High Performance
Real-Time Operating Systems

Flash Translation Layer

User's Guide and
Reference Manual
Support

Copyright

Disclaimer

SCIOPTA Systems AG, makes no representations or warranties with respect to the contents hereof and specifically disclaims any implied warranties of merchantability of fitness for any particular purpose. Further, SCIOPTA Systems AG, reserves the right to revise this publication and to make changes from time to time in the contents hereof without obligation to SCIOPTA Systems AG to notify any person of such revision or changes.

Trademark

SCIOPTA is a registered trademark of SCIOPTA Systems AG.

EU Headquarters

SCIOPTA Systems GmbH
Hauptstrasse 293
79576 Weil am Rhein
Germany
Tel. +49 7621 940 919 0
Fax +49 7621 940 919 19
email: sales@sciopta.com
www.sciopta.com

Corporate Headquarters

SCIOPTA Systems AG
Fiechthagstrasse 19
4103 Bottmingen
Switzerland
Tel. +41 61 423 10 62
Fax +41 61 423 10 63
email: sales@sciopta.com
www.sciopta.com

Document No. S13243RL1

1 Table of Contents

2 SCIOPTA Real-Time Operating System

2.1 Introduction

SCIOPTA is a high performance fully pre-emptive real-time operating system for hard real-time application available for many target platforms.

Available modules:

- Pre-emptive Multitasking Real-Time Kernel
- SCIOPTA Memory Management System - Support for MMU/MPU
- Board Support Packages
- IPS - Internet Protocols (TCP/IP) including IPS Applications (Web Server, TFTP, FTP, DNS, DHCP, Telnet and SMTP)
- FAT File System
- Flash File System, NOR and NAND
- Universal Serial Bus, USB Device
- Universal Serial Bus, USB Host
- DRUID - System Level Debugger including kernel awareness packages for source debuggers
- SCIOPTA PEG - Embedded GUI
- CONNECTOR - support for distributed multi-CPU systems
- SCAPI - SCIOPTA API for Windows or LINUX host
- SCSIM - SCIOPTA Simulator

SCIOPTA Real-Time Operating System contains design objects such as SCIOPTA modules, processes, messages and message pools. SCIOPTA is designed on a message based architecture allowing direct message passing between processes. Messages are mainly used for interprocess communication and synchronization. SCIOPTA messages are stored and maintained in memory pools. The memory pool manager is designed for high performance and memory fragmentation is avoided. Processes can be grouped in SCIOPTA modules, which allows you to design a very modular system. Modules can be static or created and killed during runtime as a whole. SCIOPTA modules can be used to encapsulate whole system blocks (such as a communication stack) and protect them from other modules in the system.

The SCIOPTA Real-Time Kernel has a very high performance. The SCIOPTA architecture is specifically designed to provide excellent real-time performance and small size. Internal data structures, memory management, interprocess communication and time management are highly optimized. SCIOPTA Real-Time kernels will also run on small single-chip devices without MMU.

2.2 CPU Family

SCIOPTA is delivered for a specific CPU Family such as: ARM®7/9, ARM®11, ARM® Cortex-M™, ARM® Cortex™-R, ARM® Cortex™-A, Renesas RX, Freescale™ PowerPC, apm PowerPC, Freescale™ ColdFire and Marvell Xscale.

Please consult the latest version of the SCIOPTA Price List for the complete list.

2.3 About This Manual

The **SCIOPTA** Real-time Operating System is a message based RTOS and is therefore very well suited for distributed multi-CPU systems.

The purpose of this **Flash Translation Layer - User´s Guide and Reference Manual** is to give all needed information how to use the **SCIOPTA** Flash Translation Layer.

Please see also the other **SCIOPTA** manuals, mainly the **SCIOPTA** - Kernel, User's Guide and Reference Manual.

This manual includes only target processor independent information. All target processor related information can be found in the **SCIOPTA - Target Manual** which is different for each **SCIOPTA** supported processor family and includes:

- Installation information
- Getting started examples
- Description of the system configuration (SCONF tool)
- Information about the system building procedures
- Description of the board support packages (BSP)
- List of distributed files
- Release notes and version history

3 Introduction

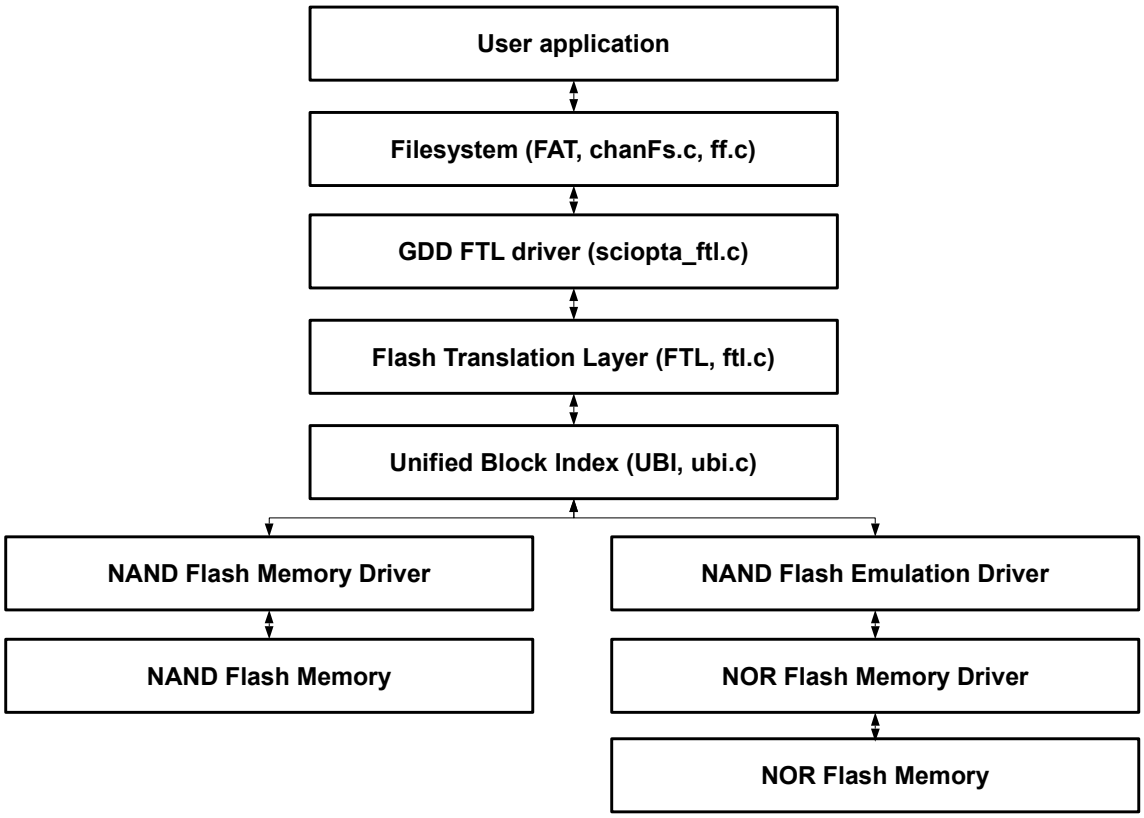
NAND flash memory is an array of blocks of fixed sizes. Each block is divided into a number of physical pages. Each physical page has got additional memory area, called the “spare area” which is available to the user. In addition, the physical page may divided into smaller virtual pages if flash device supports them.

Physical page may be programmed only after entire block is erased. Each block erase causes the block to get increasingly worn off. Each block has got erase count limit defined by the manufacturer, after which the data may loose integrity. This happens if a physical page is reprogrammed very often, which means that the block has to be erased each time.

The Unified Block Index (UBI) and Flash Translation Layer (FTL) together free the user application (usually filesystem) from erasing blocks and reprogramming pages. Following diagram shows how the UBI and FTL align with other part of the system.

FTL is designed to work with NAND flash devices, however it is possible to add an adaptation layer, which emulates NAND flash memory on top of NOR flash memory. Refer to (7) for details about implementing adaptation layer.

A logical page size of FTL layer is the same as physical page size of NAND flash device.



4 Unified Block Index (UBI)

4.1 Introduction

NAND flash memory is an array of blocks of fixed sizes. Each block is divided into a number of physical pages. Each physical page has got additional memory area, called the “spare area” which is available to the user. In addition, the physical page may be divided into smaller virtual pages if flash device supports them.

Physical page can be programmed only after entire block is erased. Each block erase causes the block to get increasingly worn off. Each block has got erase count limit defined by the manufacturer, after which the data may loose integrity. This happens if a physical page gets reprogrammed very often, which means that the block has to be erased each time.

The UBI (Unified Block Index) is a layer which provides an abstraction over NAND flash memory blocks. It takes a series of N **physical blocks** of NAND flash memory and presents them to the upper layer as a series of M **logical blocks** (where M < N). When upper layer erases a logical block, it gets replaced with another block from the pool of free blocks. The UBI runs background periodic procedure, called wear-leveling. The purpose of this procedure is to find the most worn off block in the free blocks pool and exchange it with the least worn off block of the blocks currently in use. Executing wear-leveling procedure periodically, assures that blocks are evenly worn-off, thus freeing the upper layer from worrying about wearing off blocks in the NAND flash memory.

4.2 Requirements for NAND flash driver

4.2.1 Introduction

Every NAND flash driver must follow the following set of requirements to be compatible with UBI.

- Support for SDD_DEV_OPEN (read and write) message to open the device.
- Support for SDD_DEV_CLOSE message to close the device
- Support for SDD_DEV_READ message to read data area of flash. The driver must accept any read size as long as it is aligned to virtual page size and read operation does not go beyond flash size.
- Support for SDD_DEV_WRITE message to write data area of flash. The driver must accept any write size as long as it is aligned to virtual page size and write operation does not go beyond flash size.
- Support for SDD_FILE_SEEK message to move current read/write position. The driver must accept any offset as long as it is aligned to virtual page size and does not go beyond flash size.
- Support for SDD_DEV_IOCTL message to perform certain commands on flash device. Refer to (4.2.3) for the list of ioctl commands, which must be supported.

4.2.2 Reading data and spare area

The driver must return error **EFSBITFLIPPED**, if there was a bit error when reading data or spare area and the bit error was corrected by ECC. If **EFSBITFLIPPED** is returned, the ECC corrected data must also be returned. For more informations about handling bitflips, refer to (4.3.9).

