User Guide

V5

Express Logic

858.613.6640
Toll Free 888.THREADX
FAX 858.521.4259

www.expresslogic.com

Part Number: 000-0021
Revision 5.6

Contents

Chapter 1

Overview

LevelX provides NAND and NOR flash wear leveling facilities to embedded applications. Since both NAND and NOR flash memory can only be erased a finite number of times, it's critical to distribute the flash memory use evenly. This is typically called "wear leveling" and is the purpose behind LevelX.

The algorithm that chooses which flash block to reuse is primarily based on the erase count, but not entirely. The block with the lowest erase count might not be chosen if there is another block that has an erase count within an acceptable delta from the minimal erase count and that has a greater number of obsolete mappings. In such cases, the block with the greatest number of obsolete mappings will be erased and reused, thus saving overhead in moving valid mapping entries.

LevelX supports multiple instances of NAND and/or NOR parts, i.e., the application can utilize separate instances of LevelX within the same application. Each instance requires its own control block provided by the application as well as its own flash driver.

LevelX presents to the user an array of logical sectors that are mapped to physical flash memory inside of LevelX. To enhance performance, LevelX also provides a cache of the most recent logical sector mappings. The size of this cache is user defined. Applications may use LevelX in conjunction with FileX or may read/write logical sectors directly. LevelX has no dependency on FileX and very little dependency on ThreadX (only primitive ThreadX data types are used).

LevelX is designed for fault tolerance. Flash updates are performed in a multiple-step process that can be interrupted in each step. LevelX automatically recovers to the optimal state during the next operation.

LevelX requires a flash driver for physical access to the underlying flash memory. Example NAND and NOR simulated drivers are provided and can be used as a good starting point for implementing actual LevelX drivers. In addition, driver requirements are detailed later in this documentation.

The following chapters describe the functional operation for the NAND and NOR LevelX support.

Chapter 2

Installation and Use

Installation and use of LevelX is straightforward and described in the following sections of this chapter.

Distribution

LevelX is distributed in ANSI C where each function is contained in its own separate C file. The files in the LevelX distribution are as follows:

```
lx_api.h
lx_nand_flash_256byte_ecc_check.c
lx_nand_flash_256byte_ecc_compute.c
lx_nand_flash_block_full_update.c
lx_nand_flash_block_reclaim.c
lx_nand_flash_close.c
lx_nand_flash_defragment.c
lx_nand_flash_extended_cache_enable.c
lx_nand_flash_initialize.c
lx_nand_flash_logical_sector_find.c
lx_nand_flash_next_block_to_erase_find.c
lx_nand_flash_open.c
lx_nand_flash_page_ecc_check.c
lx_nand_flash_page_ecc_compute.c
lx_nand_flash_partial_defragment.c
lx_nand_flash_physical_page_allocate.c
lx_nand_flash_sector_mapping_cache_invalidate.c
lx_nand_flash_sector_read.c
lx_nand_flash_sector_release.c
lx_nand_flash_sector_write.c
lx_nand_flash_system_error.c
lx_nor_flash_block_reclaim.c
lx_nor_flash_close.c
lx_nor_flash_defragment.c
lx_nor_flash_extended_cache_enable.c
lx_nor_flash_initialize.c
lx_nor_flash_logical_sector_find.c
lx_nor_flash_next_block_to_erase_find.c
lx_nor_flash_open.c
lx_nor_flash_partial_defragment.c
lx_nor_flash_physical_sector_allocate.c
lx_nor_flash_sector_mapping_cache_invalidate.c
lx_nor_flash_sector_read.c
lx_nor_flash_sector_release.c
lx_nor_flash_sector_write.c
lx_nor_flash_system_error.c
```

There are also simulator and FileX driver samples for both LevelX NAND and NOR instances, as follows:

```
demo_filex_nand_flash.c
fx_nand_flash_simulated_driver.c
lx_nand_flash_simulator.c
```

```
demo_filex_nor_flash.c
fx_nor_flash_simulated_driver.c
lx_nor_flash_simulator.c
```

Of course, if only NAND flash is required, only the LevelX NAND flash files (***lx_nand_*.c***) are needed. Similarly, if only NOR flash is required, only the NOR flash files (***lx_nor_*.c***) are needed.

Configuration Options

LevelX can be configured at compile time via the conditional defines described below. Simply add the desired define to the compilation of each LevelX source to use the option.

Define	Meaning
LX_DIRECT_READ	Defined, this option bypasses the NOR flash driver read routine in favor or reading the NOR memory directly, resulting in a significant performance increase.
LX_FREE_SECTOR_DATA_VERIFY	Defined, this causes the LevelX NOR instance open logic to verify free NOR sectors are all ones.
LX_NAND_SECTOR_MAPPING_CACHE_SIZE	By default this value is 16 and defines the logical sector mapping cache size. Large values improve performance, but cost memory. The minimum size is 8 and all values must be a power of 2.
LX_NAND_FLASH_DIRECT_MAPPING_CACHE	Defined, this creates a direct mapping cache, such that there are no cache misses. It also requires that LX_NAND_SECTOR_MAPPING_CACHE_SIZE represents the exact number of total pages in your flash device.
LX_NOR_DISABLE_EXTENDED_CACHE	Defined, this disabled the extended NOR cache.
LX_NOR_EXTENDED_CACHE_SIZE	By default this value is 8, which represents a maximum of 8 sectors that can be cached in a NOR instance.
LX_NOR_SECTOR_MAPPING_CACHE_SIZE	By default this value is 16 and

	defines the logical sector mapping cache size. Large values improve performance, but cost memory. The minimum size is 8 and all values must be a power of 2.
LX_THREAD_SAFE_ENABLE	Defined, this makes LevelX thread-safe by using a ThreadX mutex object throughout the API.

Using LevelX

Using LevelX by itself or with FileX is easy. Simply include ***lx_api.h*** in the code that references the LevelX API and ensure that the LevelX object code is available at link time. Please examine the ***demo_filex_nand_flash.c*** and ***demo_filex_nor_flash.c*** for examples of how to use LevelX.

