER_FTL_MAX_CACHEFRAG

The maximum number of cacheable fragments on each flash drive. The optimum is 4, but if RAM usage is a concern you can reduce this to a minimum of 1. The default is 1.

MDRIVER_FTL_MAX_FREE_BLOCKS

The maximum number of free blocks on each SafeFTL drive in the system. This value must be <= MDRIVER_FTL_MAX_FREE_BLOCKS.

Changing this value modifies the format of the flash drive and may affect compatibility with existing SafeFTL volumes.

Set this value to ((n^2) -1) where *n* is the smallest integer that allows a valid *t_ftl_phy.n_freeblocks* value in all FTL drive configurations. The default is 254.

MDRIVER_FTL_DELETE_CONTENT

Set this to 1 to enable the SafeFTL functions that erase the original data when replacement blocks are allocated. The default is 0.

In normal operation, when new data is written to a logical address the original physical block containing the overwritten data is not affected. This original data cannot be accessed through normal API calls, but does remain in the physical block until it gets cleaned up at some point in the future. If you use this option, you know that data that is erased is truly removed from the system.

> **Note:** Enabling this option has a significant impact on performance.

This option also defines an interface for the FAT/SafeFAT file system to support the **f_deletecontent()** function, which also removes the data from the file when the file is deleted. To support this, set MDRIVER_FTL_DELETE_CONTENT to 1.

If you enable MDRIVER_FTL_DELETE_CONTENT while using a page size of only 512 bytes, set MDRIVER_FTL_MAX_FREE_BLOCKS to 27 (it cannot be more).

MDRIVER_FTL_IDLE_ERASE

The default is 0. When this is set to 1, FTL starts a low priority IDLE task which tries to erase blocks in the background. If the underlying physical layer driver (**t_ftl_driver**) supports SUSPEND ERASE and RESUME ERASE, this feature can speed up infrequent write operations.

> **Note:** The underlying physical layer driver must handle the situation when an erase operation is still in progress and a read/write request arrives.

MDRIVER_FTL_IDLE_ERASE_TASK_STACK_SIZE

The size of the stack available for the IDLE erase block task (see MDRIVER_FTL_IDLE_ERASE above). The default is 512.

MDRIVER_FTL_REWRITE

Keep the default of 1 to allow block rewrite during long read sequences. Blocks are scheduled for rewrite if the low level driver signals LL_REWRITE during a page read.

To prohibit block rewrite, set this to 0.

MDRIVER_FTL_REWRITE_DURING_READ

If this is 0 (the default), block rewrite is only performed during write operations when a LOG block merge did not take place. A LOG merge takes some time so the default setting prohibits the execution of a block rewrite to eliminate long delays from the caller's perspective. However, if LOG merge happens in every normal write cycle, this would totally disable the block rewrite feature. To avoid this, block rewrite is triggered after a number of consecutive LOG merge operations. This number is determined by the number of pages in a block.

MDRIVER_FTL_CACHE_WRITE

Set this to 1 if the NAND supports the "PROGRAM PAGE CACHE" command and the driver implements the **pf_writecache()** routine. FTL will speed up contiguous writes by calling **pf_writecache()** instead of the normal **pf_write()**.

MDRIVER_FTL_DATABLOCK_LOGGING

Set this to 1 to allow FTL to allocate LOG pages in data blocks. This improves random write speed at the cost of data area decrease.

When this option is set, *t_ftl_phy.n_logpageperblock* holds the number of LOG pages at the end of data blocks (see the file **api_mdriver_ftl.h**).

A driver may completely disable DATABLOCK_LOGGING for a volume by setting *n_logpageperblock* to 0.

MDRIVER_FTL_MULTILUN

The default is 0. Set this to 1 to enable cache write during merge in multi-LUN or multi-chip configurations. In this case FTL tries to allocate LOG blocks in such a way that read and write operations during a merge will use blocks in different LUNS. This makes it possible to use **pf_writecache()** instead of the normal **pf_write()**.