The driver must always return read data, even if ECC uncorrectable error has occurred.

4.2.3 Ioctl commands

Following ioctl commands must be supported by the flash device driver.

“Required size of arg” in following commands means that **ioctl.outlineArg** or **ioctl.inlineArg** must be at least of the size specified.

4.2.3.1 Getting number of physical blocks

Command:	ioctl.cmd = NANDFLASH_IOCTL_GET_NUM_BLOCKS
Required size of arg:	sizeof(uint32_t)
Description:	This command returns back the number of physical blocks available in the flash device. The value is stored in outlineArg/inlineArg as **uint32_t** type.

4.2.3.2 Getting physical block size

Command:	ioctl.cmd = NANDFLASH_IOCTL_GET_PHY_BLOCK_SIZE
Required size of arg:	sizeof(uint32_t)
Description:	This command returns back the physical block size of flash. For NAND flash this is a block size excluding spare area. For NOR flash this entire block size (as there is no spare area). The value is stored in outlineArg/inlineArg as **uint32_t** type.

4.2.3.3 Getting logical block size

Command:	ioctl.cmd = NANDFLASH_IOCTL_GET_LOG_BLOCK_SIZE
Required size of arg:	sizeof(uint32_t)
Description:	This command returns back the logical block size of flash. For NAND flash this is a the same as physical block size excluding spare area. For NOR flash this is a physical block size minus the spare area created artificially by the driver to emulate NAND flash. The value is stored in outlineArg/inlineArg as **uint32_t** type.

4.2.3.4 Getting physical page size

Command:	ioctl.cmd = NANDFLASH_IOCTL_GET_PHY_PAGE_SIZE
Required size of arg:	sizeof(uint32_t)
Description:	This command returns back the physical page size. The value is stored in outlineArg/inlineArg as **uint32_t** type.

4.2.3.5 Getting virtual page size

Command:	ioctl.cmd = NANDFLASH_IOCTL_GET_VIRT_PAGE_SIZE
Required size of arg:	sizeof(uint32_t)
Description:	This command returns back the virtual page size. The value is stored in outlineArg/inlineArg as **uint32_t** type.

4.2.3.6 Getting spare area size

Command:	ioctl.cmd = NANDFLASH_IOCTL_GET_SPARE_AREA_SIZE
Required size of arg:	sizeof(uint32_t)
Description:	This command returns back the spare area size available to the user. The value is stored in outlineArg/inlineArg as **uint32_t** type.

4.2.3.7 Erasing physical block

Command:	ioctl.cmd = NANDFLASH_IOCTL_ERASE_BLOCK
Required size of arg:	sizeof(uint32_t)
Description:	This command erases the physical block. The zero-based physical block index should be passed to flash device driver in outlineArg/inlineArg as **uint32_t** type.

4.2.3.8 Erasing physical block only if dirty (optional)

Command:	ioctl.cmd = NANDFLASH_IOCTL_ERASE_BLOCK_IF_DIRTY
Required size of arg:	sizeof(uint32_t)
Description:	This command erases the physical block only if it is dirty (not contains only FF values). The zero-based physical block index should be passed to flash device driver in outlineArg/inlineArg as **uint32_t** type.

This ioctl command is optional. **If this command will not be implemented, a driver must return ENOTSUP error code.**

4.2.3.9 Testing for bad physical block

Command:	ioctl.cmd = NANDFLASH_IOCTL_CHECK_BLOCK_BAD
Required size of arg:	sizeof(uint32_t)
Description:	This command checks if physical blocks is marked as bad block. The zero-based physical block index should be passed to flash device driver in outlineArg/inlineArg as **uint32_t** type. The bad block status is stored by device driver in outlineArg/inlineArg as **uint32_t** type. If status is non-zero, the block is bad.

4.2.3.10 Marking physical block as bad

Command:	ioctl.cmd = NANDFLASH_IOCTL_MARK_BLOCK_AS_BAD
Required size of arg:	sizeof(uint32_t)
Description:	This commands marks physical block as bad block. The zero-based physical block index should be passed to flash device driver in outlineArg/inlineArg as **uint32_t** type.

4.2.3.11 Testing all physical blocks for bad block marks

Command:	ioctl.cmd = NANDFLASH_IOCTL_GET_BAD_BLOCK_TABLE
Required size of arg:	Number of physical blocks divided by 8
Description:	This command checks all physical blocks for bad block marks. The result is stored in outlineArg/inlineArg. Each block occupies one bit. Block number 0 corresponds to LSB bit of byte 0 of outlineArg/inlineArg. If bit is set, the block is marked bad.

4.2.3.12 Reading spare area

Command:	ioctl.cmd = NANDFLASH_IOCTL_READ_SPARE
Required size of arg:	Spare area size returned by ioctl command NANDFLASH_IOCTL_GET_SPARE_AREA_SIZE.
Description:	This command reads spare area of physical page. The physical page number is determined from the current read/write pointer, which can be moved by using SDD_FILE_SEEK message. It is enough for the pointer to point to any place within physical page. Moving it to the beginning of physical page is not necessary.

4.2.3.13 Writing spare area

Command:	ioctl.cmd = NANDFLASH_IOCTL_WRITE_SPARE
Required size of arg:	Spare area size returned by ioctl command NANDFLASH_IOCTL_GET_SPARE_AREA_SIZE.
Description:	This command writes spare area of physical page. The physical page number is determined from the current read/write pointer, which can be moved by using SDD_FILE_SEEK message.

It is enough for the pointer to point to any place within physical page. Moving it to the beginning of physical page is not necessary.

4.3 Using UBI

4.3.1 Formatting flash

NAND flash memory can be used by UBI after being formatted. To format flash memory use function **UBI_FormatOnly** (4.6.4).

4.3.2 Initializing UBI

To initialize UBI with specific flash device, use function **UBI_Initialize** (4.6.13). This function checks if data structures are correct and from this point on the flash memory can be used by the upper layer.

4.3.3 Getting flash parameters

To get certain parameters of flash memory available to the upper layer, use following functions:

- **UBI_GetBlocksCount** (4.6.5), for number of logical blocks available to the upper layer,
- **UBI_GetPhyPagesPerBlock** (4.6.8), for number of physical pages per logical block,
- **UBI_GetPhyPageSize** (4.6.6), for size of physical page,
- **UBI_GetVirtPageSize** (4.6.10), for size of virtual page,
- **UBI_GetSpareSize** (4.6.9), for size of spare area available to the user.

4.3.4 Running UBI background task

UBI background task is responsible for running wear-leveling procedure and exchanging blocks in case of CRC errors caused by memory bitflips. Call function **UBI_Background** to execute these tasks.

One **UBI_Background** call services one most worn off block from the pool of free blocks, exchanging it with the least worn block of the ones currently in use.

There are three possible strategies to call **UBI_Background**:

1. Call **UBI_Background** periodically. The wear-leveling will do its work only when it is needed. Each call will also check for memory bitflips.
2. Keep calling **UBI_Background** until flash health indicators fall to the safe level. See 4.3.10 for details about checking flash health.
3. Call **UBI_Background** not more often than after every block erase.

Strategies 2 and 3 have a drawback, because they may miss bitflip events. Bitflips events occur when reading flash memory. The UBI reacts to bitflip events in background task, but when power is off, the information about these events is lost. Bitflip event may not happen again when particular page is read again, but it does not mean, the data will not loose integrity in the future. For details about bitflips refer to (4.3.9).

4.3.5 Writing flash pages

To write flash physical page, use function **UBI_WritePhyPage** (4.6.18).

To write flash virtual page, use function **UBI_WriteVirtPage** (4.6.19).

4.3.6 Reading flash pages

To read flash physical page, use function **UBI_ReadPhyPage** (4.6.14).

To read flash virtual page, use function **UBI_ReadVirtPage** (4.6.15).

4.3.7 Erasing logical blocks

To erase flash memory logical block, use function **UBI_EraseBlock** (4.6.3).

4.3.8 Enabling writes verification

To enable or disable flash writes verification, use function **UBI_VerifyWritesEnable** (4.6.17).

4.3.9 Bitflips

Depending on its implementation and capabilities, an underlying NAND driver can detect and correct bit errors in data or spare area read. The UBI has got no knowledge about this mechanism, but it can handle EFSBITFLIPPED status, if returned by the NAND driver when bit error is detected. Receiving EFSBITFLIPPED from the driver is a signal for the UBI, that particular block may cause problems. Background task will exchange this block with one from the pool of free blocks for its later erase and reuse. Refer to (4.3.4) for details about running background task.

4.3.10 Getting and interpreting flash health

To get flash memory health status, use function **UBI_Health** (4.6.12).

```
typedef struct ubi_health_s {
  int       warning;
  uint32_t  rootBlks;
  uint32_t  badRootBlks;
  uint32_t  bitFlips;
  uint32_t  dataBlks;
  uint32_t  freeDataBlks;
  uint32_t  badDataBlks;
  uint32_t  eraseCountMax;
  uint32_t  eraseCountAvg;
} ubi_health_t;
```

warning	Flash health warning
	If !=0, indicates that either number of good root blocks or number of free data blocks fell below safe level and if next block becomes bad, the flash will no longer be usable for writing. It is recommended to backup data in this case and re-format flash.
rootBlks	Root blocks total count
	Indicates total number of blocks allocated for UBI root structures (good and bad blocks). Refer to (4.5.5.1) for informations about configuring the number of root blocks.
badRootBlks	Bad root blocks count
	Indicates how many of blocks allocated for UBI root structures are bad.
bitFlips	Detected bitflips count
	Indicates how many bitflips events occurred since formatting. Refer to (4.3.9) for informations about bitflips.

dataBlks	Data blocks count
	Indicates the number of logical data blocks available to the upper layer.
freeDataBlks	Free data blocks count
	Indicates the number of free data blocks. Refer to (4.5.5.1) for informations about configuring the number of free data blocks.
badDataBlks	Bad data blocks count
	Indicates the number of bad data blocks.
eraseCountMax	Maximum erase count
	Indicates how many times the most worn off block has been erased.
eraseCountAvg	Average erase count
	Indicates average wear off of all blocks excluding root blocks.