Chapter 3

LevelX NAND Support

NAND flash memory is commonly utilized for large data storage, which is typical of file systems. NAND memory consists of ***blocks***. Within each NAND block is a series of ***pages***. NAND blocks are erasable, which means that all pages within the NAND block are erased (set to all ones). Each NAND block page has a set of ***spare bytes*** that are utilized by LevelX for bookkeeping, bad block management, and error detection. NAND block pages are available in a variety of sizes. The most common page sizes are:

Page Size	Spare Bytes
256	8
512	16
2048	64

NAND memory differs from NOR memory in that there is no direct access, i.e., NAND memory cannot be read directly from the processor like NOR memory. NAND memory can only be written to after an erase a limited number of times. Again, this differs from NOR memory that can be written an unlimited number of times providing the write request is clearing set bits. Finally, the spare bytes associated with each page are unique to NAND flash. Typical spare byte configurations are:

Number of Spare Bytes	Format	
8	Bytes 0-2:	ECC bytes
	Bytes 3,4,6,7:	LevelX Sector Mapping
	Byte 5:	Bad block flag
16	Bytes 0-3,6-7:	ECC bytes
	Bytes 8-11:	LevelX Sector Mapping
	Bytes 12-15:	Unused
	Byte 5:	Bad block flag
64	Byte 0:	Bad block flag
	Bytes 2-5:	LevelX Sector Mapping
	Bytes 6-39:	Unused
	Bytes 40-63:	ECC bytes

LevelX Utilizes 4 of the spare bytes of each NAND page for keeping track of the logical sector mapped to the physical NAND page. These 4 bytes are used to implement a 32-bit unsigned integer with a LevelX proprietary format. The upper bit of the 32-bit field (bit 31) is used to indicate the logical sector-to-page mapping is valid. If this bit is 0, the information in this page is no longer valid. The next bit—bit 30—is used to indicate this page is in the process of becoming obsolete and a new sector is being written. Bit 29 is used to indicate when the mapping entry write is complete. If bit 29 is 0, the mapping entry write is complete. If bit 29 is set, the mapping entry was in the process of being written. Bits 30 and 29 are used in recovering from a potential power loss while updating a new flash page. Finally, the lower 29-bits (28-0) contain the logical sector number for the page.

LevelX Mapping Entry

Bit(s)	Meaning
31	Valid flag. When set and logical sector is not all ones indicates mapping is valid
30	Obsolete flag. When clear, this mapping is either obsolete or is in the process of becoming obsolete.
29	Mapping entry write is complete when this bit is 0
0-28	Logical sector mapped to this physical page—when not all ones.

LevelX also utilizes the first page of each NAND block for the block erase count as well as the list of mapped pages when the block is full. The format of the first page of a NAND block in LevelX is shown below:

LevelX Block Page 0 Format

[Block Erase Count]
[Page 1 Sector Mapping]

…

[Page “n” Sector Mapping]
[0xF0F0F0F0]

Note: *The page mapping information is only written when the block is full, i.e., all the pages of the block have been written to. This enables faster search for free pages and logical sector mapping during run-time.*

NAND Bad Block Support

NAND memory is also more likely to have bad blocks than NOR memory. This is largely because NAND manufacturers can increase yield by allowing bad blocks and requiring software to work-around such bad blocks. LevelX handles NAND bad block management by simply mapping around bad blocks.

LevelX also provides APIs for 256-byte Hamming Error Correction Codes (ECC) for the underlying LevelX driver to utilize for calculating new ECC codes or to perform 1-bit error correction on page reading within each 256-byte section of the page.

NAND Driver Requirements

LevelX requires an underlying NAND flash driver that is specific to the underlying flash part and hardware implementation. The driver is specified to LevelX during initialization via the API ***lx_nand_flash_open***. The prototype of the LevelX driver is:

```
INT nand_driver_initialize(LX_NAND_FLASH *instance);
```

The "***instance***" parameter specifies the LevelX NAND control block. The driver initialization function is responsible for setting up all the other driver-level services for the associated LevelX instance. The services required for each LevelX NAND instance are:

Read Page
Write Page
Block Erase
Block Erased Verify
Page Erased Verify
Block Status Get
Block Status Set
Block Extra Bytes Get
Block Extra Bytes Set
System Error Handler

Driver Initialization

These services are setup via setting function pointers in the ***LX_NAND_FLASH*** instance within the driver's initialization function. The driver initialization function also specifies the total number of block, pages

per block, bytes per page, and a RAM area large enough to read one page into memory. The driver initialization function likely also performs additional device and/or implementation-specific initialization duties before returning ***LX_SUCCESS***.

Driver Read Page

The LevelX NAND driver “read page” service is responsible for reading a specific page in a specific block of the NAND flash. All error checking and correcting logic is the responsibility of the driver service. If successful, the LevelX NAND driver returns ***LX_SUCCESS***. If not successful, the LevelX NAND driver returns ***LX_ERROR***. The prototype of the LevelX NAND driver “read page” service is:

```
INT nand_driver_read_page(ULONG block,ULONG page,
                          ULONG *destination, ULONG words);
```

Where “***block***” and “***page***” identify which page to read and “***destination***” and “***words***” specify where to place the page contents and how many 32-bit words to read.

Driver Write Page

The LevelX NAND driver “write page” service is responsible for writing a specific page into the specified block of the NAND flash. All error checking and ECC computation is the responsibility of the driver service. If successful, the LevelX NAND driver returns ***LX_SUCCESS***. If not successful, the LevelX NAND driver returns ***LX_ERROR***. The prototype of the LevelX NAND driver “write page” service is:

```
INT nand_driver_write_page(ULONG block, ULONG page,
                           ULONG *source, ULONG words);
```

Where “***block***” and “***page***” identify which page to write and “***source***” and “***words***” specify the source of the write and how many 32-bit words to write.

Note: LevelX relies on the driver for low-level error detection when writing to the flash page, which typically involves reading back the page and comparing with the write buffer to ensure the write was successful.

Driver Block Erase

The LevelX NAND driver “block erase” service is responsible for erasing the specified block of the NAND flash. If successful, the LevelX NAND

driver returns ***LX_SUCCESS***. If not successful, the LevelX NAND driver returns ***LX_ERROR***. The prototype of the LevelX NAND driver “block erase” service is:

```
INT nand_driver_block_erase(ULONG block,
                            ULONG erase_count);
```

Where “***block***” identifies which block to erase. The parameter “***erase_count***” is provided for diagnostic purposes. For example, the driver may want to alert another portion of the application software when the erase count exceeds a specific threshold.

Note: LevelX relies on the driver for low-level error detection when the block is erased, which typically involves ensuring that all pages of the block are all ones.

Driver Block Erased Verify

The LevelX NAND driver “block erased verify” service is responsible for verifying that the specified block of the NAND flash is erased. If it is erased, the LevelX NAND driver returns ***LX_SUCCESS***. If the block is not erased, the LevelX NAND driver returns ***LX_ERROR***. The prototype of the LevelX NAND driver “block erased verify” service is:

```
INT nand_driver_block_erased_verify(ULONG block);
```

Where “***block***” specifies which block to verify that it is erased.

Note: LevelX relies on the driver to examine all pages and all bytes of each page – including spare and data bytes – to ensure they are erased (contain all ones).

Driver Page Erased Verify

The LevelX NAND driver “page erased verify” service is responsible for verifying that the specified page of the specified block of the NAND flash is erased. If it is erased, the LevelX NAND driver returns ***LX_SUCCESS***. If the page is not erased, the LevelX NAND driver returns ***LX_ERROR***. The prototype of the LevelX NAND driver “page erased verify” service is:

```
INT nand_driver_page_erased_verify(ULONG block,
                                   ULONG page);
```

Where “***block***” specifies which block and “***page***” specifies the page to verify that it is erased.
Note: LevelX relies on the driver to examine all bytes of the specified page – including spare and data bytes – to ensure they are erased (contain all ones).