If you set this to 1, you must set MDRIVER_FTL_CACHE_WRITE too.

If you set MDRIVER_FTL_MULTILUN to 1, *t_ftl_phy.n_blockperlun* holds the number of blocks in each LUN (see the file **api_mdriver_ftl.h**). FTL assumes that each group of *n_blockperlun* blocks belongs to a different LUN. If *n_reservedblocks* is also set, the reserved blocks start at LUN #0.

MDRIVER_FTL_FAST_INIT

Set this to 1 to allow FTL to store MAP page locations into dedicated non-volatile memory (NVRAM) and use this information during **ml_init()**. This way **ml_init()** does not need to search the whole media for MAP blocks; this speeds up initialization of big NAND FTL volumes. The default is 0.

Note: The Background Merge feature only works with MDRIVER_FTL_DATABLOCK_LOGGING.

MDRIVER_FTL_BACKGROUND_MERGE

When this option is set to 1, full data blocks are scheduled for merge and a low priority background task performs the merge operations later. The default is 0.

MDRIVER_FTL_BM_QUEUE

The maximum number of blocks that can be scheduled for merge. The default is 16.

MDRIVER_FTL_BM_TASK_STACK_SIZE

The stack size of the low priority merge background task. The default is 512.

3.2 The as_ftldrive_init Flash Drive Structure

The file **src/config/config_mdriver_ftl.c** contains the flash drive structure for the flash drives which are to be made available to the SafeFTL.

The following is an example of this structure.

```
t_ftldrive_init  as_ftldrive_init[MDRIVER_FTL_MAX_DRIVE] =
{
  { nand_ram_init, 0U }
  , { ftldrv_w25n_init, 0U }
  , { ftldrv_w25n_init, 1U }
  , { ftl_nor_init, 0U }
  , { ftl_nor_init, 1U }
 };
```

Each available flash drive must have an entry in this table, specifying its initialization function and the parameter to pass to it in that function.

The flash drives are numbered from 0 to (MDRIVER_FTL_MAX_DRIVE-1). The index to this table is used to reference the flash drive. This differs for NAND and NOR drives as follows.

NAND drives

The number is the number of the instance of that driver type. So in the above example:

- there is one RAM NAND drive (its initialization function is **nand_ram_init()**) and this is numbered 0U.
- there are two Winbond W25N01GV NAND drives (their initialization function is **ftldrv_w25n_init()**) and these are numbered 0U and 1U.

NOR drives

Every NOR drive has **ftl_nor_init()** as its initialization function in the above table. The numbers that follow are indexes into a separate NOR table, the *as_ftl_nor_init[]* array, defined in the FTL NOR Base Flash Driver's own **config_mdriver_ftl.c** file. This is described in the *HCC FTL NOR Base Flash Driver User Guide*. The *as_ftl_nor_init[]* array, defines the different NOR flash types and this table has the same logic as above.

Note:

- Each NAND flash driver has its own initialization function that goes in this table. If the NAND flash driver can support multiple drives, each of those drives has an entry in this table.
- Each NOR flash driver connects to the generic NOR flash module. For each drive on each NOR flash module, there must be an entry in this table.

4 Application Programming Interface

This section documents the Application Programming Interface (API). It includes all the functions that are available to an application program.

4.1 Upper Layer Functions

Upper layer API functions are SafeFTL's interface to the user application, which is typically a file system. HCC Embedded's FAT file system supports sector sizes of 512, 2048, and 4096 bytes.

The file **src/media-drv/ftl/common/ftldrv.c** is a sample driver which uses the **ftl_xxx()** API functions. The interface reads or writes one or more sectors.

> **Note:** If your file system uses a sector size that is different from SafeFTL's sector size, modify the interface driver (in the **ftldrv.c** sample) to handle this.

