

NAND Flash Translation Layer (NFTL) 4.6.0

User Guide

Introduction

This user guide describes how to implement the Micron NAND Flash Translation Layer (NFTL) software for the purpose of using a NAND Flash memory device for data storage.

NFTL is a software sector manager that resides between a FAT (or sector-based) file system and Flash memory to enable Micron NAND Flash memory. The NFTL provides a small RAM/ROM footprint, consistent performance, and sector-based power loss recovery. With its user-friendly application program interface (API), Micron NFTL enables standard FAT-based file systems for Flash support. In addition, the NFTL API supports the storage of simple, numerical data parameters.

Nonvolatile memory such as NAND Flash can permanently store data or code without the need for a constant source of power. This capability is ideal for portable applications such as digital cameras, MP3 players, PDAs, and data storage in mobile phones. However, the Flash technology requires additional software to manage the data.

For this reason, the NFTL software handles all operations required to manage embedded Flash memory devices. This eliminates the need for a developer to write additional code during the development cycle. Using this software, a developer needs only to use simple file system commands to interact with a NAND Flash memory device.

The NFTL software handles all management functions so that operating systems can read and write to NAND Flash memory devices in the same way as reading and writing to disk drives. NFTL has been fully tested and debugged for use with a wide range of Micron NAND Flash memory devices, including single-level cell (SLC) large/small page family chips. The size of the NFTL code is around 300KB, but can vary slightly depending on the user's configuration.

NFTL offers the best combination of Flash memory management software and NAND Flash memory devices to use in-system in consumer, automotive, mobile phone, and mass storage applications. The software minimizes the cost of embedded devices, as it allows the user to choose the most cost-effective NAND Flash memory in any density.

The architecture of NFTL allows Micron NAND Flash to be easily integrated with most standard file systems. Minimized porting and engineering efforts helps reduce time-to-market.

In addition, NFTL has a modular architecture. Each module can be changed without affecting other modules. Also, each module can be extracted to be used independently in a different environment. For each software module, NFTL provides functions that can be defined by the user and allows certain hardware and software features to be configured. The RAM space and the device operation settings are configured simply by changing a few lines of code.

Contents

Using NAND Flash Memory For Data Storage	4
NFTL Architecture	5
Flash Abstraction Layer (FAL)	5
Translation Module	6
Wear Leveling Module	6
Garbage Collection Module	6
Bad Block Management Module	6
Background Task Module	6
Cache Manager Module	6
Hardware Adaptation Layer (HAL)	7
Device Dependent Module	7
Platform Dependent Layer	7
Error Correction Code (ECC) Module	7
Module Interfaces	7
NFTL Software Features	9
System Requirements	11
Interface Description	12
NFTL Exported Functions	12
Enumerations, Constants, and Structures	13
API Functions	15
NFTL_Initialize Function	15
NFTL_FormatPartition Function	15
NFTL_MountPartition Function	16
NFTL_UnmountPartition Function	16
NFTL_WriteSector Function	17
NFTL_WriteSectors Function	17
NFTL_ReadSector Function	18
NFTL_DeleteSector Function	18
NFTL_DefragPartition Function	19
NFTL_GetPartitionStatus Function	20
NFTL_GetPartitionsInfo Function	20
NFTL_IsPartitionMounted Function	20
NFTL_DeInit Function	21
NFTL_StartBackground Function	21
NFTL_StopBackground Function	22
NFTL_IsBackgroundRunning Function	22
Getting Started	24
User Configuration	24
Unsupported Configurations for common.h	24
NFTL Allowed Configurations	26
Special Features	26
PL Features	26
Other Customization	26
Porting Guide	27
Integrating NFTL into a System	27
Configuring the OS Adaptation Module	27
Selecting the Correct Device Driver	28
Setting Up the NAND Platform Dependent Driver	28
Setting Up the OneNAND™ Platform Dependent Driver	29
Maximum Number of Erasable Blocks (OneNAND™ Devices Only)	30

Programming Guide	31
Normal Use of NFTL V4.6.0	31
Step 1: Create the t_partitionInfo structure	31
Step 2: NFTL_Initialize	32
Step 3: NFTL_MountPartition	32
Step 4: Function operation	32
Step 5: NFTL_DeInit	32
Demo code for the five steps	32
First Use of NFTL V4.6.0 on a Previously-used NAND Flash Memory	35
Debug Mode Using NFTL V4.6.0	36
Demo code	36
Definitions and Acronyms	38
Source Code Organization	40
NFTL Frequently Asked Questions	42
What are the advantages of using NFTL?	42
Does NFTL support large page (2112-byte) and small page (528-byte) NAND Flash memory devices?	42
How does NFTL manage bad blocks?	42
How many bad blocks can be managed by NFTL?	42
What is the sector size managed by NFTL? Is it possible to change it?	42
When does NFTL perform garbage collection?	42
How much RAM is required to perform NFTL data management correctly?	42
If the NAND Flash device is unformatted, is it possible to manage data storage on it through NFTL?	43
If the NAND Flash device is unformatted, is it possible to manage data storage on it through NFTL?	43
Does NFTL use error correction code (ECC)?	43
Can custom ECC hardware be used with the NFTL software?	43
Does NFTL support wear leveling?	43
Does NFTL support power loss recovery (PLR)?	43
What are the operations required to mount NFTL to a file system?	43
How much NAND physical space is accessible by the user to manage data storage (through NFTL)?	44
References	45
Revision History	46
Rev. L, 02/11	46
Rev. K, 08/10	46
Rev. J, 07/10	46
Rev. 9, 03/10	47
Rev. 8, 11/09	47
Rev. 7, 07/09	48
Rev. 6, 05/09	48
Rev. 5, 04/09	48
Rev. 4, 04/09	48
Rev. 3, 12/08	48
Rev. 2, 10/08	48
Rev. 1, 03/07	48
Disclaimer	49

Using NAND Flash Memory For Data Storage

Files in embedded file systems read and write data in small blocks called “sectors” (typically 512 bytes, 1024 bytes or 2048 bytes in size). Unlike disk drives, NAND Flash memory is not re-writable. Instead, data that is already written must be erased before executing a new Write command.