Driver Block Status Get

The LevelX NAND driver “block status get” service is responsible for retrieving the bad block flag of the specified block of the NAND flash. If it is successful, the LevelX NAND driver returns ***LX_SUCCESS***. If it is not successful, the LevelX NAND driver returns ***LX_ERROR***. The prototype of the LevelX NAND driver “block status get” service is:

```
INT nand_driver_block_status_get(ULONG block,
                                 UCHAR *bad_block_byte);
```

Where “***block***” specifies which block and “***bad_block_byte***” specifies the destination for the bad block flag.

Driver Block Status Set

The LevelX NAND driver “block status set” service is responsible for setting the bad block flag of the specified block of the NAND flash. If it is successful, the LevelX NAND driver returns ***LX_SUCCESS***. If it is not successful, the LevelX NAND driver returns ***LX_ERROR***. The prototype of the LevelX NAND driver “block status set” service is:

```
INT nand_driver_block_status_set(ULONG block,
                                 UCHAR bad_block_byte);
```

Where “***block***” specifies which block and “***bad_block_byte***” specifies the value of the bad block flag.

Driver Block Extra Bytes Get

The LevelX NAND driver “block extra bytes get” service is responsible for retrieving extra bytes associated with a specific page of a specific block of the NAND flash. If it is successful, the LevelX NAND driver returns ***LX_SUCCESS***. If it is not successful, the LevelX NAND driver returns ***LX_ERROR***. The prototype of the LevelX NAND driver “block extra bytes get” service is:

```
INT nand_driver_block_extra_bytes_get(ULONG block,
        ULONG page, UCHAR *destination, UINT size);
```

Where “***block***” specifies which block, “***page***” specifies the specific page and “***destination***” specifies the destination for the extra bytes. The parameter “***size***” specifies how many extra bytes to get.

Driver Block Extra Bytes Set

The LevelX NAND driver “block extra bytes set” service is responsible for setting extra bytes in a specific page of a specific block of the NAND flash. If it is successful, the LevelX NAND driver returns ***LX_SUCCESS***. If it is not successful, the LevelX NAND driver returns ***LX_ERROR***. The prototype of the LevelX NAND driver “block extra bytes set” service is:

```
INT nand_driver_block_extra_bytes_set(ULONG block,
            ULONG page, UCHAR *source, UINT size);
```

Where “***block***” specifies which block, “***page***” specifies the specific page and “***source***” specifies the source of the extra bytes. The parameter “***size***” specifies how many extra bytes to set.

Driver System Error

The LevelX NAND driver “system error handler” service is responsible for setting handling system errors detected by LevelX. The processing in this routine is application dependent. If it is successful, the LevelX NAND driver returns ***LX_SUCCESS***. If it is not successful, the LevelX NAND driver returns ***LX_ERROR***. The prototype of the LevelX NAND driver “system error” service is:

```
INT nand_driver_system_error(UINT error_code,
                        ULONG block, ULONG page);
```

Where “***block***” specifies which block, and “***page***” specifies the specific page the error represented by “***error_code***” occurred.

NAND Simulated Driver

LevelX provides a simulated NAND flash driver that simply uses RAM to simulate the operation of a NAND flash part. By default, the NAND simulated driver provides 8 NAND flash blocks with 16 pages per block and 2048 bytes per page.

The simulated NAND flash driver initialization function is ***lx_nand_flash_simulator_initialize*** and is defined in ***lx_nand_flash_simulator.c***. This driver also provides a good template for writing specific NAND flash drivers.

NAND FileX Integration

As mentioned earlier, LevelX does not rely on FileX for operation. All the LevelX APIs may be called directly by the application software to store/retrieve raw data to the logical sectors provided by LevelX. However, LevelX also supports FileX.

The file ***fx_nand_flash_simulated_driver.c*** contains an example FileX driver for use with the NAND flash simulation. An interesting aspect of this driver is that it combines 512-byte logical sectors typically used by FileX into single logical sector read/write requests to the LevelX simulator using 2048-byte pages. This results in more efficient use of the NAND flash memory. The NAND flash FileX driver for LevelX provides a good starting point for writing custom FileX drivers.

Note: The FileX NAND flash format should be one full block size of sectors less than the NAND flash provides. This will help ensure best performance during the wear level processing. Additional techniques to improve write performance in the LevelX wear leveling algorithm include:

1. Ensure that all writes are exactly one or more clusters in size and start on exact cluster boundaries.
2. Pre-allocate clusters before performing large file write operations via the FileX ***fx_file_allocate*** class of APIs.
3. Ensure the FileX driver is enabled to receive release sector information and requests made to the driver to release sectors are handled in the driver by calling ***lx_nor_flash_sector_release***.
4. Periodic use of ***lx_nand_flash_defragment*** to free up as many NAND blocks as possible and thus improve write performance.
5. Utilize the ***lx_nand_flash_extended_cache_enable*** API to provide a RAM cache of various NAND block resources for faster performance.

Chapter 4

LevelX NAND APIs

The LevelX NAND APIs available to the application are:

lx_nand_flash_close
Close NAND flash instance

lx_nand_flash_defragment
Defragment NAND flash instance

lx_nand_flash_extended_cache_enable
Enable/disable extended NAND cache

lx_nand_flash_initialize
Initialize NAND flash support

lx_nand_flash_open
Open NAND flash instance

lx_nand_flash_page_ecc_check
Check page for ECC errors with correction

lx_nand_flash_page_ecc_compute
Computes ECC for page

lx_nand_flash_partial_defragment
Partial defragment of NAND flash instance

lx_nand_flash_sector_read
Read NAND flash sector

lx_nand_flash_sector_release
Release NAND flash sector

lx_nand_flash_sector_write
Write NAND flash sector

lx_nand_flash_close

Close NAND flash instance

Prototype

```
UINT lx_nand_flash_close(LX_NAND_FLASH *nand_flash);
```

Description

This service closes the previously opened NAND flash instance.

Input Parameters

nand_flash	NAND flash instance pointer.

Return Values

LX_SUCCESS	(0x00)	Successful request.
LX_ERROR	(0x01)	Error closing flash instance.

Allowed From

Threads

Example

```
/* Close NAND flash instance "my_nand_flash".  */
status =  lx_nand_flash_close(&my_nand_flash);

/* If status is LX_SUCCESS the request was successful.  */
```

See Also

lx_nand_flash_defragment, lx_nand_flash_extended_cache_enable, lx_nand_flash_initialize, lx_nand_flash_open, lx_nand_flash_page_ecc_check, lx_nand_flash_page_ecc_compute, lx_nand_flash_partial_defragment, lx_nand_flash_sector_read, lx_nand_flash_sector_release, lx_nand_flash_sector_write

lx_nand_flash_defragment

Defragment NAND flash instance

Prototype

```
UINT lx_nand_flash_defragment(LX_NAND_FLASH *nand_flash);
```

Description

This service defragments the previously opened NAND flash instance. The defragment process maximizes the number of free pages and blocks.