4.4 Power Loss Recovery

UBI deals with power fails by attempting to write the same data structures as before the power fail. This is done to make the UBI structures integral. The underlying flash memory should allow writing the same data, and UBI verifies written data. However unlikely it is, if UBI fails to write data in Power Loss Recovery procedure, it will switch to read-only mode, allowing upper layers to access the data stored in flash.

4.5 Configuration

4.5.1 Introduction

The UBI uses header “ubi_cfg.h” to read the user configuration. A template configuration header can be found in following location:

<installation_folder>\sciopta\<version>\include\ftl\ubi_cfg_template.h

4.5.2 Configuring UBI for flash device

The UBI divides flash memory into three main areas:

- Boot area – reserved blocks, which UBI will not modify in any way. These blocks can be used by user for storing target's boot data. Refer to (4.5.5.1) for informations about configuring the size of boot area.
- Root area – these blocks are used by the UBI to store this part of its structures which changes least frequently. This means they will be rarely erased and thus least exposed to wearing off. Refer to (4.5.5.1) for informations about configuring the size of root area.
- Data area (catalogue, directories, data blocks and free blocks) – these blocks will be frequently erased. When UBI formats this area, it checks first which blocks are bad, and these blocks will not be used. Blocks which are left are divided initially in the following order: catalogue, directory blocks, each followed by data blocks which belong to it, and finally free blocks.

When configuring UBI for particular flash device, user must choose between number of blocks available to the upper layer and lifespan of a flash memory. If flash memory will be frequently written it is highly recommended to increase the size of root area (4.5.5.1) and percentage of free blocks in data area (4.5.5.1) at the cost of flash size available to the upper layer.

4.5.2.1 Choosing root area size

For the root area the size does not need to be too large to assure it's long lifespan. Assuming that each block contains N physical pages, one root block will be erased after every N * N of block erase requests from the upper layer. Following formula is an approximation of how many times (on average) each **data block** will be erased if each **root block** was erased once.

~ (root_area_size * N * N) / M,

where: N – number of physical pages in block, M – number of blocks in flash

For example for flash memory with 512 blocks, each containing 64 physical pages, all root blocks will be erased once whilst all other blocks will be erased approximately 24 times.

4.5.2.2 Choosing free blocks percentage

It is required to reserve more than 3 blocks as free blocks for the UBI. The number of free blocks is chosen as a percentage of available blocks. The minimal percentage value is 1%, which may result in 4 or more free blocks, depending on the size of flash memory. If not, the percentage must be increased.

If any of blocks other than root blocks becomes bad, number of free blocks may, at some point, fall down from 4 to 3. The UBI_Health function (4.6.12) will start returning health warning status. This is because if any other block becomes bad again, the UBI layer will not be writeable anymore. User should check health status periodically and act accordingly when health warning is reported.

Because of the reason described above, it is highly recommended to choose a number of free blocks bigger than minimal, to get longer lifespan before another flash memory format is required.

4.5.3 Configuring for minimal RAM memory usage

To configure UBI for minimal memory usage, set following setting:

UBI_DIR_CACHE_SIZE = 1 (refer to 4.5.5.4)

4.5.4 Configuring for maximum performance

To configure UBI for maximum performance, set following setting:

UBI_DIR_CACHE_SIZE = 0xFFFFFFFF (refer to 4.5.5.4)

4.5.5 Options reference

4.5.5.1 Flash memory layout

These options specify how the flash device is divided to Boot Area, Root Area and Data Area.

Option name:	UBI_AREA_BOOT_SIZE
Valid values:	0, 1, 2, ...
Description:	This option specifies how many physical blocks starting from the beginning of flash device are reserved and should not be used by UBI. **Changing this option requires reformatting the flash memory to create new FTL structures.**

Option name:	UBI_AREA_ROOT_SIZE
Valid values:	3, 4, 5, 6, ...
Description:	This option specifies how many physical blocks after Boot Area (option UBI_AREA_BOOT_SIZE) should be used by UBI to store it's structures. During the usage of UBI, blocks in this area may be marked bad due to the erase/write errors. If number of good blocks in this area falls down to 2, the UBI_Health function (4.6.12) will return a health warning. Refer to (4.3.10) for informations about getting and interpreting flash health. Bad blocks which are already in the specified area will decrease the value specified by this option. If number of remaining good blocks is less than 3, the flash device cannot be used by UBI and error will be returned when using UBI_FormatOnly function (4.6.4). In this case increasing the value of this option should allocate at least 3 good blocks. **Changing this option requires reformatting the flash memory to create new FTL structures.**

Option name:	UBI_FREE_BLOCKS_PERCENTAGE
Valid values:	Any number in range 1 to 99, inclusive.
Description:	This option specifies a percentage of data blocks, which should form a pool of free blocks. **Changing this option requires reformatting the flash memory to create new FTL structures.**

4.5.5.2 Block erasing policy

Option name:	UBI_ERASE_BLOCK_ONLY_IF_DIRTY
Valid values:	0 – Always erase 1 – Erase only if block is dirty (does not contain only FFs)
Description:	This option enables optional feature of flash drivers, which allows to speed up block erasing when formatting flash memory. The feature of the flash driver first checks whether a block contains only FF values. If yes, block is not erased. **WARNING**: This feature may be dangerous if block of NAND flash went bad, but was not yet flagged as a bad one. Without erasing, UBI will not know that such block is bad. In case of NOR memory a block erasing may end up with error, which is an indication for UBI to treat such block as a bad one. In both cases (NAND and NOR) turning this option on may lead to later data corruption.

4.5.5.3 Wear-leveling

Option name:	UBI_WL_THRESHOLD
Valid values:	1, 2, 3, …, 0xFFFFFFFF
Description:	This option selects the threshold of when the wear-leveling procedure should exchange the most worn off block with the least worn off block. The exchange happens if difference between erase count for those blocks is larger then UBI_WL_THRESHOLD.

Option name:	UBI_ERASE_COUNT_MAX
Valid values:	1000, 1001, 1002, …, 0xFFFFFFFF
Description:	This option sets the maximum erase count for each physical block. The UBI increases the erase count whenever a physical block is erased. If erase count of a physical block reaches this limit, the block is removed from free blocks pool and will no longer be used.

4.5.5.4 Caching

Option name:	UBI_DIR_CACHE_SIZE
Valid values:	1, 2, 3, …, 0xFFFFFFFF
Description:	This option selects the size of cache for UBI structures. The cache size is in units of physical page size. The cache actually allocated is never bigger than required. Thus setting this option to 0xFFFFFFFF assures, that UBI works with maximum performance, because all structures are cached.

4.5.5.5 Read-only

Option name:	UBI_READ_ONLY
Valid values:	0 – Read only disabled 1 – Read only enabled
Description:	If this option is enabled, the UBI is compiled in read-only mode (decreases code footprint).

4.5.5.6 Support for virtual pages

Option name:	UBI_SUPPORT_FLASH_VIRTUAL_PAGES
Valid values:	0 – Support for virtual pages disabled 1 – Support for virtual pages enabled
Description:	If this option is enabled, the UBI is compiled with support for flash device virtual pages.

4.5.5.7 Trap interface

Option name:	UBI_USE_TRAP_INTERFACE
Valid values:	0 – Disabled 1 – Enabled
Description:	If this option is enabled, Sciopta trap interface will be used by UBI sources.

4.5.5.8 Debug log

Option name:	UBI_LOG_LEVEL
Valid values:	0 – log is disabled 1 – only error messages are printed 2 – diagnostic and error messages are printed
Description:	This option enables sending UBI log messages to Log Deamon.

4.6 Function interface reference

4.6.1 UBI_Background

This function runs UBI background tasks. Refer to (4.3.4) for detailed informations about calling this function.

UBI_Initialize (4.6.13) must be successfully called prior to calling this function.

4.6.1.1 Syntax

```
ubi_error_t UBI_Background(
  ubi_t  *ubi
);
```

4.6.1.2 Parameters

ubi	UBI handle
	Pointer to the structure holding UBI state.

4.6.1.3 Return value

If the function succeeds the return value is **UBI_ERR_SUCCESS**.

If the function fails, it returns an error code. Refer to (4.7) for error reference.

4.6.2 UBI_Deinitialize

This function deinitializes the UBI by closing NAND flash device and releasing all allocated memory.

UBI_Initialize (4.6.13) must be successfully called prior to calling this function.

4.6.2.1 Syntax

```
ubi_error_t UBI_Deinitialize(
  ubi_t  *ubi
);
```

4.6.2.2 Parameters

ubi	UBI handle
	Pointer to the structure holding UBI state.

4.6.2.3 Return value

If the function succeeds the return value is **UBI_ERR_SUCCESS**.

If the function fails, it returns an error code. Refer to (4.7) for error reference.

4.6.3 UBI_EraseBlock

This function erases a logical block.

UBI_Initialize (4.6.13) must be successfully called prior to calling this function.

4.6.3.1 Syntax

```
ubi_error_t UBI_EraseBlock(
  ubi_t     *ubi,
  uint32_t  block
);
```

4.6.3.2 Parameters

ubi	UBI handle
	Pointer to the structure holding UBI state.
block	Logical block index
	Specifies zero-based logical block index to erase.

4.6.3.3 Return value

If the function succeeds the return value is **UBI_ERR_SUCCESS**.

If the function fails, it returns an error code. Refer to (4.7) for error reference.

4.6.4 UBI_FormatOnly

This function opens a NAND flash device, formats it and closes the device.

UBI_Initialize (4.6.13) must be successfully called prior to calling this function.

4.6.4.1 Syntax

```
ubi_error_t UBI_FormatOnly(
  ubi_t       *ubi,
  const char  *device_name
);
```

4.6.4.2 Parameters

ubi	UBI handle
	Pointer to the structure holding UBI state.
device_name	NAND flash device name
	Specifies NAND flash device name as it is registered to Device Manager.

4.6.4.3 Return value

If the function succeeds the return value is **UBI_ERR_SUCCESS**.

If the function fails, it returns an error code. Refer to (4.7) for error reference.

4.6.5 UBI_GetBlocksCount

This function returns the number of logical blocks available.

UBI_Initialize (4.6.13) must be successfully called prior to calling this function.

4.6.5.1 Syntax

```
ubi_error_t UBI_GetBlocksCount(
  ubi_t     *ubi,
  uint32_t  *blocks_count
);
```

4.6.5.2 Parameters

ubi	UBI handle
	Pointer to the structure holding UBI state.
blocks_count	Logical blocks count
	Returned value is a number of logical blocks available to the upper layer.