NAND Flash is writable only one page at a time and it is erasable by blocks that are composed of pages (for more information, refer to the data sheets listed in References (page 45)). Typically, the sector size is equal to the physical page size. This means that each erase operation involves more than one sector.

To overcome the physical organization of NAND Flash devices, there are two solutions that can be implemented:

- Use a translation layer between the file system and storage media to mask any differences between Flash memory devices and traditional storage devices.
- Develop a new file system that is customized for Flash device characteristics.

The NAND Flash Translation Layer (NFTL) software implements the first solution. NFTL emulates the rewriting of the sectors by remapping the new write process to another location in the Flash memory. The previously written sector is marked “not valid” and is erased at a later time. This NFTL technique frees unusable space in the memory device.

The “P/E cycles” are the number of possible WRITE/ERASE operations on a block. For a single level cell (SLC) NAND device it is equal to 100,000 cycles. Functionality of the device beyond this threshold is not guaranteed.

One of the objectives of NFTL is to guarantee that the flash medium is uniformly used. That is, each block is erased the same number of times. In addition, NFTL extracts the hardware functionality of Flash devices through a software module that manages the low-level functionality of the storage media. This module also contains software for managing various possible errors during the flash operation. NFTL ensures maximum functionality and the best possible performance for all Micron Flash memory devices.

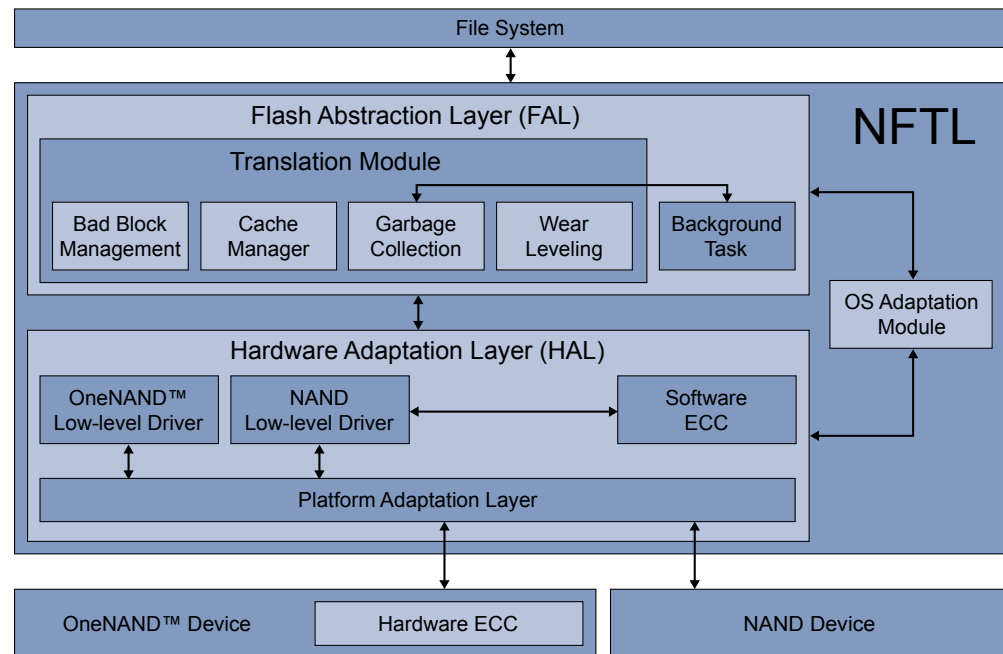
NFTL Architecture

The NFTL software, which interacts directly with the file system, consists of two layers:

- Flash Abstraction Layer (FAL)
- Hardware Adaptation Layer (HAL)

The following figure shows how the software is used in an embedded operating system (OS) that uses NAND Flash memory for data storage.

Figure 1: Design Architecture



Flash Abstraction Layer (FAL)

The Flash Abstraction Layer (FAL) provides a high-level abstraction of the physical organization of NAND Flash memory devices. It emulates the rewriting of sectors in hard disks by remapping new data to another location in the memory array and marking the previous sector invalid. The FAL contains the following modules:

- Translation
- Wear Leveling
- Garbage Collection
- Bad Block Management
- Cache Manager
- Background Task

In addition, the FAL includes an algorithm that recovers the most current data when a sudden power loss occurs.

Translation Module

The Translation module, which is the primary interface in the FAL, provides the translation from virtual to physical addresses and converts the logical operations into physical operations on the Flash memory device. It also handles the exporting of all operations available on storage media (for example: write sector, read sector, and format partition).

Wear Leveling Module

In Micron NAND Flash memory, each physical block can be programmed or erased reliably over 100,000 times for SLC chips. For write-intensive applications, it is recommended that a wear leveling algorithm be implemented to monitor and spread the number of PROGRAM/ERASE cycles per block.