Input Parameters

nand_flash	NAND flash instance pointer.

Return Values

LX_SUCCESS	(0x00)	Successful request.
LX_ERROR	(0x01)	Error defragmenting flash instance.

Allowed From

Threads

Example

```
/* Defragment NAND flash instance "my_nand_flash".  */
status =  lx_nand_flash_defragment(&my_nand_flash);

/* If status is LX_SUCCESS the request was successful.  */
```

See Also

lx_nand_flash_close, lx_nand_flash_extended_cache_enable, lx_nand_flash_initialize, lx_nand_flash_open, lx_nand_flash_page_ecc_check, lx_nand_flash_page_ecc_compute, lx_nand_flash_partial_defragment, lx_nand_flash_sector_read, lx_nand_flash_sector_release, lx_nand_flash_sector_write

lx_nand_flash_extended_cache_enable

Enable/disable extended NAND cache

Prototype

```
UINT lx_nand_flash_extended_cache_enable(LX_NAND_FLASH *nand_flash,
                                         VOID *memory, ULONG size);
```

Description

This service implements a cache layer in RAM using the memory supplied by the application. The total amount of memory required for full cache operation can be calculated as follows:

size (in_bytes) = number_of_blocks (rounded up to be divisible by 4) +
((number_of_blocks * pages_per_block) * 4) +
((number_of_blocks * (pages_per_block + 1)) * 4)

If the supplied memory is not large enough to accommodate the full NAND cache, this routine will enable as much of the NAND flash cache as possible based on the memory supplied.

The NAND cache is disabled if the memory address specified is NULL. Hence, the NAND cache maybe be used in a temporary fashion.

Input Parameters

nand_flash	NAND flash instance pointer.
memory	Starting address for cache memory aligned for ULONG access. A value of LX_NULL disables the cache.
size	The size in bytes of the memory supplied.

Return Values

LX_SUCCESS	(0x00)	Successful request.
LX_ERROR	(0x01)	Not enough memory for one element of the NAND cache.

Allowed From

Threads

Example

```
/* Enable the NAND flash cache for the instance
        "my_nand_flash".  */
status =
        lx_nand_flash_extended_cache_enable(&my_nand_flash,
                        &my_memory, sizeof(my_memory));

/* If status is LX_SUCCESS the request was successful.  */
```

See Also

lx_nand_flash_close, lx_nand_flash_defragment, lx_nand_flash_initialize, lx_nand_flash_open, lx_nand_flash_page_ecc_check, lx_nand_flash_page_ecc_compute, lx_nand_flash_partial_defragment, lx_nand_flash_sector_read, lx_nand_flash_sector_release, lx_nand_flash_sector_write

lx_nand_flash_initialize

Initialize NAND flash support

Prototype

```
UINT lx_nand_flash_initialize(void);
```

Description

This service initializes LevelX NAND flash support. It must be called before any other LevelX NAND APIs.

Input Parameters

None

Return Values

LX_SUCCESS	(0x00)	Successful request.
LX_ERROR	(0x01)	Error initializing NAND flash support.

Allowed From

Initialization, Threads

Example

```
/* Initialize NAND flash support.  */
status =  lx_nand_flash_initialize();

/* If status is LX_SUCCESS the request was successful.  */
```

See Also

lx_nand_flash_close, lx_nand_flash_defragment, lx_nand_flash_extended_cache_enable, lx_nand_flash_open, lx_nand_flash_page_ecc_check, lx_nand_flash_page_ecc_compute, lx_nand_flash_partial_defragment, lx_nand_flash_sector_read, lx_nand_flash_sector_release, lx_nand_flash_sector_write

lx_nand_flash_open

Open NAND flash instance

Prototype

```
UINT lx_nand_flash_open(LX_NAND_FLASH *nand_flash, CHAR *name,
                    UINT (*nand_driver_initialize) (LX_NAND_FLASH *));
```

Description

This service opens a NAND flash instance with the specified NAND flash control block and driver initialization function. Note that the driver initialization function is responsible for installing various function pointers for reading, writing, and erasing blocks/pages of the NAND hardware associated with this NAND flash instance.

Input Parameters

nand_flash	NAND flash instance pointer.
name	Name of NAND flash instance.
nand_driver_initialize	Function pointer to NAND flash driver initialization function. Please refer to Chapter 3 of this guide for more details on NAND flash driver responsibilities.

Return Values

LX_SUCCESS	(0x00)	Successful request.
LX_ERROR	(0x01)	Error opening NAND flash instance.
LX_NO_MEMORY	(0x08)	Driver did not provide buffer for reading one page into RAM.

Allowed From

Threads

Example

```
/* Open NAND flash instance "my_nand_flash" with
         the driver "my_nand_driver_initialize".  */
status =  lx_nand_flash_open(&my_nand_flash,"my nand flash",

        my_nand_driver_initialize);

/* If status is LX_SUCCESS the request was successful.  */
```

See Also

lx_nand_flash_close, lx_nand_flash_defragment, lx_nand_flash_extended_cache_enable, lx_nand_flash_initialize, lx_nand_flash_page_ecc_check, lx_nand_flash_page_ecc_compute, lx_nand_flash_partial_defragment, lx_nand_flash_sector_read, lx_nand_flash_sector_release, lx_nand_flash_sector_write

lx_nand_flash_page_ecc_check

Check page for ECC errors with correction

Prototype

```
UINT lx_nand_flash_page_ecc_check(LX_NAND_FLASH *nand_flash,
                                  UCHAR *page_buffer, UCHAR *ecc_buffer);
```

Description

This service verifies the integrity of the supplied NAND page buffer with the supplied ECC. Page size (defined in the NAND flash instance pointer) is assumed to be a multiple of 256-bytes and the ECC code supplied is capable of correcting a 1 bit error in each 256-byte portion of the page.

Input Parameters

nand_flash	NAND flash instance pointer.
page_buffer	Pointer to NAND flash page buffer.
ecc_buffer	Pointer to ECC for NAND flash page. Note that there are 3 ECC bytes per 256-byte portion of the page.

Return Values

LX_SUCCESS	(0x00)	NAND page has no errors.
LX_NAND_ERROR_CORRECTED	(0x06)	One or more 1-bit errors were corrected in the NAND page—correction(s) are in the page buffer.
LX_NAND_ERROR_NOT_CORRECTED	(0x07)	Too many errors to correct on the NAND page.