4.6.5.3 Return value

If the function succeeds the return value is **UBI_ERR_SUCCESS**.

If the function fails, it returns an error code. Refer to (4.7) for error reference.

4.6.6 UBI_GetPhyPageSize

This function returns the size of a physical page.

UBI_Initialize (4.6.13) must be successfully called prior to calling this function.

4.6.6.1 Syntax

```
ubi_error_t UBI_GetPhyPageSize(
  ubi_t      *ubi,
  uint32_t   *phy_page_size
);
```

4.6.6.2 Parameters

ubi	UBI handle
	Pointer to the structure holding UBI state.
phy_page_size	Physical page size
	Returned value is a physical page size in bytes.

4.6.6.3 Return value

If the function succeeds the return value is **UBI_ERR_SUCCESS**.

If the function fails, it returns an error code. Refer to (4.7) for error reference.

4.6.7 UBI_GetPhyPageStatus

This function returns physical page status (only data area).

UBI_Initialize (4.6.13) must be successfully called prior to calling this function.

4.6.7.1 Syntax

```
ubi_error_t UBI_GetPhyPageStatus(
  const ubi_t *ubi,
  uint32_t block,
  uint32_t page,
  ubi_content_state_t *state
);
```

4.6.7.2 Parameters

ubi	UBI handle
	Pointer to the structure holding UBI state.
block	Logical block index
	Specifies zero-based logical block index where physical page is located.
page	Physical page index
	Specifies zero-based physical page index within logical **block**.
state	State of physical page
	Pointer to ubi_content_state_t enumeration.

4.6.7.3 Return value

If the function succeeds the return value is **UBI_ERR_SUCCESS**.

If the function fails, it returns an error code. Refer to (4.7) for error reference.

Page status (data area) is returned through parameter **state:**

- UBI_CONTENT_STATE_CLEAN – data is not programmed (contains only FFs)
- UBI_CONTENT_STATE_DAMAGED – data area is damaged
- UBI_CONTENT_STATE_PROGRAMMED – data area is correctly programmed

4.6.8 UBI_GetPhyPagesPerBlock

This function returns the number of physical pages per logical block.

UBI_Initialize (4.6.13) must be successfully called prior to calling this function.

4.6.8.1 Syntax

```
ubi_error_t UBI_GetPhyPagesPerBlock(
  ubi_t       *ubi,
  uint32_t    *phy_pages_per_block
);
```

4.6.8.2 Parameters

ubi	UBI handle
	Pointer to the structure holding UBI state.
phy_pages_per_block	Physical pages per block
	Returned value is a number of physical pages per logical block.

4.6.8.3 Return value

If the function succeeds the return value is **UBI_ERR_SUCCESS**.

If the function fails, it returns an error code. Refer to (4.7) for error reference.

4.6.9 UBI_GetSpareSize

This function returns the size of spare area available to the user.

UBI_Initialize (4.6.13) must be successfully called prior to calling this function.

4.6.9.1 Syntax

```
ubi_error_t UBI_GetSpareSize(
  ubi_t       *ubi,
  uint32_t    *spare_size
);
```

4.6.9.2 Parameters

ubi	UBI handle
	Pointer to the structure holding UBI state.
spare_size	Spare size
	Returned value is a spare area size available to the user.

4.6.9.3 Return value

If the function succeeds the return value is **UBI_ERR_SUCCESS**.

If the function fails, it returns an error code. Refer to (4.7) for error reference.

4.6.10 UBI_GetVirtPageSize

This function returns the size of a virtual page.

UBI_Initialize (4.6.13) must be successfully called prior to calling this function.

4.6.10.1 Syntax

```
uint32_t UBI_GetVirtPageSize(
  ubi_t      *ubi,
  uint32_t   *virt_page_size
);
```

4.6.10.2 Parameters

ubi	UBI handle
	Pointer to the structure holding UBI state.
virt_page_size	Virtual page size
	Returned value is a virtual page size in bytes.

4.6.10.3 Return value

If the function succeeds the return value is **UBI_ERR_SUCCESS**.

If the function fails, it returns an error code. Refer to (4.7) for error reference.

4.6.11 UBI_GetVirtPageStatus

This function returns virtual page status.

UBI_Initialize (4.6.13) must be successfully called prior to calling this function.

4.6.11.1 Syntax

```
ubi_error_t UBI_GetVirtPageStatus(
  const ubi_t *ubi,
  uint32_t block,
  uint32_t page,
  ubi_content_state_t *state
);
```

4.6.11.2 Parameters

ubi	UBI handle
	Pointer to the structure holding UBI state.
block	Logical block index
	Specifies zero-based logical block index where physical page is located.
page	Virtual page index
	Specifies zero-based virtual page index within logical block.
state	State of virtual page
	Pointer to ubi_content_state_t enumeration.

4.6.11.3 Return value

If the function succeeds the return value is **UBI_ERR_SUCCESS**.

If the function fails, it returns an error code. Refer to (4.7) for error reference.

Page status is returned through parameter **state:**

- UBI_CONTENT_STATE_CLEAN – data is not programmed (contains only FFs)
- UBI_CONTENT_STATE_DAMAGED – data area is damaged
- UBI_CONTENT_STATE_PROGRAMMED – data area is correctly programmed

4.6.12 UBI_Health

This function is used to retrieve current flash health state. Refer to (4.3.10) for informations about interpreting flash health.

UBI_Initialize (4.6.13) must be successfully called prior to calling this function.

4.6.12.1 Syntax

```
ubi_error_t UBI_Health(
  ubi_t         *ubi,
  ubi_health_t  *health
);
```

4.6.12.2 Parameters

ubi	UBI handle
	Pointer to the structure holding UBI state.
health	Pointer to the **ubi_health_t** structure
	UBI_Health stores health informations to the structure pointed by this parameter.

4.6.12.3 Return value

If the function succeeds the return value is **UBI_ERR_SUCCESS**.

If the function fails, it returns an error code. Refer to (4.7) for error reference.

4.6.13 UBI_Initialize

This function is used to initialize the UBI. After opening a NAND flash device, the function reads and checks UBI structures in the flash and allocates all the necessary buffers.

4.6.13.1 Syntax

If UBI is compiled in read-only mode (4.5.5.5):

```
ubi_error_t UBI_Initialize(
  ubi_t       *ubi,
  const char  *device_name
);
```

If UBI is compiled in non read-only mode (4.5.5.5):

```
ubi_error_t UBI_Initialize(
  ubi_t       *ubi,
  const char  *device_name,
  int         readonly
);
```

4.6.13.2 Parameters

ubi	UBI handle
	Pointer to the structure holding UBI state.
device_name	NAND flash device name
	Specifies NAND flash device name as it is registered to Device Manager.
readonly	Read-only mode
	If != 0, UBI is initialized in read-only mode.

4.6.13.3 Return value

If the function succeeds the return value is **UBI_ERR_SUCCESS**.

If the function fails, it returns an error code. Refer to (4.7) for error reference.

4.6.14 UBI_ReadPhyPage

This function is used to read data and/or spare area of a physical page.

UBI_Initialize (4.6.13) must be successfully called prior to calling this function.

4.6.14.1 Syntax

```
ubi_error_t UBI_ReadPhyPage(
  ubi_t     *ubi,
  uint32_t  block,
  uint32_t  page,
  uint8_t   *data,
  uint8_t   *spare,
  uint32_t  spare_size,
  int       read_twice
);
```

4.6.14.2 Parameters

ubi	UBI handle
	Pointer to the structure holding UBI state.
block	Logical block index
	Specifies zero-based logical block index to read data from.
page	Physical page index
	Specifies zero-based physical page index within logical **block** to read data from.
data	Data buffer
	Specifies buffer to store read data to. The function assumes that the buffer is large enough to fit one physical page. Use function **UBI_GetPhyPageSize** (4.6.6) to get size of physical page. If **data** = NULL, the data area will not be read.
spare	Spare buffer
	Specifies buffer to store read spare area to. If **spare** = NULL, the spare area will not be read.
spare_size	Spare area size to read
	Specifies spare area size to read. The **spare** buffer must be large enough to fit **spare_size** bytes.
read_twice	Read data/spare twice
	If non-zero, the function will read data and/or spare location twice to make sure read bits are stable. This is particularly important for areas of flash which were being written when power loss occurred. If two reads give the same result (either success or error), function returns with this result. If results of two subsequent reads are different, function will read the same location third time and result of this third read will be returned.

4.6.14.3 Return value

If the function succeeds the return value is **UBI_ERR_SUCCESS**.

If the function fails, it returns an error code. Refer to (4.7) for error reference.

4.6.15 UBI_ReadVirtPage

This function is used to read virtual page data.

UBI_Initialize (4.6.13) must be successfully called prior to calling this function.

4.6.15.1 Syntax

```
ubi_error_t UBI_ReadVirtPage(
  ubi_t     *ubi,
  uint32_t  block,
  uint32_t  page,
  uint8_t   *data,
  int       read_twice
);
```

4.6.15.2 Parameters

ubi	UBI handle
	Pointer to the structure holding UBI state.
block	Logical block index
	Specifies zero-based logical block index to read data from.
page	Virtual page index
	Specifies zero-based virtual page index within logical **block** to read data from.
data	Data buffer
	Specifies buffer to store read data to. The function assumes that the buffer is large enough to fit one virtual page. Use function **UBI_GetVirtPageSize** (4.6.10) to get size of virtual page.
read_twice	Read data twice
	If non-zero, the function will read data location twice to make sure read bits are stable. This is particularly important for areas of flash which were being written when power loss occurred. If two reads give the same result (either success or error), function returns with this result. If results of two subsequent reads are different, function will read the same location third time and result of this third read will be returned.

4.6.15.3 Return value

If the function succeeds the return value is **UBI_ERR_SUCCESS**.

If the function fails, it returns an error code. Refer to (4.7) for error reference.

4.6.16 UBI_VerifyPhyPage

This function verifies if content of physical page matches supplied data and/or spare buffer content.

UBI_Initialize (4.6.13) must be successfully called prior to calling this function.