The functions are the following:

Function	Description
ftl_init()	Initializes SafeFTL.
ftl_format()	Performs a low level format of a drive; all data are destroyed by this function.
ftl_initfunc()	Initializes a specified drive and returns to the caller a description of how to use the driver.
ftldrive_getmem()	Finds out how much memory the specified drive requires.
ftldrive_setmem()	Allocates the memory that the specified drive requires.
ftl_erase()	Erases all non-bad blocks on the media.
ftl_stats()	Gets statistics for the selected FTL volume.
ftl__get_blockinfo()	Gets wear and address information for selected data or management blocks.

ftl_init

Use this function to initialize SafeFTL.

> **Note:** Call this before using any other SafeFTL function.

Format

```
void ftl_init ( void )
```

Arguments

Argument
None.

Return values

Return value
None.

ftl_format

Use this function to perform a low level format of a drive; all data are destroyed by this function.

Note:

- Call this function once before using the device.
- Call this function only once in the lifetime of the drive. If a re-format is needed, erase all blocks in the flash and then call **ftl_format()** again.

A call of **ftl_initfunc()** will not succeed until this low level format of the media has been completed.

Note: The format may fail if the flash has been used by a system other than HCC's FTL. In this case erase the flash, but note the following:

- **Do not erase flash blocks that were marked as bad by the manufacturer.** The code that erases blocks should check whether each block is bad and only erase good blocks.
- If a system that used the flash did not preserve the manufacturer's bad block information, so that we cannot determine which blocks are good or bad, then regard the flash device as damaged.
- One approach is to erase all blocks and rely on the system managing bad blocks, but this is not recommended. Manufacturers specifically state that blocks marked bad at manufacture should never be used.

Format

```
t_ftl_ret ftl_format ( uint32_t drvnum )
```

Arguments

Argument	Description	Type
drvnum	The number of the drive to be formatted in the *as_ftldrive_init[]* structure.	uint32_t

Return Values

Return value	Description
0	Successful execution.
Else	Low level format failed.

Example

```
    /* Manufacturing/developing initialization */
    .
if (ftl_init())
{
    if (ftl_format(0))
    {
        /* Fatal error, device is not useable */
    }
}
    .
    /* Low level format is completed; SafeFTL can be used */
    .
```

ftl_initfunc

Use this function to initialize a specified drive and return to the caller a description of how to use the driver.

This is typically called by a file system. The description returned is a set of API functions conforming to the *HCC Media Driver Interface Specification*. The drive being called must be defined in the *as_ftldrive_init[]* flash drive structure.

Note:

- If this function finds that the flash drive is not formatted, it can automatically format it if the FTL_ALLOW_FORMAT bit on the *driver_param* argument is set. If this bit is not set, you must call any format operation that is performed.
- If a format is performed, all pages in the flash drive are erased.

Format

```
F_DRIVER * ftl_initfunc ( unsigned long driver_param )
```

Arguments

Argument	Description	Type
driver_param	The number of the drive entry in the *as_ftldrive_init[]* table. The topmost bit is reserved for specifying whether a format of the flash drive is allowed.	unsigned long

Return Values

Return value	Description
NULL	The drive number is out of range.
Else	A pointer to an F_DRIVER structure, as defined in the *HCC Media Driver Interface Specification*.

ftldrive_getmem

Use this function to find out how much memory the specified drive requires.

After this call, use **ftldrive_setmem()** to assign an area of memory of the required size to the drive.

Format

```
t_ftl_ret ftldrive_getmem (
    uint32_t     drvnum,
    uint32_t *   pi_memsize )
```

Arguments

Argument	Description	Type
drvnum	The number of the drive entry in the *as_ftldrive_init[]* table.	uint32_t
pi_memsize	Where to write the amount of memory required.	uint32_t *

Return Values

Return value	Description
0	Successful execution.
Else	The specified drive was not initialized correctly.

ftldrive_setmem

Use this function to allocate the memory that the specified drive requires.

Before calling this, find out the amount of memory needed by calling **ftldrive_getmem()**.