Allowed From

LevelX Driver

Example

```
/* Check the NAND page pointed to by "page_pointer" of the
   NAND flash instance "my_nand_flash" .  */
status =  lx_nand_flash_page_ecc_check(&my_nand_flash,

        page_pointer, ecc_pointer);

/* If status is LX_SUCCESS the page is valid.  */
```

See Also

lx_nand_flash_close, lx_nand_flash_defragment, lx_nand_flash_extended_cache_enable, lx_nand_flash_initialize, lx_nand_flash_open, lx_nand_flash_page_ecc_compute, lx_nand_flash_partial_defragment, lx_nand_flash_sector_read, lx_nand_flash_sector_release, lx_nand_flash_sector_write

lx_nand_flash_page_ecc_compute

Compute ECC for page

Prototype

```
UINT lx_nand_flash_page_ecc_compute(LX_NAND_FLASH *nand_flash,
                                    UCHAR *page_buffer, UCHAR *ecc_buffer);
```

Description

This service computes the ECC of the supplied NAND page buffer and returns the ECC in the supplied ECC buffer. Page size is assume to be a multiple of 256-bytes (defined in the NAND flash instance pointer). The ECC code is used to verify the integrity of the page when it is read at a later time.

Input Parameters

nand_flash	NAND flash instance pointer.
page_buffer	Pointer to NAND flash page buffer.
ecc_buffer	Pointer to destination for ECC of the NAND flash page. Note that must be 3 bytes of ECC storage per 256-byte portion of the page. For example, a 2048 byte page would require 24 bytes for the ECC.

Return Values

LX_SUCCESS	(0x00)	ECC successfully calculated.
LX_ERROR	(0x01)	Error calculating ECC.

Allowed From

LevelX Driver

Example

```
/* Compute ECC for the NAND page pointed to by
  "page_pointer" of the NAND flash instance
  "my_nand_flash".  */
status =  lx_nand_flash_page_ecc_compute(&my_nand_flash,
        page_pointer, ecc_pointer);

/* If status is LX_SUCCESS the ECC was calculated and
   Can be found in the memory pointed to by
   "ecc_pointer."  */
```

See Also

lx_nand_flash_close, lx_nand_flash_defragment, lx_nand_flash_extended_cache_enable, lx_nand_flash_initialize, lx_nand_flash_open, lx_nand_flash_page_ecc_check, lx_nand_flash_partial_defragment, lx_nand_flash_sector_read, lx_nand_flash_sector_release, lx_nand_flash_sector_write

lx_nand_flash_partial_defragment

Partial defragment of NAND flash instance

Prototype

```
UINT lx_nand_flash_partial_defragment(LX_NAND_FLASH *nand_flash,
                                      UINT max_blocks);
```

Description

This service defragments the previously opened NAND flash instance up to the maximum number of blocks specified. The defragment process maximizes the number of free pages and blocks.

Input Parameters

nand_flash	NAND flash instance pointer.
max_blocks	Maximum number of blocks.

Return Values

LX_SUCCESS	(0x00)	Successful request.
LX_ERROR	(0x01)	Error defragmenting flash instance.

Allowed From

Threads

Example

```
/* Defragment 1 block of NAND flash instance
   "my_nand_flash".  */
status = lx_nand_flash_partial_defragment(&my_nand_flash, 1);

/* If status is LX_SUCCESS the request was successful.  */
```

See Also

lx_nand_flash_close, lx_nand_flash_defragment, lx_nand_flash_extended_cache_enable, lx_nand_flash_initialize, lx_nand_flash_open, lx_nand_flash_page_ecc_check, lx_nand_flash_page_ecc_compute, lx_nand_flash_sector_read, lx_nand_flash_sector_release, lx_nand_flash_sector_write

lx_nand_flash_sector_read

Read NAND flash sector

Prototype

```
UINT lx_nand_flash_sector_read(LX_NAND_FLASH *nand_flash,
                               ULONG logical_sector, VOID *buffer);
```

Description

This service reads the logical sector from the NAND flash instance and if successful returns the contents in the supplied buffer. Note that NAND sector size is always the page size of the underlying NAND hardware.

Input Parameters

nand_flash	NAND flash instance pointer.
logical_sector	Logical sector to read.
buffer	Pointer to destination for contents of the logical sector. Note that the buffer is assumed to be the size of the NAND flash page size and aligned for ULONG access.

Return Values

LX_SUCCESS	(0x00)	Successful request.
LX_ERROR	(0x01)	Error reading NAND flash sector.

Allowed From

Threads

Example

```
/* Read logical sector 20 of the NAND flash instance
  "my_nand_flash" and place contents in "buffer".  */
status =  lx_nand_flash_sector_read(&my_nand_flash,

        20, buffer);

/* If status is LX_SUCCESS, "buffer" contains the contents
   of logical sector 20. */
```

See Also

lx_nand_flash_close, lx_nand_flash_defragment, lx_nand_flash_extended_cache_enable, lx_nand_flash_initialize, lx_nand_flash_open, lx_nand_flash_page_ecc_check, lx_nand_flash_page_ecc_compute, lx_nand_flash_partial_defragment, lx_nand_flash_sector_release, lx_nand_flash_sector_write

lx_nand_flash_sector_release

Release NAND flash sector

Prototype

```
UINT lx_nand_flash_sector_release(LX_NAND_FLASH *nand_flash,
                                  ULONG logical_sector);
```

Description

This service releases the logical sector mapping in the NAND flash instance. Releasing a logical sector when not used makes the LevelX wear leveling more efficient.

Input Parameters

nand_flash	NAND flash instance pointer.
logical_sector	Logical sector to release.

Return Values

LX_SUCCESS	(0x00)	Successful request.
LX_ERROR	(0x01)	Error NAND flash sector write.

Allowed From

Threads

Example

```
/* Release logical sector 20 of the NAND flash instance
  "my_nand_flash".  */
status =  lx_nand_flash_sector_release(&my_nand_flash, 20);

/* If status is LX_SUCCESS, logical sector 20 has been
   released. */
```

See Also

lx_nand_flash_close, lx_nand_flash_defragment, lx_nand_flash_extended_cache_enable, lx_nand_flash_initialize, lx_nand_flash_open, lx_nand_flash_page_ecc_check, lx_nand_flash_page_ecc_compute, lx_nand_flash_partial_defragment, lx_nand_flash_sector_read, lx_nand_flash_sector_write

lx_nand_flash_sector_write

Write NAND flash sector

Prototype

```
UINT lx_nand_flash_sector_write(LX_NAND_FLASH *nand_flash,
                                ULONG logical_sector, VOID *buffer);
```

Description

This service writes the specified logical sector in the NAND flash instance.

Input Parameters

nand_flash	NAND flash instance pointer.
logical_sector	Logical sector to write.
buffer	Pointer to the contents of the logical sector. Note that the buffer is assumed to be the size of the NAND flash page size and aligned for ULONG access.

Return Values

LX_SUCCESS	(0x00)	Successful request.
LX_NO_SECTORS	(0x02)	No more free sectors are available to perform the write.
LX_ERROR	(0x01)	Error releasing NAND flash sector.