In memory devices that do not use a wear leveling algorithm, not all blocks are used at the same rate. Blocks with long-lived, static data do not endure as many WRITE cycles as the blocks with frequently-changed data.

The Wear Leveling module ensures that the memory array is used uniformly by monitoring and evenly distributing the number of erase cycles per block. Each time a block is requested by the Translation module, the Wear Leveling module allocates the least used block (for more information, refer to the *AN1822 Wear Leveling in NAND Flash Memory application note*).

Garbage Collection Module

As the FAL emulates rewriting sectors in hard disks by remapping new data to another location of the memory array and marking the previous sector invalid, eventually it may be necessary to free some of the invalid memory space to allow further data to be written. To do this, the FAL implements the Garbage Collection module, where the valid sectors are copied into a new free area and the old area is erased.

The Garbage Collection module can be invoked from external modules through the Translation Module interface (for more information, refer to the *AN1821 Garbage Collection in Single Level Cell NAND Flash Memory application note*).

Bad Block Management Module

The Bad Block Management module determines how to set a block as bad.

Bad Blocks are blocks that contain one or more invalid bits whose reliability is not guaranteed. Bad Blocks may be present when the device is shipped, or may develop during the lifetime of the device. The Bad Block Management module hides the bad blocks from the FAL, preventing the FAL from accessing them.

Background Task Module

NFTL's Background Task module allows the execution of garbage collection on NFTL partitions while the system is in an idle state. This feature reduces the risk of having free space while writing, thus improving the user's perception of system performance.

Cache Manager Module

The Cache Manager module manages the internal data regarding the mapping of sectors to physical pages. It allows for faster lookups of most used sectors or all sectors according to the maxCacheEntries parameter of the partition. This is achieved by reducing the number of READ accesses to the device.

Hardware Adaptation Layer (HAL)

The Hardware Adaptation Layer (HAL) manages the hardware functions of the NAND Flash device. It defines a standard device interface, a family-dependent layer of device drivers (that is, NAND vs OneNAND™), and a further layer that manages platform-specific hardware functionality. To increase the reliability of the system, it also implements features to manage error bits, such as error correction code (ECC). These features are necessary because, like hard disks, the NAND Flash device may contain some invalid blocks.

Device Dependent Module

This layer manages the differences between NAND and OneNAND™ devices. Also, it is in charge of recognizing the specific devices during run time.

Platform Dependent Layer

The Platform Dependent Layer manages the platform-dependent features, including the support of hardware ECC. This layer is strongly customer-dependent, and as such, must be modified/updated/rewritten according to customer needs.

For more information about customizing the Platform Dependent Layer, refer to Porting Guide (page 27).

Error Correction Code (ECC) Module

When digital data is stored in a memory device, it is crucial to have a mechanism that can detect and correct a certain number of errors. The Error Correction Code (ECC) module encodes data in such a way that it can identify and correct certain errors in the data. If the ECC does not correct the error successfully, the FAL will return a message indicating that the operation has failed.

The ECC code implemented in NFTL is able to correct 1-bit errors and detect 2-bit errors per each 512 bytes of data.

Module Interfaces

The following table describes the modules and how they interact.

Table 1: Module Interface Information

Layer	Module	Accesses	Starting Points
FAL	Bad Block Management	Hardware Adaptation Module	When the HAL detects errors in the LLD operation.
FAL	Background Task	Garbage Collection	According to the overall OS scheduling policy.
FAL	Cache Manager	Translation Module	On each READ/WRITE access to NFTL.
HAL	Error Correction Code	Hardware Adaptation Module	During READ or WRITE operations to correct errors.
FAL	Garbage Collector	Translation Module	When requested from the external operation or when a threshold is reached.
HAL	Platform Dependent Layer	Hardware Adaptation Module Advanced Logic	For each access to the NAND Flash memory device.

Table 1: Module Interface Information (Continued)

Layer	Module	Accesses	Starting Points
FAL	Translation Interface	External Modules	During external operation.
FAL	Wear Leveling	Translation Module	For each allocation of a new block.

NFTL Software Features

The NFTL software supports the following features:

- **Configurable:** For each software module, NFTL provides functions that can be defined by the user. These functions allow certain hardware and software features to be configured. The RAM space and device operation settings can be configured by simply changing a few lines of code.
- **Simple integration:** The NFTL architecture allows Micron NAND Flash memory devices to be easily integrated with most standard file systems. It has a user-friendly API that supports storage of simple, numerical data parameters.

Minimized porting and engineering efforts help reduce the time-to-market.

- **Modular architecture:** NFTL has a modular architecture. Each module can be changed without affecting other modules, or can be extracted to be used independently in a different environment.
- **Power loss recovery (PLR):** NFTL is able to recognize the system state at each power-up, even if a sudden power loss occurs. In addition, if data is corrupted, NFTL recovers the valid data and erases the remaining data.

Before using a new free block, NFTL will erase it. A write confirm called COMMIT is used to ensure that data is valid when writing the data.