Format

```
t_ftl_ret ftldrive_setmem (
    uint32_t     drvnum,
    uint32_t *   pi_buf )
```

Arguments

Argument	Description	Type
drvnum	The number of the drive entry in the *as_ftldrive_init[]* table.	uint32_t
pi_buf	A pointer to the buffer to be used by the drive. This buffer must have the space required by the drive, the value previously obtained by calling **ftldrive_getmem()**.	uint32_t *

Return Values

Return value	Description
0	Successful execution.
Else	The drive has not been initialized.

ftl_erase

Use this function to erase all non-bad blocks on the media.

> **Note: ftl_erase()** is not thread safe; do not call it for an initialized FTL volume.

Format

```
t_ftl_ret ftl_erase( uint32_t drvnum )
```

Arguments

Argument	Description	Type
drvnum	The number of the drive entry in the *as_ftldrive_init[]* table. FTL references the given media using this drive number.	uint32_t

Return Values

Return value	Description
0	Successful execution.
1	Operation failed.

ftl_stats

Use this function to retrieve statistics for the selected FTL volume.

Format

```
t_ftl_ret ftl_stats (
    uint32_t        drvnum,
    t_ftl_stats *   p_stats )
```

Arguments

Argument	Description	Type
drvnum	The number of the drive entry in the *as_ftldrive_init[]* table. FTL references the given media using this drive number.	uint32_t
p_stats	A pointer to the *t_ftl_stats* structure to fill with statistics.	t_ftl_stats *

Return Values

Return value	Description
0	Successful execution.
Else	Operation failed.

ftl_get_blockinfo

Use this function to get wear and address information for selected data or management blocks.

Format

```
t_ftl_ret ftl_get_blockinfo (
    uint32_t            drvnum,
    uint8_t             b_data,
    uint32_t            start_block,
    uint32_t            n_blocks,
    t_ftl_blockinfo *   p_info )
```

Arguments

Argument	Description	Type
drvnum	The number of the drive entry in the *as_ftldrive_init[]* table. FTL references the given media using this drive number.	uint32_t
b_data	One of the following: • TRUE – collect info for data blocks • FALSE – collect info for management blocks.	uint8_t
start_block	The index of the first block to get info for.	uint32_t
n_blocks	The number of elements the *p_info* array has space for.	uint32_t
p_info	A pointer to the array to hold the information.	t_ftl_blockinfo *

Return Values

Return value	Description
0	Successful execution.
Else	Operation failed.

4.2 Lower Layer Functions

The low level **ll_xxx()** functions are called from higher level functions to communicate directly with the physical hardware device (or simulated device).

These functions must be implemented (ported) according to the operational requirements. Some functions can easily be accelerated via hardware (for example, by using internal DMA); other functions need complicated hardware solutions for hardware acceleration.

> **Note:** All pointers passed to the low level drivers for reading and writing data are 32-bit aligned.

The functions are the following:

Function	Description
xxx_init()	Gets access information for the drive, based on its entry in the flash drive structure.
ll_getphy()	Gets detailed information about the physical configuration of a drive.
ll_read()	Reads a page, including its spare area, from a physical block.
ll_readpart()	Reads a specified section of a page: the first half, the second half, the whole data page, or the spare area.
ll_write()	Writes a page of a physical block, including its spare area.
ll_writedouble()	Writes a page of a physical block, using two buffers.
ll_erase()	Erases a block.
ll_isbadblock()	Checks whether a physical block has been manufactured as a bad block.
ll_readonebyte()	Reads a single specified byte from the spare area of a page.

Spare Area Usage

The low level driver is responsible for guaranteeing the integrity of the data area of the pages in the NAND flash and also the "spare" area used by SafeFTL.

The size of the spare area on NAND flash devices varies. It is normally proportional to the size of the page and no less than 16 bytes. SafeFTL uses a logical spare area of 16 bytes, the last four of which may be overwritten by the low level driver for Error Correction Code (ECC) purposes. The ECC requirements vary, depending on the NAND device type. The size of the ECC used to protect the data also varies, depending on the NAND device type used, and also on any NAND controller generating and decoding the ECC.