Allowed From

Threads

Example

```
/* Write logical sector 20 of the NAND flash instance
  "my_nand_flash" with the contents pointed to by
  "buffer".  */
status =  lx_nand_flash_sector_write(&my_nand_flash,
                                     20, buffer);

/* If status is LX_SUCCESS, logical sector 20 has been
   written with the contents of "buffer". */
```

See Also

lx_nand_flash_close, lx_nand_flash_defragment, lx_nand_flash_extended_cache_enable, lx_nand_flash_initialize, lx_nand_flash_open, lx_nand_flash_page_ecc_check, lx_nand_flash_page_ecc_compute, lx_nand_flash_partial_defragment, lx_nand_flash_sector_read, lx_nand_flash_sector_release

Chapter 5

LevelX NOR Support

NOR flash memory is composed of ***blocks*** that are typically evenly divisible by 512 bytes. There are no concept of a flash ***page*** in NOR flash memory. Also, there are no ***spare*** bytes in NOR flash memory, hence LevelX must use the NOR flash memory itself for all management information. Direct read access is possible in NOR flash memory. Write access typically requires a special sequence of operations. NOR flash memory may be written to multiple times, providing that bits are being cleared. Bits in NOR flash memory can only be set once, via the erase block operation.

LevelX divides each NOR flash block into 512-byte logical ***sectors***. Furthermore, LevelX uses the first “n” sectors of each NOR flash block to store management information. The format of the LevelX NOR flash memory management information is:

LevelX NOR Block Format

Byte Offset	Contents
0	[Block Erase Count]
4	[Minimum Mapped Sector]
8	[Maximum Mapped Sector]
12	[Free Sector Bit Map]
m	[Sector 0 Mapping Entry]
	…
m+4*(n-1)	[Sector “n” Mapping Entry]
	…
s	[Sector 0 Contents]
	…
s+512*(n-1)	[Sector “n” Contents]

The 32-bit ***Block Erase Count*** contains the number of times the block has been erased. The main goal of LevelX is to keep the erase count of all blocks relatively close to help prevent any one block from wearing out prematurely. The 32-bit ***Minimum Mapped Sector*** and ***Maximum Mapped Sector*** fields are written only when all the logical sectors in the block have been mapped and written to. These fields are useful for optimization of the sector read operation. The ***Free Sector Bit Map*** entry is a bit map where each set bit corresponds to an unmapped sector in the

block. This field is used to make the free sector search more efficient. This is a variable length field that requires one word for every 32 sectors in the block. The ***Sector Mapping Entry*** array contains mapping information for each sector in the block. Each entry has the following format:

Sector Mapping Entry

Bit(s)	Meaning
31	Valid flag. When set and logical sector not all ones indicates mapping is valid
30	Obsolete flag. When clear, this mapping is either obsolete or is in the process of becoming obsolete.
29	Mapping entry write is complete when this bit is 0
0-28	Logical sector mapped to this physical sector—when not all ones.

The upper bit of the 32-bit field (bit 31) is used to indicate the logical sector mapping is valid. If this bit is 0, the information in this entry (and corresponding sector contents) is no longer valid. The next bit - bit 30 - is used to indicate this sector is in the process of becoming obsolete and a new sector is being written. Bit 29 is used to indicate when the mapping entry write is complete. If bit 29 is 0, the mapping entry write is complete. If bit 29 is set, the mapping entry was in the process of being written. Bits 30 and 29 are used in recovering from a potential power loss while updating a new sector mapping. Finally, the lower 29-bits (28-0) contain the logical sector number for the sector. If a sector has not been mapped, all bits will be set, i.e., it will have a value of 0xFFFFFFFF.

NOR Driver Requirements

LevelX requires an underlying NOR flash driver that is specific to the underlying flash part and hardware implementation. The driver is specified to LevelX during initialization via the API ***lx_nor_flash_open***. The prototype of the LevelX driver is:

```
INT nor_driver_initialize(LX_NOR_FLASH *instance);
```

The "***instance***" parameter specifies the LevelX NOR control block. The driver initialization function is responsible for setting up all the other driver-level services for the associated LevelX instance. The services required for each LevelX NOR instance are:

Read Sector
Write Sector

Block Erase
Block Erased Verify
System Error Handler

Driver Initialization

These services are setup via setting function pointers in the ***LX_NOR_FLASH*** instance within the driver's initialization function. The driver initialization function also is responsible for:

1. Specifying the base address of the flash.
2. Specifying the total number of blocks and the number of words per block.
3. A RAM buffer for reading one sector of flash (512 bytes) and aligned for ULONG access.

The driver initialization function likely also performs additional device and/or implementation-specific initialization duties before returning ***LX_SUCCESS***.

Driver Read Sector

The LevelX NOR driver "read sector" service is responsible for reading a specific sector in a specific block of the NOR flash. All error checking and correcting logic is the responsibility of the driver service. If successful, the LevelX NOR driver returns ***LX_SUCCESS***. If not successful, the LevelX NOR driver returns ***LX_ERROR***. The prototype of the LevelX NOR driver "read sector" service is:

```
INT nor_driver_read_sector(ULONG *flash_address,
                           ULONG *destination, ULONG words);
```

Where "***flash_address***" specifies the address of a logical sector within a NOR flash block of memory and "***destination***" and "***words***" specify where to place the sector contents and how many 32-bit words to read.

Driver Write Sector

The LevelX NOR driver "write sector" service is responsible for writing a specific sector into a block of the NOR flash. All error checking is the responsibility of the driver service. If successful, the LevelX NOR driver returns ***LX_SUCCESS***. If not successful, the LevelX NOR driver returns ***LX_ERROR***. The prototype of the LevelX NOR driver "write sector" service is:

```
INT nor_driver_write_sector(ULONG *flash_address,
```

```
                                    ULONG *source, ULONG words);
```

Where “***flash_address***” specifies the address of a logical sector within a NOR flash block of memory and “***source***” and “***words***” specify the source of the write and how many 32-bit words to write.

Note: LevelX relies on the driver to verify that the write sector was successful. This is typically done by reading back the programmed value to ensure it matches the requested value to be written.

Driver Block Erase

The LevelX NOR driver “block erase” service is responsible for erasing the specified block of the NOR flash. If successful, the LevelX NOR driver returns ***LX_SUCCESS***. If not successful, the LevelX NOR driver returns ***LX_ERROR***. The prototype of the LevelX NOR driver “block erase” service is:

```
INT nor_driver_block_erase(ULONG block,
                           ULONG erase_count);
```

Where “***block***” identifies which NOR block to erase. The parameter “***erase_count***” is provided for diagnostic purposes. For example, the driver may want to alert another portion of the application software when the erase count exceeds a specific threshold.

Note: LevelX relies on the driver to examine all bytes of the block to ensure they are erased (contain all ones).

Driver Block Erased Verify

The LevelX NOR driver “block erased verify” service is responsible for verifying that the specified block of the NOR flash is erased. If it is erased, the LevelX NOR driver returns ***LX_SUCCESS***. If the block is not erased, the LevelX NOR driver returns ***LX_ERROR***. The prototype of the LevelX NOR driver “block erased verify” service is:

```
INT nor_driver_block_erased_verify(ULONG block);
```

Where “***block***” specifies which block to verify that it is erased.