4.6.16.1 Syntax

```
ubi_error_t UBI_VerifyPhyPage(
  const ubi_t *ubi,
  uint32_t block,
  uint32_t page,
  const void *data,
  const void *spare,
  uint32_t spare_size
);
```

4.6.16.2 Parameters

ubi	UBI handle
	Pointer to the structure holding UBI state.
block	Logical block index
	Specifies zero-based logical block index where physical page is located.
page	Physical page index
	Specifies zero-based physical page index within logical **block**.
data	Data buffer
	Specifies buffer with data to be used for verification. The function assumes that the buffer is of size equal to one physical page. Use function **UBI_GetPhyPageSize** (4.6.6) to get size of physical page.
spare	Spare buffer
	Specifies buffer with spare area content to verify against.
spare_size	Spare area size to verify
	Specifies number of bytes of spare area to be verified.

4.6.16.3 Return value

If the function succeeds the return value is **UBI_ERR_SUCCESS**.

If the function fails, it returns an error code. Refer to (4.7) for error reference.

4.6.17 UBI_VerifyWritesEnable

This function enables or disables flash writes verification. By default the verification is on. If verification is on, after every flash write, the UBI reads back the written data and verifies if data matches.

UBI_Initialize (4.6.13) must be successfully called prior to calling this function.

4.6.17.1 Syntax

```
ubi_error_t UBI_VerifyWritesEnable(
  ubi_t  *ubi,
  int    enable
);
```

4.6.17.2 Parameters

ubi	UBI handle
	Pointer to the structure holding UBI state.
enable	Enable writes verification
	If non-zero, writes verification is enabled.

4.6.17.3 Return value

If the function succeeds the return value is **UBI_ERR_SUCCESS**.

If the function fails, it returns an error code. Refer to (4.7) for error reference.

4.6.18 UBI_WritePhyPage

This function is used to write data and/or spare area of a physical page.

UBI_Initialize (4.6.13) must be successfully called prior to calling this function.

4.6.18.1 Syntax

```
ubi_error_t UBI_WritePhyPage(
  ubi_t          *ubi,
  uint32_t       block,
  uint32_t       page,
  int            write_spare_first,
  int            ro_on_error,
  const uint8_t  *data,
  const uint8_t  *spare,
  uint32_t       spare_size
);
```

4.6.18.2 Parameters

ubi	UBI handle
	Pointer to the structure holding UBI state.
block	Logical block index
	Specifies zero-based logical block index to write data to.
page	Physical page index
	Specifies zero-based physical page index within logical **block** to write data to.
write_spare_first	Write spare data first
	If non-zero, the spare data will be written to spare area before writing normal data.
ro_on_error	Read-only on error
	If error occurs when writing physical page, switch UBI to read-only mode.
data	Data buffer
	Specifies buffer with data to be written. The function assumes that the buffer is of size equal to at least one physical page. Use function **UBI_GetPhyPageSize** (4.6.6) to get size of physical page.
spare	Spare buffer
	Specifies buffer with data to write to spare area.
spare_size	Spare area size to write
	Specifies number of bytes to write to spare area.

4.6.18.3 Return value

If the function succeeds the return value is **UBI_ERR_SUCCESS**.

If the function fails, it returns an error code. Refer to (4.7) for error reference.

4.6.19 UBI_WriteVirtPage

This function is used to write virtual page data.

UBI_Initialize (4.6.13) must be successfully called prior to calling this function.

4.6.19.1 Syntax

```
ubi_error_t UBI_WriteVirtPage(
  ubi_t          *ubi,
  uint32_t       block,
  uint32_t       page,
  const uint8_t  *data
);
```

4.6.19.2 Parameters

ubi	UBI handle
	Pointer to the structure holding UBI state.
block	Logical block index
	Specifies zero-based logical block index to write data to.
page	Virtual page index
	Specifies zero-based virtual page index within logical **block** to write data to.
data	Data buffer
	Specifies buffer with data to be written. The function assumes that the buffer is of size equal to at least one virtual page. Use function **UBI_GetVirtPageSize** (4.6.10) to get size of virtual page.

4.6.19.3 Return value

If the function succeeds the return value is **UBI_ERR_SUCCESS**.

If the function fails, it returns an error code. Refer to (4.7) for error reference.

4.7 Errors reference

For list of possible errors refer to header file:

<installation_folder>\sciopta\<version>\include\ftl\ubi.h

5 Flash Translation Layer (FTL)

5.1 Introduction

The FTL (Flash Translation Layer) is a layer which provides an abstraction over NAND flash memory pages. The FTL layer is located between user application (usually a filesystem) and UBI layer (4.1).

A flash page cannot be reprogrammed without first erasing entire block. The FTL takes care about erasing the blocks and keeps the translation tables, which map page numbers used by user application to page numbers in flash memory. The process of erasing blocks and reprogramming pages is transparent to the user application.

The UBI gives access to N physical pages (number of blocks times number of physical pages per block), and FTL gives access to M logical pages, where (M < N). Logical page may be divided into virtual pages (if memory device supports them).

5.2 Using FTL

5.2.1 Formatting flash

NAND flash memory can be used by FTL after being formatted. To format flash memory use function **FTL_FormatOnly** (5.5.3).

5.2.2 Initializing FTL

To initialize FTL with specific flash device, use function **FTL_Initialize** (5.5.5). This function checks if data structures are correct and from this point on the flash memory can be used by the upper layer.

5.2.3 Running FTL background task

The only FTL background task's role is to call **UBI_Background** function (4.6.1). Refer to (4.3.4) for informations about how **UBI_Background** function should be used, and apply the same rules to calling **FTL_Background** function (5.5.1).

5.2.4 Writing flash pages

To write flash logical page, use function **FTL_WriteLog** (5.5.10).

To write flash virtual page, use function **FTL_WriteVirt** (5.5.11).

5.2.5 Reading flash pages

To read flash logical page, use function **FTL_ReadLog** (5.5.6).

To read flash virtual page, use function **FTL_ReadVirt** (5.5.7).

5.2.6 Trimming flash pages

In order to keep the highest write performance possible, it is required to mark pages as no longer needed. If page is trimmed, its content will not be copied by internal FTL engine when performing certain operations, such as reclaiming new data blocks for data writes.

To trim a range of logical pages, use function **FTL_Trim** (5.5.8).

5.2.7 Enabling writes verification

To enable or disable flash writes verification, use function **FTL_VerifyWritesEnable** (5.5.9).

5.2.8 Getting and interpreting flash health

To get flash memory health status, use function **FTL_Health** (5.5.4).

```
typedef struct ftl_health_s {
  ubi_health_t ubi;
  uint32_t totalPages;
  uint32_t usedPages;
} ftl_health_t;
```

ubi	UBI layer health status
	Contains UBI layer health status. Refer to (4.3.10) for details about interpreting

	UBI layer flash health.
totalPages	Number of logical pages
	Indicates a number of logical pages available to the upper layer.
usedPages	Number of pages being used
	Indicates a number of pages being in use. Pages which are not in use are read if they contained zeros.

5.3 Power Loss Recovery

FTL deals with power fails by attempting to write the same data structures as before the power fail. This is done to make the FTL structures integral. The underlying flash memory should allow writing the same data, and FTL verifies written data. However unlikely it is, if FTL fails to write data in Power Loss Recovery procedure, it will switch to read-only mode, allowing upper layers to access the data stored in flash.

Power Loss Recovery assures that FTL_WriteLog (5.5.10) and FTL_WriteVirt (5.5.11) functions are atomic. This means that either an old or new data is available in logical/virtual page after power loss.

Power Loss Recovery assures that FTL_Trim functions will not damage FTL structures if power is lost, however, it is not guaranteed that all pages will be trimmed after power is back.

5.4 Configuration

5.4.1 Introduction

The FTL uses header “ftl_cfg.h” to read the user configuration. A template configuration header can be found in following location:

<installation_folder>\sciopta\<version>\include\ftl\ftl_cfg_template.h

5.4.2 Configuring FTL for flash device

FTL layer can be configured in many different ways, depending on the requirements. In general the configuration target can be chosen between performance, RAM usage and amount of flash memory available to the upper layer. Following factors influence the FTL performance:

- writing and reading pages sequentially or randomly (sequential is faster)
- writing logical pages or virtual pages (writing logical pages is faster, unless virtual page was not written before)
- using FTL_Trim (5.5.8) (trimmed logical pages are written faster, trimming unused pages improves overall FTL performance)

Following options influence FTL performance:

- FTL_ADDITIONAL_FREE_BLOCKS_PERCENTAGE (5.4.5.2) - bigger percentage means faster writes
- FTL_USE_VPMAP_CACHE (5.4.5.3) - bigger value improves write and read performance
- FTL_USE_DIRTY_CACHE (5.4.5.3) - bigger value improves only write performance

5.4.3 Configuring for minimal RAM memory usage

To configure UBI for minimal memory usage, set following setting:

FTL_USE_VPMAP_CACHE = 0 (refer to 5.4.5.3)

FTL_USE_DIRTY_CACHE = 0 (refer to 5.4.5.3)

5.4.4 Configuring for maximum performance

To configure UBI for maximum performance, set following setting:

FTL_USE_VPMAP_CACHE = 0xFFFFFFFF (refer to 5.4.5.3)

FTL_USE_DIRTY_CACHE = 0xFFFFFFFF (refer to 5.4.5.3)

5.4.5 Options reference

5.4.5.1 Flash device identification

Option name:	FTL_MEMORY_DEVICE_NAME
Valid values:	Flash device driver name as registered into Device Manager
Description:	The FTL uses this name to get flash device driver object from Device Manager.

5.4.5.2 Flash memory layout

Option name:	FTL_ADDITIONAL_FREE_BLOCKS_PERCENTAGE
Valid values:	0, 1, 2, …, 99
Description:	The more blocks are assigned to be free blocks, the better the write performance (but less the available flash memory size). **Changing this option requires reformatting the flash memory to create new FTL structures.**

5.4.5.3 Caching

Option name:	FTL_USE_VPMAP_CACHE
Valid values:	0, 1, 2, 3, …, 0xFFFFFFFF
Description:	This option selects the size of cache for FTL VPMAP structures (translations between logical pages and physical pages in the flash memory). The cache size is in units of physical/logical page size. The cache actually allocated is never bigger than required. Thus setting this option to 0xFFFFFFFF assures, that FTL works with maximum performance, because all VPMAP structures are cached.