SafeFTL requires the low level driver to be integrated in such a way that both of the following apply:

- The data area of a page is error free.
- The first 12 bytes of the logical spare area are error free.

xxx_init

Use this function to get access information for the drive, based on its entry in the flash drive structure.

> **Note:** This function is provided by every flash driver that SafeFTL is to use.

Format

```
t_ftl_ret ( *t_pf_ll_init )(
        uint32_t           drvnum,
        t_ftl_driver * *   pps_ftl_driver )
```

Arguments

Argument	Description	Type
drvnum	Number of flash drive in the *as_ftldrive_init* structure.	uint32_t
pps_ftl_driver	Pointer to a *t_ftl_driver* structure for the flash drive access information.	t_ftl_driver * *

Return Values

Return Value	Description
0	Successful execution.
Else	Operation failed.

ll_getphy

Use this function to get detailed information about the physical configuration of a drive.

The SafeFTL gets the function pointer to use for this when the flash driver init function **xxx_init()** is called.

Format

```
static uint16_t ll_getphy (
    t_ftl_driver *   ps_drv,
    t_ftl_phy *      ps_phy )
```

Arguments

Argument	Description	Type
ps_drv	A pointer to the *t_ftl_driver* structure of the required flash driver.	t_ftl_driver *
ps_phy	A pointer to the drive structure. This returns values giving the physical arrangement of the flash drive.	t_ftl_phy *

Return Values

Return value	Description
LL_OK	Successful execution.
LL_ERROR	The drive does not exist.

ll_read

Use this function to read a page, including its spare area, from a physical block.

The page data must be returned ECC-corrected. It is the responsibility of the lower level driver to handle the ECC requirements of the target device.

Format

```
static uint16_t ll_read (
    t_ftl_driver *   ps_drv,
    uint32_t         i_pba,
    uint16_t         i_ppo,
    uint8_t *        pc_dst )
```

Arguments

Argument	Description	Type
ps_drv	A pointer to the *t_ftl_driver* structure of the required flash driver.	t_ftl_driver *
i_pba	The physical block address.	uint32_t
i_ppo	The physical page offset within the block.	uint16_t
pc_dst	Where to write the data. This must be large enough to hold both the data and the spare area.	uint8_t *

Return Values

Return value	Description
LL_OK	Successful execution.
LL_ERASED	The page has been erased (all data and the spare area are set to 0xFF).
LL_REWRITE	The affected block should be scheduled for rewrite.
LL_ERROR	An error occurred.

ll_readpart

Use this function to read a specified section of a page: the first half of the page, the second half of the page, the whole data page, or the spare area.

> **Note:** The low level driver must resolve any ECC issues before returning the requested data.

Only the specified part of the page is stored in the specified buffer. The *i_index* parameter specifies which part of the page to read, as follows:

i_index	Area to read
LL_RP_1STHALF	First half of the page; read 0<= data < PAGE_SIZE/2.
LL_RP_2NDHALF	Second half of the page; read PAGE_SIZE/2 <= data<PAGE_SIZE.
LL_RP_DATA	The data page (PAGE_SIZE).
LL_RP_SPARE	The spare area; PAGE_SIZE <= data.

Format

```
static uint16_t ll_readpart (
    t_ftl_driver *   ps_drv,
    uint32_t         i_pba,
    uint16_t         i_ppo,
    uint8_t *        pc_dst,
    uint16_t         i_index )
```

Arguments

Argument	Description	Type
ps_drv	A pointer to the *t_ftl_driver* structure of the required flash driver.	t_ftl_driver *
i_pba	The physical block address.	uint32_t
i_ppo	The page offset in that block.	uint16_t
pc_dst	Where to write the requested data.	uint8_t *
i_index	The data that needs to be read (see the above table).	uint16_t

Return Values

Return value	Description
LL_OK	Successful execution.
LL_ERASED	The page is erased. The data and spare area are all set to 0xFF.
LL_ERROR	Operation failed.

ll_write

Use this function to write a page of a physical block, including its spare area.