Note: LevelX relies on the driver to examine all bytes of the specified to ensure they are erased (contain all ones).

Driver System Error

The LevelX NOR driver “system error handler” service is responsible for setting handling system errors detected by LevelX. The processing in this routine is application dependent. If it is successful, the LevelX NOR driver returns ***LX_SUCCESS***. If it is not successful, the LevelX NOR driver returns ***LX_ERROR***. The prototype of the LevelX NOR driver “system error” service is:

```
INT nor_driver_system_error(UINT error_code);
```

Where “***error_code***” represents the error that occurred.

NOR Simulated Driver

LevelX provides a simulated NOR flash driver that simply uses RAM to simulate the operation of a NOR flash part. By default, the NOR simulated driver provides 8 NOR flash blocks with 16 512-byte sectors per block.

The simulated NOR flash driver initialization function is ***lx_nor_flash_simulator_initialize*** and is defined in ***lx_nor_flash_simulator.c***. This driver also provides a good template for writing specific NOR flash drivers.

NOR FileX Integration

As mentioned earlier, LevelX does not rely on FileX for operation. All the LevelX APIs may be called directly by the application software to store/retrieve raw data to the logical sectors provided by LevelX. However, LevelX also supports FileX.

The file ***fx_nor_flash_simulated_driver.c*** contains an example FileX driver for use with the NOR flash simulation. The NOR flash FileX driver for LevelX provides a good starting point for writing custom FileX drivers.

Note: The FileX NOR flash format should be one full block size of sectors less than the NOR flash provides. This will help ensure best performance during the wear level processing. Additional techniques to improve write performance in the LevelX wear leveling algorithm include:

1. Ensure that all writes are exactly one or more clusters in size and start on exact cluster boundaries.
2. Pre-allocate clusters before performing large file write operations via the FileX ***fx_file_allocate*** class of APIs.
3. Periodic use of ***lx_nor_flash_defragment*** to free up as many NOR blocks as possible and thus improve write performance.
4. Ensure the FileX driver is enabled to receive release sector information and requests made to the driver to release sectors are handled in the driver by calling ***lx_nor_flash_sector_release***.

Chapter 6

LevelX NOR APIs

The LevelX NOR APIs available to the application are:

lx_nor_flash_close
Close NOR flash instance

lx_nor_flash_defragment
Defragment NOR flash instance

lx_nor_flash_extended_cache_enable
Enable/disable extended NOR cache

lx_nor_flash_initialize
Initialize NOR flash support

lx_nor_flash_open
Open NOR flash instance

lx_nor_flash_partial_defragment
Partial defragment of NOR flash instance

lx_nor_flash_sector_read
Read NOR flash sector

lx_nor_flash_sector_release
Release NOR flash sector

lx_nor_flash_sector_write
Write NOR flash sector

lx_nor_flash_close

Close NOR flash instance

Prototype

```
UINT lx_nor_flash_close(LX_NOR_FLASH *nor_flash);
```

Description

This service closes the previously opened NOR flash instance.

Input Parameters

nor_flash	NOR flash instance pointer.

Return Values

LX_SUCCESS	(0x00)	Successful request.
LX_ERROR	(0x01)	Error closing flash instance.

Allowed From

Threads

Example

```
/* Close NOR flash instance "my_nor_flash".  */
status =  lx_nor_flash_close(&my_nor_flash);

/* If status is LX_SUCCESS the request was successful.  */
```

See Also

lx_nor_flash_defragment, lx_nor_flash_extended_cache_enable, lx_nor_flash_initialize, lx_nor_flash_open, lx_nor_flash_partial_defragment, lx_nor_flash_sector_read, lx_nor_flash_sector_release, lx_nor_flash_sector_write

lx_nor_flash_defragment

Defragment NOR flash instance

Prototype

```
UINT lx_nor_flash_defragment(LX_NOR_FLASH *nor_flash);
```

Description

This service defragments the previously opened NOR flash instance. The defragment process maximizes the number of free sectors and blocks.

Input Parameters

nor_flash	NOR flash instance pointer.

Return Values

LX_SUCCESS	(0x00)	Successful request.
LX_ERROR	(0x01)	Error defragmenting flash instance.

Allowed From

Threads

Example

```
/* Defragment NOR flash instance "my_nor_flash".  */
status =  lx_nor_flash_defragment(&my_nor_flash);

/* If status is LX_SUCCESS the request was successful.  */
```

See Also

lx_nor_flash_close, lx_nor_flash_extended_cache_enable, lx_nor_flash_initialize, lx_nor_flash_open, lx_nor_flash_partial_defragment, lx_nor_flash_sector_read, lx_nor_flash_sector_release, lx_nor_flash_sector_write

lx_nor_flash_extended_cache_enable

Enable/disable extended NOR cache

Prototype

```
UINT lx_nor_flash_extended_cache_enable(LX_NOR_FLASH *nor_flash,
                                        VOID *memory, ULONG size);
```

Description

This service implements a NOR sector cache layer in RAM using the memory supplied by the application. Each 512 bytes of memory supplied translates to one NOR sector that can be cached. The sectors cached are those that contain the block control information, e.g., erase count, free sector map, and sector mapping information. Data sectors are not stored in this cache.

Input Parameters

nor_flash	NOR flash instance pointer.
memory	Starting address for cache memory, aligned for ULONG access. A value of LX_NULL disables the cache.
size	The size in bytes of the memory supplied (should be multiple of 512 bytes).

Return Values

LX_SUCCESS	(0x00)	Successful request.
LX_ERROR	(0x01)	Not enough memory for one NOR sector.
LX_DISABLED	(0x09)	NOR extended cache disabled by configuration option.

Allowed From

Threads

Example

```
/* Enable the NOR flash cache for the instance
        "my_nor_flash".  */
status =
        lx_nor_flash_extended_cache_enable(&my_nor_flash,
                        &my_memory, sizeof(my_memory));

/* If status is LX_SUCCESS the request was successful.  */
```

See Also

lx_nor_flash_close, lx_nor_flash_defragment, lx_nor_flash_initialize, lx_nor_flash_open, lx_nor_flash_partial_defragment, lx_nor_flash_sector_read, lx_nor_flash_sector_release, lx_nor_flash_sector_write

lx_nor_flash_initialize

Initialize NOR flash support

Prototype

```
UINT lx_nor_flash_initialize(void);
```

Description

This service initializes LevelX NOR flash support. It must be called before any other LevelX NOR APIs.

Input Parameters

None

Return Values

LX_SUCCESS	(0x00)	Successful request.
LX_ERROR	(0x01)	Error initializing NOR flash support.