- **Data integrity:** NFTL maintains data integrity through the Bad Block Management and Error Correction Code (ECC) modules.
- **Garbage Collection:** The Garbage Collector module frees occupied NAND flash physical space for disk emulation.
- **Garbage Collection in background:** NFTL offers the capability to use the multithreading features of the underlying operating system to improve performance when dirty space must be collected and cleared. Using specific APIs, it is possible to start and stop background garbage collection while the system is up and running, and start the thread as soon as the partition has been mounted. For further customization to suit specific needs, it is also possible to manage the thread independently per each partition, allowing garbage collection in the background for one or more partitions on the device.
- **Wear leveling:** NFTL guarantees uniform usage of each physical block, prolonging the device lifetime.
- **Multi block erase:** This is a feature of OneNAND™. Please refer to the applicable data sheet for more information.
- **Operation logging:** The last operation performed by NFTL is saved. This is useful for remote debugging.
- **Static wear leveling:** Using this method, long-lived static data is copied to another block so that the original block can be used for data that is changed more frequently. This is triggered when the difference between the maximum and the minimum number of write cycles per block reaches a specific threshold. With this particular technique, the mean age of physical NAND blocks is constantly maintained.
- **Debug messages:** Debug messages can be printed to the screen and can be enabled/disabled for any module.
- **Multi-plane support:** Some devices have a multi-plane feature that allows a multi-device operation on a block located on a different plane. Please refer to the applicable datasheet for more details.

- **Bad block table:** The list of bad blocks is saved in a bad block table that is stored at a location in the device.
- **Advanced bad block marking:** NFTL adds information to bad blocks in order to store the reason the block was marked as a bad block.

A bad block can be generated due to one of the following events:

- Factory shipped: The block is marked as bad from the factory.
- Write fail: A PROGRAM operation on a device failed. Valid data on the block is saved and the block is marked as bad.
- Erase fail: A block ERASE operation failed and the block is marked as bad.
- Read fail: A READ operation failed due to multiple 1-bit ECC errors.

Bad blocks are marked in a field of the spare area. This field is specified in the data-sheet for each device. A block is consider bad if the value in this field does not contain 0xFFh.

The following values in this field indicate the failed operation and the cause for the bad block:

- BAD_BLOCK_FACTORY_SHIPPED 0x00
- BAD_BLOCK_AFTER_WRITE 0x01
- BAD_BLOCK_AFTER_ERASE 0x02
- BAD_BLOCK_AFTER_READ 0x04



- **Refresh on 1-bit ECC errors:** When a correctable 1-bit ECC error occurs during a READ operation, any data in the block containing the error is moved to a free block. The block in which the 1-bit error occurred is then erased.
- **Safe write:** The combination of the define SAFE_WRITE and the field safepartition in t_partitionInfo are used to identify a safe partition. In the safe partition, a ready verify is performed for each WRITE operation. If a compare error occurs, the sector is rewritten and the block is marked as a bad block.
- **Erase verify:** A commit mark is added to erased blocks. This feature increases the power loss safe level of NFTL.
- **Multiple partition types:** NFTL supports two different types of partitions, allowing data to be stored with different policies:
 - *Standard partitions:* These partitions offer the entire set of power loss recovery and data preservation features of NFTL. They can be used to store on the NAND Flash any data that must be preserved between sessions.
 - *Swap partitions:* This type of partition does not offer any data preservation between sessions or power loss recovery on data. However, it offers high READ/WRITE performances and can be used to store session-related data on the NAND device.

System Requirements

The size of the NFTL code depends on the user-defined configuration. However, the code size is typically about 200KB. RAM is required to execute the NFTL code and to assist NFTL in managing storage to the NAND flash memory device.

RAM usage can range from about 30KB to 80KB, depending on enabled features, configuration, and the quantity of data written in the NAND Flash device.

This RAM is used as shown below:

- Approximately 20KB are used to execute the NFTL code.
- Approximately 20KB are used to contain system static variables.
- Approximately 26KB to 65KB are used for system dynamic variables. This space in the RAM is generally required to ensure a high throughput for each (typical usage) read/write operation.

NFTL uses these RAM variables to recognize the system status and to convert logical operations to physical operations. As a result, NFTL does not need to access the NAND Flash.

Note: The compiled code size was calculated for NFTL running on a development board with a Mainstone II board running the Nucleus operating system over an x16 NAND device. The sizes provided are for reference only. The code size depends on the enabled features, configuration, the processor's instruction set, and compiler optimization.

Interface Description

NFTL manages up to 255 partitions in a single NAND device, and supports all file systems that use:

- 512-byte sectors on SLC, small page NAND devices
- 2,048-byte sectors on large page SLC NAND Flash
- 4,096-byte sectors on large page SLC NAND flash using the multi-plane feature

The main interfaces exported by NFTL through the Translation module interface are:

- [Format Partition](#)
- [Sector Read](#)
- [Sector Write](#)
- [Multiple Sector Write](#)
- [Sector Delete](#)

The file system (FS) requirements include operation execution without knowing any specific information about the selected devices. As a result:

- The FAL is concerned with abstracting data to the NAND Flash architecture.
- The HAL executes the operations in the NAND Flash memory.

Another interface allows the file system to defragment the NAND Flash device to increase free space and reduce the size of the data structure that maintains the sector history on the device. This operation increases the throughput and improves software performance.