Format

```
static uint16_t ll_write (
    t_ftl_driver *   ps_drv,
    uint32_t         i_pba,
    uint16_t         i_ppo,
    uint8_t *        pc_data,
    uint8_t *        pc_spare )
```

Arguments

Argument	Description	Type
ps_drv	A pointer to the *t_ftl_driver* structure of the required flash driver.	t_ftl_driver *
i_pba	The physical block address.	uint32_t
i_ppo	The page offset in the block.	uint16_t
pc_data	A pointer to the data to write.	uint8_t *
pc_spare	A pointer to the spare area data to write; see MAX_SPARE_SIZE in the file **ml_types.h**.	uint8_t *

Return Values

Return value	Description
LL_OK	Successful execution.
LL_ERROR	Operation failed.

ll_writedouble

Use this function to write a page of a physical block, using two buffers.

The two buffer parameters are pointers to the following areas:

- The first half of the data.
- The second half of the data, followed by the spare area.

Format

```
static uint16_t ll_writedouble (
    t_ftl_driver *   ps_drv,
    uint32_t         i_pba,
    uint16_t         i_ppo,
    uint8_t *        pc_buf0,
    uint8_t *        pc_buf1 )
```

Arguments

Argument	Description	Type
ps_drv	A pointer to the *t_ftl_driver* structure of the required flash driver.	t_ftl_driver *
i_pba	The physical block address.	uint32_t
i_ppo	The physical page in that block.	uint16_t
pc_buf0	A pointer to the first half of the data to write.	uint8_t *
pc_buf1	A pointer to the second half of the data to write and the spare area.	uint8_t *

Return Values

Return value	Description
LL_OK	Successful execution.
LL_ERROR	Operation failed.

ll_erase

Use this function to erase a block.

Format

```
static uint16_t ll_erase (
    t_ftl_driver *   ps_drv,
    uint32_t         i_pba )
```

Arguments

Argument	Description	Type
ps_drv	A pointer to the *t_ftl_driver* structure of the required flash driver.	t_ftl_driver *
i_pba	The physical block address of the block to erase.	uint32_t

Return Values

Return value	Description
LL_OK	Successful execution.
LL_ERROR	Operation failed.

ll_isbadblock

Use this function to check whether a physical block has been manufactured as a bad block.

The implementation of this function is device-specific because different devices use different methods to mark bad blocks. Check the documentation of the specific NAND flash device to determine how to implement the function.

Format

```
static uint16_t ll_isbadblock (
    t_ftl_driver *   ps_drv,
    uint32_t         i_pba )
```

Arguments

Argument	Description	Type
ps_drv	A pointer to the *t_ftl_driver* structure of the required flash driver.	t_ftl_driver *
i_pba	The physical block address.	uint32_t

Return Values

Return value	Description
0	The block is good.
1	The block is bad.

ll_readonebyte

Use this function to read a single specified byte from the spare area of a page.

> **Note:** No ECC check or correction is required on this byte as long as no more than a single bit error is possible on the byte. The byte that is read has internal one bit ECC protection.

Format

```
static uint16_t ll_readonebyte (
    t_ftl_driver *   ps_drv,
    uint32_t         i_pba,
    uint16_t         i_ppo,
    uint8_t          i_sparepos,
    uint8_t *        pc_byte )
```

Arguments

Argument	Description	Type
ps_drv	A pointer to the *t_ftl_driver* structure of the required flash driver.	t_ftl_driver *
i_pba	The physical block address.	uint32_t
i_ppo	The physical page offset within that block.	uint16_t
i_sparepos	The offset of the requested byte in the spare area.	uint8_t
pc_byte	Where to store the byte.	uint8_t *

Return Values

Return value	Description
LL_OK	Successful execution.
LL_ERROR	Operation failed.

4.3 Error Codes

If a function executes successfully, it returns with LL_OK, a value of zero. The following table shows the meaning of the error codes.