Allowed From

Initialization, Threads

Example

```
/* Initialize NOR flash support.  */
status =  lx_nor_flash_initialize();

/* If status is LX_SUCCESS the request was successful.  */
```

See Also

lx_nor_flash_close, lx_nor_flash_defragment, lx_nor_flash_extended_cache_enable, lx_nor_flash_partial_defragment, lx_nor_flash_open, lx_nor_flash_sector_read, lx_nor_flash_sector_release, lx_nor_flash_sector_write

lx_nor_flash_open

Open NOR flash instance

Prototype

```
UINT lx_nor_flash_open(LX_NOR_FLASH *nor_flash, CHAR *name,
                       UINT (*nor_driver_initialize) (LX_NOR_FLASH *));
```

Description

This service opens a NOR flash instance with the specified NOR flash control block and driver initialization function. Note that the driver initialization function is responsible for installing various function pointers for reading, writing, and erasing blocks of the NOR hardware associated with this NOR flash instance.

Input Parameters

nor_flash	NOR flash instance pointer.
name	Name of NOR flash instance.
nor_driver_initialize	Function pointer to NOR flash driver Initialization function. Please refer to Chapter 5 of this guide for more details on NOR flash driver responsibilities.

Return Values

LX_SUCCESS	(0x00)	Successful request.
LX_ERROR	(0x01)	Error opening NOR flash instance.
LX_NO_MEMORY	(0x08)	Driver did not provide buffer for reading one sector into RAM.

Allowed From

Threads

Example

```
/* Open NOR flash instance "my_nor_flash" with
         the driver "my_nor_driver_initialize".  */
status =  lx_nor_flash_open(&my_nor_flash,"my NOR flash",

        my_nor_driver_initialize);

/* If status is LX_SUCCESS the request was successful.  */
```

See Also

lx_nor_flash_close, lx_nor_flash_defragment, lx_nor_flash_extended_cache_enable, lx_nor_flash_initialize, lx_nor_flash_partial_defragment, lx_nor_flash_sector_read, lx_nor_flash_sector_release, lx_nor_flash_sector_write

lx_nor_flash_partial_defragment

Partial defragment of NOR flash instance

Prototype

```
UINT lx_nor_flash_partial_defragment(LX_NOR_FLASH *nor_flash, UINT max_blocks);
```

Description

This service defragments the previously opened NOR flash instance up to the maximum number of blocks specified. The defragment process maximizes the number of free sectors and blocks.

Input Parameters

nor_flash	NOR flash instance pointer.
max_blocks	Maximum number of blocks.

Return Values

LX_SUCCESS	(0x00)	Successful request.
LX_ERROR	(0x01)	Error defragmenting flash instance.

Allowed From

Threads

Example

```
/* Defragment of one block in NOR flash instance
        "my_nor_flash".  */
status =  lx_nor_flash_partial_defragment(&my_nor_flash, 1);

/* If status is LX_SUCCESS the request was successful.  */
```

See Also

lx_nor_flash_close, lx_nor_flash_defragment, lx_nor_flash_extended_cache_enable, lx_nor_flash_initialize, lx_nor_flash_open, lx_nor_flash_sector_read, lx_nor_flash_sector_release, lx_nor_flash_sector_write

lx_nor_flash_sector_read

Read NOR flash sector

Prototype

```
UINT lx_nor_flash_sector_read(LX_NOR_FLASH *nor_flash,
                              ULONG logical_sector, VOID *buffer);
```

Description

This service reads the logical sector from the NOR flash instance and if successful returns the contents in the supplied buffer. Note that NOR sector size is always 512 bytes.

Input Parameters

nor_flash	NOR flash instance pointer.
logical_sector	Logical sector to read.
buffer	Pointer to destination for contents of the logical sector. Note that the buffer is assumed to be 512 bytes and aligned for ULONG access.

Return Values

LX_SUCCESS	(0x00)	Successful request.
LX_ERROR	(0x01)	Error reading NOR flash sector.

Allowed From

Threads

Example

```
/* Read logical sector 20 of the NOR flash instance
  "my_nor_flash" and place contents in "buffer".  */
status =  lx_nor_flash_sector_read(&my_nor_flash,

        20, buffer);

/* If status is LX_SUCCESS, "buffer" contains the contents
   of logical sector 20. */
```

See Also

lx_nor_flash_close, lx_nor_flash_defragment, lx_nor_flash_extended_cache_enable, lx_nor_flash_initialize, lx_nor_flash_open, lx_nor_flash_partial_defragment, lx_nor_flash_sector_release, lx_nor_flash_sector_write

lx_nor_flash_sector_release

Release NOR flash sector

Prototype

```
UINT lx_nor_flash_sector_release(LX_NOR_FLASH *nor_flash,
                                 ULONG logical_sector);
```

Description

This service releases the logical sector mapping in the NOR flash instance. Releasing a logical sector when not used makes the LevelX wear leveling more efficient.

Input Parameters

nor_flash	NOR flash instance pointer.
logical_sector	Logical sector to release.

Return Values

LX_SUCCESS	(0x00)	Successful request.
LX_ERROR	(0x01)	Error NOR flash sector write.

Allowed From

Threads

Example

```
/* Release logical sector 20 of the NOR flash instance
  "my_nor_flash".  */
status =  lx_nor_flash_sector_release(&my_nor_flash, 20);

/* If status is LX_SUCCESS, logical sector 20 has been
   released. */
```

See Also

lx_nor_flash_close, lx_nor_flash_defragment, lx_nor_flash_extended_cache_enable, lx_nor_flash_initialize, lx_nor_flash_open, lx_nor_flash_partial_defragment, lx_nor_flash_sector_read, lx_nor_flash_sector_write

lx_nor_flash_sector_write

Write NOR flash sector

Prototype

```
UINT lx_nor_flash_sector_write(LX_nor_FLASH *NOR_flash,
                               ULONG logical_sector, VOID *buffer);
```

Description

This service writes the specified logical sector in the NOR flash instance.

Input Parameters

nor_flash	NOR flash instance pointer.
logical_sector	Logical sector to write.
buffer	Pointer to the contents of the logical sector. Note that the buffer is assumed to be 512 bytes aligned for ULONG access.

Return Values

LX_SUCCESS	(0x00)	Successful request.
LX_NO_SECTORS	(0x02)	No more free sectors are available to perform the write.
LX_ERROR	(0x01)	Error releasing NOR flash sector.

Allowed From

Threads

Example

```
/* Write logical sector 20 of the NOR flash instance
  "my_nor_flash" with the contents pointed to by
  "buffer".  */
status =  lx_nor_flash_sector_write(&my_nor_flash,
                                    20, buffer);

/* If status is LX_SUCCESS, logical sector 20 has been
   written with the contents of "buffer". */
```

See Also

lx_nor_flash_close, lx_nor_flash_defragment, lx_nor_flash_extended_cache_enable, lx_nor_flash_initialize, lx_nor_flash_open, lx_nor_flash_partial_defragment, lx_nor_flash_sector_read, lx_nor_flash_sector_release

Index