NFTL Exported Functions

```
ubyte NFTL_IsBackgroundRunning(
    ubyte
)
t_nftl_error NFTL_StartBackground(
    ubyte
)
t_nftl_error NFTL_StopBackground(
    ubyte
)
t_nftl_error NFTL_Initialize(
    ubyte,
    t_partitionInfo*
)
t_nftl_error NFTL_DeInit(
)
t_nftl_error NFTL\_FormatPartition(
    ubyte,
    uword
)
t_nftl_error NFTL\_MountPartition(
    ubyte
)
t_nftl_error NFTL\_UnmountPartition(
    ubyte
)
t_nftl_error NFTL\_WriteSector(
    ubyte,
    uword,
    ubyte *
)
```

```

t_nftl_error NFTL_WriteSectors(
    ubyte,
    udword,
    ubyte,
    ubyte*
)
t_nftl_error NFTL_ReadSector(
    ubyte,
    udword,
    ubyte *
)
t_nftl_error NFTL_DeleteSector(
    ubyte,
    udword
)
t_nftl_error NFTL_DefragPartition(
    ubyte
)
t_nftl_error NFTL_GetPartitionStatus(
    ubyte,
    t_partitionStatus
)
t_nftl_error NFTL_GetPartitionsInfo(
    t_partitionStatus**
)
t_nftl_error NFTL_IsPartitionMounted(
    ubyte
)

```



Enumerations, Constants, and Structures

```

t_nftl_error
typedef enum {

/*The following values are potentially returned from NFTL APIs.*/
    NFTL_SUCCESS=0,
    NFTL_FAILURE=1,
    NFTL_INVALID_PARTITION=2,
    NFTL_INVALID_ADDRESS=3,
    NFTL_FLUSH_ERROR=5,
    NFTL_UNFORMATTED=6,
    NFTL_MOUNTED_PARTITION=0,
    NFTL_UNMOUNTED_PARTITION=1,
    NFTL_VERIFY_ERROR=0x18,
    NFTL_BLANK_CHECK_FAIL=0x20,
    NFTL_DISK_ERROR=0x21,

}t_nftl_error;

typedef struct {
    ubyte number;
    ubyte safePartition;
    udword startBlock;
    udword sizeInBlocks;
    udword numOfPageInBlock;
    udword fillFactor;
    ubyte blankCheckEnabled;
    ubyte EnableBackground;
    ubyte partition_type;
    udword maxCacheEntries;
} t_partitionInfo;

t_partitionStatus
typedef struct{

```

```
uword freeBlocks;  
uword badBlocks;  
uword hiddenBlocks;  
uword occupiedBlocks;  
}t_partitionStatus
```

Note: The blankCheckEnabled value must be set to 0. It will be used in future releases of NFTL.

Note: For more information about these fields and structures, refer to Normal Use of NFTL V4.6.0 (page 31), Debug Mode Using NFTL V4.6.0 (page 36), and Debug Mode Using NFTL V4.6.0 (page 36).

API Functions

The following sections provide details about each of NFTL's API functions.

NFTL_Initialize Function

This function provides NFTL with information about the partitions that will be managed.

Syntax

```
t_nftl_error NFTL_Initialize(
    ubyte partitionsNumber,
    t_partitionInfo *partitionInfos)
```

Parameters

Parameter	Description
partitionsNumber	(in) The number of partitions that NFTL must initialize.
*partitionInfos	(in) A list of t_partitionInfo that contains the information for each partition.

Return values

t_nftl_error	Description
NFTL_INVALID_PARTITION	If returned, the parameters are incorrect.
NFTL_SUCCESS	Function terminates correctly.
NFTL_FAILURE	It is impossible to create global structure.

Comments

The number of elements in *partitionInfos are equal to partitionsNumber. The partitions, as defined in partitionInfos elements, cannot overlap.

NFTL_FormatPartition Function

This function formats a partition. This function can be used even if the partition is not mounted.

Syntax

```
t_nftl_error NFTL_FormatPartition (
    ubyte partition,
    uword FillFactor)
```

Parameters

Parameter	Description
partition	(in) The number of the partition to format.
FillFactor	(in) The fillFactor for this partition. For more information, refer to Programming Guide (page 31).

Return values

t_nftl_error	Description
NFTL_INVALID_PARTITION	If returned, the parameters are incorrect.
NFTL_SUCCESS	The function terminates correctly.

t_nftl_error	Description
NFTL_FAILURE	Generic error.

Comments

partition indicates an existing partition. FillFactor is less than partitionSize. If the formatted partition is mounted before the format, it will be mounted after the format. Otherwise, it will remain unmounted.

NFTL_MountPartition Function

This function mounts a formatted partition.

Syntax

```
t_nftl_error NFTL_MountPartition (
    ubyte partitionNumber)
```

Parameters

Parameter	Description
partitionNumber	(in) The number of the partition to mount.

Return values

t_nftl_error	Description
NFTL_INVALID_PARTITION	If returned, the parameters are incorrect.
NFTL_SUCCESS	The function terminated correctly.
NFTL_FAILURE	Generic error.
NFTL_UNFORMATTED	Unformatted partition.

Comments

partitionNumber indicates an existing partition. The partition must be formatted.

NFTL_UnmountPartition Function

This function unmounts a formatted partition.

Syntax

```
t_nftl_error NFTL_UnmountPartition (
    ubyte partition)
```

Parameters

Parameter	Description
partition	(in) The number of the partition to unmount.

Return values

t_nftl_error	Description
NFTL_INVALID_PARTITION	If returned, the parameters are incorrect.
NFTL_SUCCESS	The function terminated correctly.
NFTL_FAILURE	Generic error.

Comments

partition indicates an existing partition. The partition must be formatted.

NFTL_WriteSector Function

This function writes a sector.

Syntax

```
t_nftl_error NFTL_WriteSector (
    ubyte partition,
    uword VirtualAddress,
    ubyte *Buffer)
```

Parameters

Parameter	Description
partition	(in) The number of the partition where the data will be written.
VirtualAddress	(in) The logical address where the WRITE will be performed. It represents a logical sector number .
*Buffer	(in) Contains the data that will be written.