Option name:	FTL_USE_DIRTY_CACHE
Valid values:	0, 1, 2, 3, …, 0xFFFFFFFF
Description:	This option selects the size of cache for FTL blocks dirtiness tracking structures (dirtiness of a block increases when programmed page is no longer valid). The cache size is in units of physical/logical page size. The cache actually allocated is never bigger than required. Thus setting this option to 0xFFFFFFFF assures, that FTL works with maximum performance, because all DIRTY structures are cached.

5.4.5.4 Read-only

Option name:	FTL_READ_ONLY
Valid values:	0 – Read only disabled 1 – Read only enabled
Description:	If this option is enabled, the FTL is compiled in read-only mode (decreases code footprint).

5.4.5.5 Support for virtual pages

Option name:	FTL_SUPPORT_FLASH_VIRTUAL_PAGES
Valid values:	0 – Support for virtual pages disabled 1 – Support for virtual pages enabled
Description:	If this option is enabled, the FTL is compiled with support for flash device virtual pages. **Changing this option requires reformatting the flash memory to create new FTL structures.**

5.4.5.6 Trap interface

Option name:	FTL_USE_TRAP_INTERFACE
Valid values:	0 – Disabled 1 – Enabled
Description:	If this option is enabled, Sciopta trap interface will be used by FTL sources.

5.4.5.7 Support for more than 128 physical pages per block

Option name:	FTL_SUPPORT_MORE_THAN_128_PAGES_PER_BLOCK
Valid values:	0 – Support for more than 128 physical pages per block disabled 1 – Support for more than 128 physical pages per block enabled
Description:	If this option is enabled, the FTL is compiled with support for more than 128 physical pages per block. **Changing this option requires reformatting the flash memory to create new FTL structures.**

5.4.5.8 Debug log

Option name:	FTL_LOG_LEVEL
Valid values:	0 – log is disabled 1 – only error messages are printed 2 – diagnostic and error messages are printed
Description:	This option enables sending FTL log messages to Log Deamon.

5.5 Function interface reference

5.5.1 FTL_Background

This function runs FTL background tasks. Refer to (5.2.3) for detailed informations about calling this function.

FTL_Initialize (5.5.5) must be successfully called prior to calling this function.

5.5.1.1 Syntax

```
ftl_error_t FTL_Background(
  ftl_t  *ftl
);
```

5.5.1.2 Parameters

ftl	FTL handle
	Pointer to the structure holding FTL state.

5.5.1.3 Return value

If the function succeeds the return value is **FTL_ERR_SUCCESS**.

If the function fails, it returns an error code. Refer to (5.6) for error reference.

5.5.2 FTL_Deinitialize

This function deinitializes the FTL by closing NAND flash device and releasing all allocated memory.

FTL_Initialize (5.5.5) must be successfully called prior to calling this function.

5.5.2.1 Syntax

```
ftl_error_t FTL_Deinitialize(
  ftl_t  *ftl
);
```

5.5.2.2 Parameters

ftl	FTL handle
	Pointer to the structure holding FTL state.

5.5.2.3 Return value

If the function succeeds the return value is **FTL_ERR_SUCCESS**.

If the function fails, it returns an error code. Refer to (5.6) for error reference.

5.5.3 FTL_FormatOnly

This function formats flash device. First, the UBI_FormatOnly (4.6.4) function is called to create UBI structures. Then FTL creates its necessary structures in the flash.

5.5.3.1 Syntax

```
ftl_error_t FTL_FormatOnly(
  ftl_t  *ftl,
  const char *device_name
);
```

5.5.3.2 Parameters

ftl	FTL handle
	Pointer to the structure holding FTL state.
device_name	NAND flash device name
	Specifies NAND flash device name as it is registered to Device Manager.

5.5.3.3 Return value

If the function succeeds the return value is **FTL_ERR_SUCCESS**.

If the function fails, it returns an error code. Refer to (5.6) for error reference.

5.5.4 FTL_Health

This function is used to retrieve current flash health state. Refer to (5.2.8) for informations about interpreting flash health.

FTL_Initialize (5.5.5) must be successfully called prior to calling this function.

5.5.4.1 Syntax

```
ftl_error_t FTL_Health(
  ftl_t          *ftl,
  ftl_health_t   *health
);
```

5.5.4.2 Parameters

ftl	FTL handle
	Pointer to the structure holding FTL state.
health	Pointer to the ftl_health_t structure
	FTL_Health stores health informations to the structure pointed by this parameter.

5.5.4.3 Return value

If the function succeeds the return value is **FTL_ERR_SUCCESS**.

If the function fails, it returns an error code. Refer to (5.6) for error reference.

5.5.5 FTL_Initialize

This function is used to initialize the FTL. The function first initializes the UBI layer (4.6.13) and if it was successful, the function reads and checks FTL structures in the flash and allocates all the necessary buffers.

5.5.5.1 Syntax

The **readonly** parameter is present only if FTL is compiled as not-readonly (5.4.5.4)

If virtual pages are supported (5.4.5.5):

```
ftl_error_t FTL_Initialize(
  ftl_t       *ftl,
  const char  *device_name,
  uint32_t    *logical_pages,
  uint32_t    *virtual_pages,
  uint32_t    *logical_page_size,
  uint32_t    *virtual_page_size,
  int         readonly
);
```

If virtual pages are not supported (5.4.5.5):

```
ftl_error_t FTL_Initialize(
  ftl_t       *ftl,
  const char  *device_name,
  uint32_t    *logical_pages,
  uint32_t    *logical_page_size,
  int         readonly
);
```

5.5.5.2 Parameters

ftl	FTL handle
	Pointer to the structure holding FTL state.
device_name	NAND flash device name
	Specifies NAND flash device name as it is registered to Device Manager.
logical_pages	Number of logical pages
	The FTL returns number of logical pages available to the user if initialization was successful.
virtual_pages	Number of virtual pages
	The FTL returns number of virtual pages available to the user if initialization was successful.
logical_page_size	Logical page size
	The FTL returns logical page size if initialization was successful.
virtual_page_size	Virtual page size
	The FTL returns virtual page size if initialization was successful.

5.5.5.3 Return value

If the function succeeds the return value is **FTL_ERR_SUCCESS**.

If the function fails, it returns an error code. Refer to (5.6) for error reference.

5.5.6 FTL_ReadLog

This function is used to read logical page. The function expects the **data** buffer to be at least the size indicated by FTL_Initialize function (5.5.5).

FTL_Initialize (5.5.5) must be successfully called prior to calling this function.

5.5.6.1 Syntax

```
ftl_error_t FTL_ReadLog(
  ftl_t      *ftl,
  uint32_t   logical_page_number,
  uint8_t    *data
);
```

5.5.6.2 Parameters

ftl	FTL handle
	Pointer to the structure holding FTL state.
logical_page_number	Logical page number
	Specifies logical page number to be read
data	Data buffer
	Specifies buffer to store read data to. The function assumes that the buffer is large enough to fit one logical page.

5.5.6.3 Return value

If the function succeeds the return value is **FTL_ERR_SUCCESS**.

If the function fails, it returns an error code. Refer to (5.6) for error reference.

5.5.7 FTL_ReadVirt

This function is used to read virtual page. The function expects the **data** buffer to be at least the size indicated by FTL_Initialize function (5.5.5).

FTL_Initialize (5.5.5) must be successfully called prior to calling this function.

5.5.7.1 Syntax

```
ftl_error_t FTL_ReadVirt(
  ftl_t     *ftl,
  uint32_t  virtual_page_number,
  uint8_t   *data
);
```

5.5.7.2 Parameters

ftl	FTL handle
	Pointer to the structure holding FTL state.
virtual_page_number	Virtual page number
	Specifies virtual page number to read.
data	Data buffer
	Specifies buffer to store read data to. The function assumes that the buffer is large enough to fit one virtual page.

5.5.7.3 Return value

If the function succeeds the return value is **FTL_ERR_SUCCESS**.

If the function fails, it returns an error code. Refer to (5.6) for error reference.

5.5.8 FTL_Trim

This function marks specified pages as no longer used by the user.

In order to keep the highest write performance possible, it is required to mark pages as no longer needed. If page is trimmed, its content will not be copied by internal FTL engine when performing certain operations, such as reclaiming new data blocks for data writes.

FTL_Initialize (5.5.5) must be successfully called prior to calling this function.

5.5.8.1 Syntax

```
ftl_error_t FTL_Trim(
  ftl_t      *ftl,
  uint32_t   logical_page_number_start,
  uint32_t   logical_page_number_stop
);
```

5.5.8.2 Parameters

ftl	FTL handle
	Pointer to the structure holding FTL state.
logical_page_number_start	First number of logical pages range
	Specifies the first page in a range of pages to be trimmed.
logical_page_number_stop	Last number of logical pages range
	Specifies the last page in a range of pages to be trimmed.

5.5.8.3 Return value

If the function succeeds the return value is **FTL_ERR_SUCCESS**.

If the function fails, it returns an error code. Refer to (5.6) for error reference.

5.5.9 FTL_VerifyWritesEnable

This function enables automatic verification for all writes to the flash device. By default the verification is on. If verification is on, after every flash write, the FTL reads back the written data and verifies if data matches. The UBI_VerifyWritesEnable (4.6.17) is also called with the same value of **enable** parameter.

FTL_Initialize (5.5.5) must be successfully called prior to calling this function.

5.5.9.1 Syntax

```
ftl_error_t FTL_VerifyWritesEnable(
  ftl_t  *ftl,
  int    enable
);
```

5.5.9.2 Parameters

ftl	FTL handle
	Pointer to the structure holding FTL state.
enable	Enable writes verification
	If non-zero, writes verification is enabled.

5.5.9.3 Return value

If the function succeeds the return value is **FTL_ERR_SUCCESS**.

If the function fails, it returns an error code. Refer to (5.6) for error reference.

5.5.10 FTL_WriteLog

This function is used to write logical page. The function expects the **data** buffer to be at least the size indicated by FTL_Initialize function (5.5.5).

FTL_Initialize (5.5.5) must be successfully called prior to calling this function.