Option	Value	Description
LL_OK	0U	The requested operation was successful.
LL_ERASED	1U	The block is erased. All the data and the spare area are set to 0xFF.
LL_ERROR	2U	The requested operation failed.
LL_REWRITE	4U	If the physical layer driver reports LL_REWRITE after a read, the affected block should be scheduled for rewrite. The frequency of rewrite operations may be limited by the *t_ftl_phy.rewrite_interval* parameter on a per-volume basis.

4.4 Types and Definitions

t_ftl_phy

The *t_ftl_phy* structure is returned to the flash driver by the ll_getphy() function. The flash driver fills all the fields to inform the FTL how to use the flash drive.

Note: The HCC reference drivers contain tested settings for the value of *n_freeblocks*, *n_logblocks*, *n_mapblocks*, and *n_mapblock_shadow*.

Element	Type	Description
n_blocks	uint32_t	The actual number of erasable blocks in the target flash array.
n_pageperblock	uint32_t	The number of pages per erasable block.
sz_pagedata	uint32_t	The data area available on one page. Set this to the number of bytes (512, 2048, and so on), depending on the target flash device.
sz_pagetotal	uint32_t	The total size of the page, including the data and spare areas.
n_reservedblocks	uint32_t	The number of blocks at the start of the flash area that SafeFTL should not use.
n_freeblocks	uint32_t	The minimum value of *n_freeblocks* is *n_mapblock* * *n_mapblock_shadow* + *n_logblocks* + 9. This must be increased by the total possible number of bad blocks. For example, if the minimum is 15 for a given configuration and the actual device may develop 40 bad blocks during its lifetime, set *n_freeblocks* to 55. The number of available data blocks is *n_blocks* - *n_freeblocks*, where *n_blocks* is the total number of blocks on the NAND flash.

Element	Type	Description
n_logblocks	uint32_t	The number of log blocks. This influences write performance. The absolute minimum value is 1, the recommended minimum value is 2, and the recommended normal value is 4. If contiguous writes are interrupted by short random write sequences (as in a FAT file system), increasing *n_logblocks* increases the write speed as long as *n_logblocks* is small. However, the write speed improvement becomes smaller as *n_logblocks* increases. Increasing *n_logblocks* also increases *n_freeblocks* (see above), which decreases the number of available data blocks. Another approach is to make *n_logblocks* greater than the total number of different areas of the disk that are likely to be accessed in normal operation. So, for a standard FAT file system this could be 4 (the data area, the directory entry, and maybe two FAT areas) plus 1 for each file open and 1 for special cases. For HCC SafeFAT this should be made at least 6 and increased by two for each simultaneous file open.
n_mapblocks	uint32_t	The number of blocks used for mapping in the system. Increasing *n_mapblocks* improves system performance. The maximum useful number of map blocks that can be set is given by ((*Number_blocks**8/*sz_pagedata*)+1). *n_mapblocks* must be 2^n.
n_mapblock_shadow	uint32_t	The default and recommended setting is 3. The minimum setting is 1. SafeFTL is more efficient if more map shadow blocks are used, but each additional block reduces the number of free blocks in the system.
wear_static_limit	uint32_t	The maximum value that the difference between the maximum and minimum wear count can be. Set this to a value that is not statistically significant in the context of the number of erase write cycles supported by the flash device per block. Setting this to a small value causes unnecessary wear operations and is counter-productive. We recommend setting it to a value between 5% and 10% of the supported erase/write cycles.
wear_static_count	uint32_t	The number of merge operations allowed before static wear checking must be run. This is an internal counter that determines how often the wear state of blocks is checked. The checking is performed in small steps to minimize the effect on overall performance. It is only performed if the *wear_static_limit* has been reached between two blocks. A value of 1024 works well in most configurations.