Return values

t_nftl_error	Description
NFTL_INVALID_ADDRESS	The address is incorrect.
NFTL_SUCCESS	The function terminated correctly.
NFTL_FAILURE	Generic error.
NFTL_INVALID_PARTITION	The partition number is invalid.
NFTL_UNMOUNTED_PARTITION	The partition is not mounted.
NFTL_VERIFY_ERROR	Verified error.
NFTL_DISK_ERROR	Disk error.

Comments

In a SWAP partition, all written data will be lost after a power loss or power down.

NFTL_WriteSectors Function

This function writes multiple, consecutive sectors.

Syntax

```
t_nftl_error NFTL_WriteSectors (
    ubyte partition,
    uword VirtualAddress,
    ubyte numOfSectors,
    ubyte *Buffer)
```

Parameters

Parameter	Description
partition	(in) The number of the partition where the data will be written.
VirtualAddress	(in) The first logical address where the WRITE will be performed. It represents a logical sector number .

Parameter	Description
numOfSectors	(in) The <u>number of sectors</u> to be written.
Buffer	(in) Contains the data that will be written.

Return values

t_nftl_error	Description
NFTL_INVALID_PARTITION	If returned, the parameters are incorrect.
NFTL_SUCCESS	The function terminated correctly.
NFTL_FAILURE	Generic error.

Comments

The behavior of this function changes according to the partition type:

- In SWAP partitions, all written data will be lost.
- In non-SWAP partitions, this function manages the Power Loss Recovery sector by sector. As a result, after a power loss during WriteSectors execution, some sectors will have been correctly updated, while others have been left as they were before the call.

NFTL_ReadSector Function

This function reads a sector.

Syntax

```
t_nftl_error NFTL_ReadSector (
    ubyte partition,
    udword VirtualAddress,
    ubyte *Buffer)
```

Parameters

Parameter	Description
partition	(in) The number of the partition to be read.
VirtualAddress	(in) The logical address where the READ will be performed. It represents the logical sector number.
*Buffer	(out) Will contain the data to be read.

Return values

t_nftl_error	Description
NFTL_INVALID_ADDRESS	The address is incorrect.
NFTL_SUCCESS	The function is terminated correctly.
NFTL_FAILURE	Generic error.
NFTL_INVALID_PARTITION	The partition number is invalid.
NFTL_UNMOUNTED_PARTITION	The partition is not mounted.

Comments

None.

NFTL_DeleteSector Function

This function deletes a sector.

Syntax

```
t_nftl_error NFTL_DeleteSector (
    ubyte partition,
    udword VirtualAddress)
```

Parameters

Parameter	Description
partition	(in) The number of the partition where the sector will be deleted.
VirtualAddress	(in) The logical address of the sector that will be deleted.

Return values

t_nftl_error	Description
NFTL_INVALID_ADDRESS	The address is incorrect.
NFTL_SUCCESS	The function terminated correctly.
NFTL_FAILURE	Generic error.
NFTL_INVALID_PARTITION	The partition number is invalid.
NFTL_UNMOUNTED_PARTITION	The partition is not mounted.
NFTL_VERIFY_ERROR	Verified error.
NFTL_DISK_ERROR	Disk error.

Comments

None.

NFTL_DefragPartition Function

This function frees invalid partition space.

Syntax

```
t_nftl_error NFTL_DefragPartition (
    ubyte partition)
```

Parameters

Parameter	Description
partition	(in) The number of the partition to be defragmented.

Return values

t_nftl_error	Description
NFTL_SUCCESS	The function terminated correctly.
NFTL_FAILURE	Generic error.
NFTL_INVALID_PARTITION	The partition number is invalid.
NFTL_UNMOUNTED_PARTITION	The partition is not mounted.
NFTL_VERIFY_ERROR	Verified error.

Comments

None.

NFTL_GetPartitionStatus Function

This function returns information about a partition.

Syntax

```
t_nftl_error NFTL_GetPartitionStatus (  
    ubyte      partition,  
    t_partitionStatus *partitionStatus)
```

Parameters

Parameter	Description
partition	(in) The number of the partition involved in this operation.
partitionStatus	(out) Contains the information about the partition.

Return values

t_nftl_error	Description
NFTL_SUCCESS	The function terminated correctly.
NFTL_FAILURE	Generic error.
NFTL_INVALID_PARTITION	The partition number is invalid.
NFTL_UNMOUNTED_PARTITION	The partition is not mounted.

Comments

None.

NFTL_GetPartitionsInfo Function

This function returns information about the number and type of partition.

Syntax

```
t_nftl_error NFTL_GetPartitionsInfo (  
    t_partitionInfo ** pinfo)
```

Parameters

Parameter	Description
** pinfo	(out) Will contain the partitionInfo structure for each defined partition.

Return values

t_nftl_error	Description
NFTL_SUCCESS	The function terminated correctly.
NFTL_FAILURE	Generic error.

Comments

None.

NFTL_IsPartitionMounted Function

This function returns the status of a partition (mounted/unmounted).

Syntax

```
t_nftl_error NFTL_IsPartitionMounted (
    ubyte partitionNumber)
```

Parameters

Parameter	Description
partitionNumber	(in) The number of the partition to check.

Return values

t_nftl_error	Description
NFTL_INVALID_PARTITION	The provided parameter is incorrect.
NFTL_MOUNTED_PARTITION	The function terminated correctly.
NFTL_UNMOUNTED_PARTITION	The partition is unmounted.

Comments

None.