5.5.10.1 Syntax

```
ftl_error_t FTL_WriteLog(
  ftl_t          *ftl,
  uint32_t       logical_page_number,
  const uint8_t  *data
);
```

5.5.10.2 Parameters

ftl	FTL handle
	Pointer to the structure holding FTL state.
logical_page_number	Logical page number
	Specifies logical page number to write.
data	Data buffer
	Specifies buffer with data to be written. The function assumes that the buffer is of size equal to at least one logical page.

5.5.10.3 Return value

If the function succeeds the return value is **FTL_ERR_SUCCESS**.

If the function fails, it returns an error code. Refer to (5.6) for error reference.

5.5.11 FTL_WriteVirt

This function is used to write virtual page. The function expects the **data** buffer to be at least the size indicated by FTL_Initialize function (5.5.5).

FTL_Initialize (5.5.5) must be successfully called prior to calling this function.

5.5.11.1 Syntax

```
ftl_error_t FTL_WriteVirt(
  ftl_t          *ftl,
  uint32_t       virtual_page_number,
  const uint8_t  *data
);
```

5.5.11.2 Parameters

ftl	FTL handle
	Pointer to the structure holding FTL state.
virtual_page_number	Virtual page number
	Specifies virtual page number to write.
data	Data buffer
	Specifies buffer with data to be written. The function assumes that the buffer is of size equal to at least one virtual page.

5.5.11.3 Return value

If the function succeeds the return value is **FTL_ERR_SUCCESS**.

If the function fails, it returns an error code. Refer to (5.6) for error reference.

5.6 Errors reference

For list of possible errors refer to header file:

<installation_folder>\sciopta\<version>\include\ftl\ftl.h

6 GDD-compatible FTL SCIOPTA driver

6.1 Introduction

FTL package provides a driver for the FTL, which is compatible with General Device Driver Model in SCIOPTA.

The driver allows to open, read and write FTL flash memory device. Additional I/O control commands allow to get flash layout informations, format a flash memory (refer to 5.2.1), change behaviour of FTL background task (refer to 5.2.3) and retrieve flash health status (refer to 5.2.8).

6.2 Adding driver to the Sciopta project

The driver is located in following file:

<installation_folder>\sciopta\<version>\sfs\ftl\sciopta_ftl.c

Driver process must be defined in SCONF XML configuration file using function name SCP_scioptaftl. Any process name can be used.

Driver registers itself as a device into Device Manager with name **scioptaftl**. Device Manager process path can be configured using UBI options (Fehler: Referenz nicht gefunden).

The priority of FTL driver process must not be higher than the priority of memory device driver defined by FTL option (refer to 5.4.5.1).

6.3 Message interface reference

SCIOPTA is a message based real-time operating system. Interprocess communication and coordination is done by messages. Message passing is a very fast, secure, easy to use and a good to debug method.

Messages are the preferred tool for interprocess communication in SCIOPTA. SCIOPTA is specifically designed to have a very high message passing performance. Messages can also be used for interprocess coordination or synchronization duties to initiate different actions in processes. For this purposes messages can but do not need to carry data.

A message buffer (the data area of a message) can only be accessed by one process at a time which is the owner of the message. A process becomes an owner of a message when it allocates the message by the **sc_msgAlloc** system call or when it receives the message by the **sc_msgRx** system call.

Message passing is also possible between processes on different CPUs. In this case specific communication process types on each side will be needed called **SCIOPTA CONNECTOR Processes**.

In this chapter all messages supported by SCIOPTA FTL driver are described.

The error code is included in the error member of the **sdd_baseMessage_t** structure and is used in the reply message. In the request message error must be set to zero.

The messages are defined in the following header file:

<installation_folder>\sciopta\<version>\include\sdd\sdd.msg

6.3.1 SDD_DEV_OPEN / SDD_DEV_OPEN_REPLY

6.3.1.1 Description

This message is used to open FTL memory object for read or read/write.

The user process sends an **SDD_DEV_OPEN** request message to the controller process of FTL driver object. The controller process replies with an **SDD_DEV_OPEN_REPLY** reply message.

6.3.1.2 Message IDs

Request message	**SDD_DEV_OPEN**
Reply message	**SDD_DEV_OPEN_REPLY**

6.3.1.3 sdd_devOpen_t Structure

```
typedef struct sdd_devOpen_s {
  sdd_baseMessage_t base;
  flags_t flags;
} sdd_devOpen_t;
```

6.3.1.4 Structure Members

base	Driver object descriptor.
	Specifies an FTL driver object to be opened.
flags	FTL open flags.
	Used by the **request message** and contains BSD conform flags.
O_RDONLY	Opens the FTL memory for read only. Cannot be ored with O_WRONLY.
O_WRONLY	Opens the FTL memory for write only.
O_RDWR	Opens the FTL memory for read and write. Cannot be ored with O_RDONLY or O_WRONLY.

6.3.1.5 Errors

base.error	Error code.
EBUSY	FTL driver is already opened.
EROFS	FTL is configured as read only.
ENOTSUP	Specified flags combination is not supported.
EIO	Failed initializing FTL memory.

6.3.2 SDD_DEV_CLOSE / SDD_DEV_CLOSE_REPLY

6.3.2.1 Description

This message is used to close FTL memory object.

The user process sends an **SDD_DEV_CLOSE** request message to the controller process of FTL driver object. The controller process replies with an **SDD_DEV_CLOSE_REPLY** reply message.

6.3.2.2 Message IDs

Request message	**SDD_DEV_CLOSE**
Reply message	**SDD_DEV_CLOSE_REPLY**

6.3.2.3 sdd_devClose_t Structure

```
typedef struct sdd_devClose_s {
  sdd_baseMessage_t base;
} sdd_devClose_t;
```

6.3.2.4 Structure Members

base	Driver object descriptor.
	Specifies an FTL driver object to be closed.

6.3.2.5 Errors

base.error	Error code.
EBADF	FTL driver is not opened.
EACCES	Only process which opened the driver can close it.

6.3.3 SDD_DEV_READ / SDD_DEV_READ_REPLY

6.3.3.1 Description

This message is used to read data from FTL memory.

The user process sends an **SDD_DEV_READ** request message to the reader process of FTL driver object. The reader process replies with an **SDD_DEV_READ_REPLY** reply message. The reply contains the read data.

6.3.3.2 Message IDs

Request message	**SDD_DEV_READ**
Reply message	**SDD_DEV_READ_REPLY**

6.3.3.3 sdd_devRead_t Structure

```
typedef struct sdd_devRead_s {
  sdd_baseMessage_t base;
  ssize_t size;
  ssize_t curpos;
  uint8_t *outlineBuf;
  uint8_t inlineBuf[1];
} sdd_devRead_t;
```

6.3.3.4 Structure Members

base	Driver object descriptor.
	Specifies an FTL driver object to read data from.
size	Number of bytes to read.
	In the **request message** contains a number of bytes to read from the FTL memory. In the **reply message** contains a number of bytes actually read. Size must be a multiple of virtual page size, or if virtual pages are not supported, a multiple of logical page size.
curpos	Not used.
	This parameter is not used by FTL driver.
outlineBuf	Pointer to a referenced buffer to store data to.
<readptr>	Used by the **reply message** and can contain a pointer to the buffer to put the data to. Not recommended as pointers should not be used in messages. Rather use **inlineBuf**.
0	The member **inlineBuf** is used.
inlineBuf	In-message buffer to store data to.
	Buffer used by the **reply message** if **outlineBuf** is not used. The size is variable and all data will be put into this buffer. The size of allocated SDD_DEV_READ message must be big enough to fit the requested data size.

6.3.3.5 Errors

base.error	Error code.
EBADF	FTL driver is not opened.
EACCES	Only process which opened the driver can read from it.
EINVAL	Read size must be a multiple of virtual page size, or message uses inline buffer and message size is not enough to fit all the requested data.
EPERM	FTL driver is opened in write-only mode.
EIO	Error reading FTL memory.

6.3.4 SDD_DEV_WRITE / SDD_DEV_WRITE_REPLY

6.3.4.1 Description

This message is used to read data from FTL memory.

The user process sends an **SDD_DEV_WRITE** request message to the writer process of FTL driver object. The writer process replies with an **SDD_DEV_WRITE_REPLY** reply message. The reply contains the number of bytes written.

6.3.4.2 Message IDs

Request message	**SDD_DEV_WRITE**
Reply message	**SDD_DEV_WRITE_REPLY**

6.3.4.3 sdd_devWrite_t Structure

```
typedef struct sdd_devWrite_s {
  sdd_baseMessage_t base;
  ssize_t size;
  ssize_t curpos;
  const uint8_t *outlineBuf;
  uint8_t inlineBuf[1];
} sdd_devWrite_t;
```

6.3.4.4 Structure Members

base	Driver object descriptor.
	Specifies an FTL driver object to write data to.
size	Number of bytes to write.
	In the **request message** contains a number of bytes to write to the FTL memory. In the **reply message** contains a number of bytes actually written. Size must be a multiple of virtual page size, or if virtual pages are not supported, a multiple of logical page size.
curpos	Not used.
	This parameter is not used by FTL driver.
outlineBuf	Pointer to a referenced buffer to get the data from.
<readptr>	Used by the **request message** and can contain a pointer to the buffer to get the data for writing to the FTL memory. Not recommended as pointers should not be used in messages. Rather use **inlineBuf**. Not used by the **reply message** and can have any value.
0	The member **inlineBuf** is used.
inlineBuf	In-message buffer containing data to write.
	Buffer used by the **request message** if **outlineBuf** is not used. The size is variable and all data for writing will be taken from this buffer. The allocated SDD_DEV_WRITE message must be big enough to contain all the data that are requested to be written. Not used by the **reply message** and can have any value.

6.3.4.5 Errors

base.error	Error code.
EROFS	FTL is compiled as read-only.
EBADF	FTL driver is not opened.
EACCES	Only process which opened the driver can write to it.
EINVAL	Write size must be a multiple of virtual page size, or if virtual pages are not supported, a multiple of logical page size, or message uses inline buffer and message size is not enough to contain all the data to be written.
EPERM	FTL driver is opened in read-only mode.
EIO	Error writing FTL memory.

6.3.5 SDD_FILE_SEEK / SDD_FILE_SEEK_REPLY

6.3.5.1 Description

This message is used to change current read/write offset of an opened FTL driver.

The user process sends an **SDD_FILE_SEEK** request message to the controller process of a FTL driver object. The controller process replies with an **SDD_FILE_SEEK_REPLY** reply message.