Element	Type	Description
rewrite_interval	uint32_t	The number of read/write operations allowed before a rewrite check must be run. It is not mandatory for drivers to fill this parameter; that is, it is initialized to 0.
n_logpageperblock		The number of LOG pages allocated at the end of data blocks. This is only used if MDRIVER_FTL_DATABLOCK_LOGGING is set.
n_blockperlun		The number of LUNs or chips. A value of 0 or *n_blocks* disables multi-LUN support for the actual device. This is only used if MDRIVER_FTL_MULTILUN is set.

t_ftl_driver

The *t_ftl_driver* structure is passed to the flash driver in the xxx_init() call. The flash driver fills in all the function pointers for the FTL to use when using the flash drive.

Element	Type	Description
user_data	uint32_t	User-defined data. For example, this may be used to identify a specific flash drive.
pf_getphy	(* pf_getphy)	A pointer to the ll_getphy function, used to get the physical characteristics of the flash drive.
pf_read	(* pf_read)	A pointer to the ll_read function, used to read a page on the flash drive.
pf_readpart	(* pf_readpart)	A pointer to the ll_readpart function, used to read part of a page on the flash drive.
pf_write	(* pf_write)	A pointer to the ll_write function, used to write a page to the flash drive.
pf_writedouble	(* pf_writedouble)	A pointer to the ll_writedouble function, used to write a page from two buffers to the flash drive.
pf_erase	(* pf_erase)	A pointer to the ll_erase function, used to erase a block on the flash drive.
pf_isbadblock	(* pf_isbadblock)	A pointer to the ll_isbadblock function, used to check whether a specific block is bad.
pf_readonebyte	(* pf_readonebyte)	A pointer to the ll_readonebyte function, used to read a single byte from the flash drive.

t_ftldrive_init

The *t_ftldrive_init* structure is used in the FTL drive configuration table.

Element	Type	Description
pf_ll_init	t_pf_ll_init	A pointer to the initialization function for the flash drive.
ll_param	uint32_t	The driver parameter to use when the **pf_ll_init()** function for the specified flash drive is called.

Page Read Parameters

Use the following to specify which part of a page to read.

Element	Description
LL_RP_1STHALF	Read the first half of the data area of the page.
LL_RP_2NDHALF	Read the second half of the data area of the page.
LL_RP_DATA	Read the whole data area of the page.
LL_RP_SPARE	Read the spare area of the page.

t_ftl_blockinfo

The *t_ftl_blockinfo* structure holds wear and address information for data or management blocks:

Element	Type	Description
lba	uint32_t	For data blocks this is the logical block address. For management blocks, this is -1.
pba	uint32_t	The physical block address.
wear	uint32_t	The number of erase cycles.

t_ftl_stats

The *t_ftl_stats* structure holds wear and address information for data or management blocks:

Element	Type	Description
n_blocks_total	uint32_t	The total number of blocks.
n_blocks_reserved	uint32_t	The number of blocks reserved. That is, blocks not used by the FTL.
n_blocks_data	uint32_t	The number of blocks available for data storage. That is, the logical data array.
n_blocks_mgmt	uint32_t	The total number of management blocks, including map, log, free, and bad blocks.
n_blocks_map	uint32_t	The number of map blocks.
n_blocks_bad	uint32_t	The number of bad blocks.
page_per_block	uint32_t	The number of pages per block.
bytes_per_page	uint32_t	The number of bytes per page.

5 Integration

This section describes all aspects of the SafeFTL module that require integration with your target project. This includes porting and configuration of external resources.

5.1 OS Abstraction Layer

All HCC modules use the OS Abstraction Layer (OAL) that allows the module to run seamlessly with a wide variety of RTOSes, or without an RTOS.

SafeFTL uses the following OAL components:

OAL Resource	Number Required
Tasks	0
Mutexes	1
Events	0

5.2 PSP Porting

The Platform Support Package (PSP) is designed to hold all platform-specific functionality, either because it relies on specific features of a target system, or because this provides the most efficient or flexible solution for the developer.

SafeFTL makes use of the following standard PSP functions:

Function	Package	Element	Description
psp_memcpy()	psp_base	psp_string	Copies a block of memory. The result is a binary copy of the data.
psp_memset()	psp_base	psp_string	Sets the specified area of memory to the defined value.