NFTL_Delnit Function

This function unmounts all partitions and deallocates all data structures.

Syntax

```
t_nftl_error NFTL_Delnit ()
```

Parameters

None.

Return values

t_nftl_error	Description
NFTL_SUCCESS	The function terminated correctly.
NFTL_FAILURE	Generic error.

Comments

None.

NFTL_StartBackground Function

This function starts a background thread to collect garbage in a given partition.

Syntax

```
t_nftl_error NFTL_StartBackground (
    ubyte partitionNumber)
```

Parameters

Parameter	Description
partitionNumber	(in) A specific partition number.

Return values

t_nftl_error	Description
NFTL_INVALID_PARTITION	The given partition number is out of bounds.

t_nftl_error	Description
NFTL_UNMOUNTED_PARTITION	The given partition has not been mounted.
NFTL_SUCCESS	The background thread has been activated correctly.

Comments

PartitionNumber indicates an existing and mounted partition.

This function may be called repeatedly with no problems or side effects. In all cases, just one thread will be active per partition.

NFTL_StopBackground Function

This function starts a background thread to collect garbage in a given partition.

Syntax

```
t_nftl_error NFTL_StopBackground (
    ubyte    partitionNumber)
```

Parameters

Parameter	Description
partitionNumber	(in) A specific partition number.

Return values

t_nftl_error	Description
NFTL_INVALID_PARTITION	The given partition number is out of bounds.
NFTL_SUCCESS	The background thread has been deactivated correctly.

Comments

PartitionNumber indicates an existing partition.

It is safe to call this API when no background thread has been started or if it has already been stopped.

NFTL_IsBackgroundRunning Function

This function checks if a background thread is currently up and running over a given partition.

Syntax

```
t_nftl_error NFTL_IsBackgroundRunning (
    ubyte    partitionNumber)
```

Parameters

Parameter	Description
partitionNumber	(in) A specific partition number.

Return values

t_nftl_error	Description
BACKGROUND_OFF	The given partition number is out of bounds, the partition has not been mounted, or the background thread is not running.

t_nftl_error	Description
NFTLBACKGROUND_ON	The background thread for the given partition is up and running.

Comments

None.

Getting Started


The following sample code is located in the Common.h file. All #defines should be de-commented/commented to enable/disable the feature. For example, if you want to use the power loss feature, you must decomment the PL defines.

NFTL supports two compiling procedures. The first procedure consists of modifying the common.h defines as desired and then building the NFTL project. To use this procedure, you must not define the USE_EXTERNAL_MACRO_DEFINITIONS macro. The second procedure can be performed when using a build tool (for example: MAKE) to build NFTL, passing to it the macro definitions. In this case, it is required that you define the USE_EXTERNAL_MACRO_DEFINITIONS macro.

User Configuration

The following paragraphs describe each possible configuration of common.h. However, some configurations are not supported. A list of configuration constraints is defined in NFTL Allowed Configurations (page 26).

Unsupported Configurations for common.h

Device	Sector Size	POWER COMMIT	ERASE COMMIT	MULTI-PLANE	Supported	Description	Workaround
NAND 512 Byte Small Page	512 byte	ANY	ANY	NO	YES	-	-
NAND 512 Byte Small Page	512 byte	ANY	ANY	YES	NO	Not supported	-
NAND 512 Byte Small Page	1KB	ANY	ANY	ANY	NO	Not supported	-
NAND 512 Byte Small Page	2KB	ANY	ANY	ANY	NO	Not supported	-
NAND 512 Byte Small Page	4KB	ANY	ANY	ANY	NO	Not supported	-
NAND 2KB Large Page SLC	512 byte	ANY	ANY	NO	YES with workaround	Sector Sizes that are not equal to the Page Size are not supported.	# Use 2KB sector size. # wrapper the write protocol to work as follows: 1. Read.  2. Append/copy user's 512-byte sector. 3. Write 2KB sector.
NAND 2KB Large Page SLC	512 byte	ANY	ANY	YES	NO	Not supported	-
NAND 2KB Large Page SLC	1KB	ANY	ANY	ANY	NO	Not supported	-
NAND 2KB Large Page SLC	2KB	ANY	ANY	NO	YES	-	-
NAND 2KB Large Page SLC	2KB	ANY	ANY	YES	NO	Not supported	-
NAND 2KB Large Page SLC	4KB	ANY	ANY	NO	NO	Not supported	-
NAND 2KB Large Page SLC	4KB	ANY	ANY	YES	YES	-	-
OneNAND™ 2KB Large Page SLC	512 byte	ANY	ANY	YES	NO	Not supported	NO
OneNAND™ 2KB Large Page SLC	512 byte	ANY	ANY	NO	YES with workaround	Sector Sizes that are not equal to the Page Size are not supported.	# Use 2KB sector size. # wrapper the write protocol to work as follows: 1. Read. 2. Append/copy user's 512-byte sector. 3. Write 2KB sector.
OneNAND™ 2KB Large Page SLC	1KB	ANY	ANY	ANY	NO	Not supported	-
OneNAND™ 2KB Large Page SLC	2KB	ANY	ANY	NO	YES	-	-
OneNAND™ 2KB Large Page SLC	2KB	ANY	ANY	YES	NO	Not supported	-
OneNAND™ 2KB Large Page SLC	4KB	ANY	ANY	YES	YES	-	-
OneNAND™ 2KB Large Page SLC	4KB	ANY	ANY	NO	NO	Not supported	-

NFTL Allowed Configurations

NFTL does not support all possible combinations of devices and configuration macros. The constraints are the following:

- One and only one NAND.c and ONENNAND.c source code files must be linked together with the NFTL source code.
- On NAND small page devices, it is not allowed to enable either ERASE_COMMIT_ENABLED or POWER_COMMIT_ENABLED.
- The MULTIPLANE macro should be used only with NAND and OneNAND™ devices that support multi-plane features.