6.3.5.2 Message IDs

Request message	**SDD_FILE_SEEK**
Reply message	**SDD_FILE_SEEK_REPLY**

6.3.5.3 sdd_fileSeek_t Structure

```
typedef struct sdd_fileSeek_s {
  sdd_baseMessage_t base;
  off_t offset;
  int whence;
} sdd_fileSeek_t;
```

6.3.5.4 Structure Members

base	FTL driver object descriptor.
	Specifies an FTL driver object to change current position of.
offset	New offset.
	In the **request message** contains a new FTL memory read/write offset.
	In the **reply message** contains an actual offset from the beginning of an FTL memory after changing current FTL memory position.
whence	Offset origin.
SEEK_CUR	**offset** is relative to the current position.
SEEK_END	**offset** is relative to the end of the FTL memory.
SEEK_SET	**offset** is relative to the beginning of the FTL memory.

6.3.5.5 Errors

base.error	Error code.
EBADF	FTL driver is not opened.
EACCES	Only process which opened the driver can seek the current position pointer.
EINVAL	Offset origin is invalid, or offset parameter is not a multiple of virtual page size or if virtual pages are not supported, a multiple of logical page size.
ERANGE	Calculated offset is negative, or goes beyond FTL memory size, or new offset is too large (>2^31-1) to be returned in a reply message.

6.3.6 SDD_FILE_SEEK64 / SDD_FILE_SEEK64_REPLY

6.3.6.1 Description

This message is used to change current read/write offset of an opened FTL driver.

The user process sends an **SDD_FILE_SEEK64** request message to the controller process of a FTL driver object. The controller process replies with an **SDD_FILE_SEEK64_REPLY** reply message.

6.3.6.2 Message IDs

Request message	**SDD_FILE_SEEK64**
Reply message	**SDD_FILE_SEEK64_REPLY**

6.3.6.3 sdd_fileSeek64_t Structure

```
typedef struct sdd_fileSeek64_s {
  sdd_baseMessage_t base;
  int64_t offset;
  int whence;
} sdd_fileSeek64_t;
```

6.3.6.4 Structure Members

base	FTL driver object descriptor.
	Specifies an FTL driver object to change current position of.
offset	New offset.
	In the **request message** contains a new FTL memory read/write offset.
	In the **reply message** contains an actual offset from the beginning of an FTL memory after changing current FTL memory position.
whence	Offset origin.
SEEK_CUR	**offset** is relative to the current position.
SEEK_END	**offset** is relative to the end of the FTL memory.
SEEK_SET	**offset** is relative to the beginning of the FTL memory.

6.3.6.5 Errors

base.error	Error code.
EBADF	FTL driver is not opened.
EACCES	Only process which opened the driver can seek the current position pointer.
EINVAL	Offset origin is invalid, or offset parameter is not a multiple of virtual page size or if virtual pages are not supported, a multiple of logical page size.
ERANGE	Calculated offset is negative, or goes beyond FTL memory size.

6.3.7 SDD_DEV_IOCTL / SDD_DEV_IOCTL_REPLY

6.3.7.1 Description

This message is used to get or set certain properties of an FTL driver.

The user process sends an **SDD_DEV_IOCTL** request message to the controller process of an FTL driver object. The controller process replies with an **SDD_DEV_IOCTL_REPLY** reply message.

6.3.7.2 Message IDs

Request message	**SDD_DEV_IOCTL**
Reply message	**SDD_DEV_IOCTL_REPLY**

6.3.7.3 sdd_devIoctl_t Structure

```
typedef struct sdd_devIoctl_s {
  sdd_baseMessage_t base;
  unsigned int cmd;
  int ret;
  unsigned long outlineArg;
  unsigned char inlineArg[1];
} sdd_devIoctl_t;
```

6.3.7.4 Structure Members

base	FTL driver object descriptor.
	Specifies an FTL driver object.
cmd	Object specific command.
	Specifies an ioctl command to be executed on the object. Refer to 6.3.7.5 for list of supported commands.
ret	Return value.
	Always 0.
outlineArg	Command specific argument.
	If equals NULL, **inlineArg** is used.
inlineArg	Command specific included argument.
	Argument if **outlineArg** is not used. The size is variable and the whole argument is included.

6.3.7.5 Commands

6.3.7.5.1 Enable background task

This command is used to enable FTL background task execution. For informations about running FTL background task, refer to 5.2.3.

Command:	`FTL_IOCTL_BG_TASK_ENABLE`
Data type of argument:	Unsigned 32-bit integer. If non-zero, task is enabled. Otherwise task is disabled.

6.3.7.5.2 Change background task interval

This command is used to change the interval the FTL background task is running with. For informations about running FTL background task, refer to 5.2.3.

Command: `FTL_IOCTL_BG_TASK_INTERVAL`

Data type of argument: Unsigned 32-bit integer. Time in milliseconds between background task runs.

6.3.7.5.3 Format flash

This commands is used to format FTL memory. Device driver cannot be opened with SDD_DEV_OPEN.

Command: `FTL_IOCTL_FORMAT_FLASH`

No argument is required for this command.

6.3.7.5.4 Trim pages

This command is used to trim logical pages in FTL memory (refer to 5.2.6 for more informations).

Command: `BLKDEVTRIM`

Data type of argument: **blkdev_trim_t** structure.

If virtual pages are supported:

blkdev_trim_t.sector_start – first virtual page to be trimmed

blkdev_trim_t.sector_stop – last virtual page to be trimmed

sector_start and **sector_stop+1** must be aligned to number of virtual pages per logical page.

If virtual pages are not supported:

blkdev_trim_t.sector_start – first logical page to be trimmed

blkdev_trim_t.sector_stop – last logical page to be trimmed

6.3.7.5.5 Get flash health status

This commands is used to get flash health status (refer to 5.2.8 for more informations).

Command: `FTL_IOCTL_FLASH_HEALTH`

Data type of argument: **ftl_health_t** structure.

6.3.7.5.6 Get memory layout

This command is used to get an FTL memory size, encoded in **blkdev_geometry_t** or in **blkdev_geometry64_t** structure.

Small FTL memories (< 2GB):

cmd: `BLKDEVGETPRM`

Data type of **arg**: Pointer to **blkdev_geometry_t** structure.

Large FTL memories (>= 2GB):

cmd: `BLKDEVGETPRM64`

Data type of **arg**: Pointer to **blkdev_geometry64_t** structure.

6.3.7.5.7 Get erase block size

This commands is used to get erase block size of FTL memory, which is a smallest erasable unit of FTL abstraction over flash memory device. In case of FTL this is a logical page size. This command may be used by filesystem to optimize writes.

Command: `BLKDEVGETERASEBLKSIZE`

Data type of returned value: 32-bit unsigned integer.

6.3.7.6 Errors

base.error	Error code.
EBADF	FTL driver is not opened.
EACCES	Only process which opened the driver can execute commands.
EINVAL	Background task interval must be non-zero, or sector range to trim is not aligned to logical page.
EBUSY	FTL memory cannot be formatted if driver is opened.
EIO	Error formatting FTL memory, or error trimming pages, or error getting flash health status.
ERANGE	FTL memory size is larger than 2^31-1 (use BLKDEVGETPRM64 command instead of BLKDEVGETPRM).

6.3.8 SDD_OBJ_RELEASE / SDD_OBJ_RELEASE_REPLY

6.3.8.1 Description

This message is used to release a temporary object representing an FTL driver.

The user process sends an **SDD_OBJ_RELEASE** request message to the controller process of a FTL driver object. The controller process replies with an **SDD_OBJ_RELEASE_REPLY** reply message.

6.3.8.2 Message IDs

Request message	**SDD_OBJ_RELEASE**
Reply message	**SDD_OBJ_RELEASE_REPLY**

6.3.8.3 sdd_objRelease_t Structure

```
typedef struct sdd_objRelease_s {
  sdd_baseMessage_t base;
} sdd_objRelease_t;
```

6.3.8.4 Structure Members

base	FTL driver object descriptor. In **request message** this field contains a descriptor of an FTL driver object to be released.

6.3.8.5 Errors

No errors returned.

7 Using NOR flash memories with FTL

FTL was designed to work with NAND flash. However it is possible to use NOR flash if a NAND emulation layer is created on top of a NOR flash driver.

NAND emulation layer is simply a SCIOPTA GDD compatible driver, which follows the same requirements as NAND driver must follow to able to be used with FTL. Please refer to 4.2 for details about these requirements.

NAND and NOR flashes are divided into erasable units called blocks.

Each block (i.e. 128KB) in NAND memory is divided into physical pages (i.e. 2KB), which can also be divided into smaller chunks called virtual page (i.e. 512B). Each physical page has got an additional area called “spare” area. This area is used by FTL to store some additional informations and also is used by the NAND device or memory controller to store a CRC data for physical/virtual page. The FTL expects the memory device driver to take care about calculating CRC and storing it into spare area. The FTL requires access to 12 bytes of spare area per each physical page (ioctl commands 4.2.3.12 and 4.2.3.13). The flash memory driver should also take care about calculating CRC for this data.

NOR flash normally does not have the spare area. This means, that each block of NOR flash must be divided into physical pages and spare areas per each physical page. It is up to the NAND emulation layer to decide how block is divided.

8 Manual versions

8.1 Manual version 1.2

- Chapter 4.6.7 - Added UBI_GetPhyPageStatus function
- Chapter 4.6.11 - Added UBI_GetVirtPageStatus function
- Chapter 4.6.14.2 - Added read_twice parameter
- Chapter 4.6.15.2 - Added read_twice parameter
- Chapter 4.6.16 - Added UBI_VerifyPhyPage function
- Chapter 4.6.18.2 - Added ro_on_error parameter

8.2 Manual version 1.1

- Chapter 4.2.3.2 - Chapter name changed, option name changed
- Chapter 4.2.3.3 - “Getting logical block size” chapter added
- Chapter 4.2.3.8 - “Erasing physical block only if dirty” chapter added
- Chapter 4.5.5 - UBI_DEVICE_MANAGER_PATH option removed
- Chapter 4.5.5.2 - “Block erasing policy” chapter added
- Chapter 4.6.17 - writes verification is on by default
- Chapter 5.5.9 - writes verification is on by default

8.3 Manual version 1.0

Initial documentation.

9 Index