Special Features

```
#define MULTIPLANE // enable the use of multiplane features
```

```
#define REFRESH_1_BIT // enable the copy of a block when a bit error occurs
```

```
define SWAP_ERASE // enable the erase at the boot of all the written blocks in SWAP  
type partitions; if this features is not enabled, all written blocks will be invalidated
```

PL Features

```
#define POWER_COMMIT_ENABLED // Write Commit Operation
```

Other Customization

```
#define STATIC_WL 1 //Static wear leveling enabled
```

```
#define STATIC_WEAR_THRESHOLD 999 // Max age difference between younger and  
older block before static wear leveling
```

```
#define LOGGER_ENABLED //Comment this define to disable the Logging module
```

```
#define NFTL_DEBUG
```

```
#define ERASE_COMMIT_ENABLED // Enable the commit flag for the erase operation
```

```
#define SAFE_WRITE // Enable the safe write. That is, perform a read verify for each  
write operation
```

```
#define BLANK_CHECK_ENABLED 0X01 // This define must be enabled, and its value  
must be 0x01. It will be used in future releases
```

Porting Guide

The NFTL software handles all management functions, enabling operating systems to read and write to NAND Flash memory devices in the same way as disk drives. The software has been fully tested and debugged for use with a wide range of Micron NAND Flash memory devices, including SLC large/small page family devices.

Integrating NFTL into a System

The integration of NFTL into a new or existing system requires two activities:

1. Configure the OS Adaptation module.
2. Select the correct device driver and develop/customize the Platform Dependent Layer.

The next sections describe these activities in detail. In addition, Micron offers support and reference code for some common embedded operating systems.

Configuring the OS Adaptation Module

Configuring the OS Adaptation module includes correctly assigning values to the macros and typedefs in OS_AdaptationModule.h.

According to your specific needs, you may also need to implement/define further macros or functions and add references to external items (such as header files, functions, and variables) defined elsewhere in your own code.

The following table lists and describes the integration-related macros.

Table 2: Integration Macros

Macro	Description
OS_Task_Create(func, part-Number)	This macro creates a new background thread. It uses a generic interface due to the models. The first parameter, func, is the code that the thread will execute. It has a function with a generic void* parameter that returns nothing. partNumber is the parameter that func receives when started. This macro must return 0 if the task cannot be created. Otherwise, 1 is returned. If the multithreading features of NFTL are not needed, you can define OS_TASK_CREATE as 1.
OS_Task_Close(func)	This macro terminates an existing thread. Its parameter is the number of partitions associated to the thread to be terminated. The association between the partition and thread is implementation dependent. Note that many implementations do not need to explicitly terminate threads. If the multithreading features of NFTL are not required, you can define it as an empty macro.
OS_Mutex_Create(mutex)	This macro creates at compile time a mutex, which is a binary semaphore initialized at “unlocked” and associated with the name given as a parameter. According to your OS, you may need to implement this macro, OS_Mutex_Open, or both. If the multithreading features of NFTL are not required, you can define it as an empty macro.
OS_Mutex_Open(mutex)	This macro creates at run time a mutex, which is a binary semaphore initialized at “unlocked” and associated with the name given as a parameter. According to your OS, you may need to implement this macro, OS_Mutex_Create, or both. If the multithreading features of NFTL are not required, you can define it as an empty macro.
OS_Mutex_Destroy(mutex)	This macro frees the resources allocated to the given mutex named. If the multithreading features of NFTL are not required, you can define it as empty macro.

Table 2: Integration Macros (Continued)

Macro	Description
OS_Mutex_Lock(mutex)	This macro locks the given mutex. If it is already locked, the current process waits indefinitely. If the multithreading features of NFTL are not required, you can define it as an empty macro.
OS_Mutex_Unlock(mutex)	This macro unlocks a mutex locked by the current thread. If the multithreading features of NFTL are not required, you can define it as an empty macro.
OS_Deschedule(time)	This macro de-schedules the current process and ensures it will not be scheduled again before time msecs. If the multithreading features of NFTL are not required, you can define it as an empty macro.
OS_Malloc(x)	This macro is a generic version of the ANSI malloc function. It allocates x bytes on the heap and returns the pointer to them (or NULL if it fails).
OS_Free(x) Free(x)	This macro is a generic (and safer) version of the ANSI free function. It gets a pointer to an allocated area and performs two actions: it frees the area and sets the pointer to NULL.
OS_MemCmp(x, y, size)	This macro is a generic version of the ANSI memcmp function. It compares two buffers of a given size and returns "true" if their contents are the same.
OS_MemCpy(Dest, Src, size)	This macro is a generic version of the ANSI memcpy function. It copies the size bytes from Src to Dest.
OS_MemSet(Dest, Val, size)	This macro is a generic version of the ANSI memset function. It sets to Val the size in bytes pointed from Dest.

The following table lists the required typedefs.

Table 3: Required Typedefs

Typedef	Description
ubyte	Unsigned 8-bit-wide integer type.
byte	Signed 8-bit-wide integer type.
uword	Unsigned 16-bit-wide integer type.
word	Signed 16-bit-wide integer type.
udword	Unsigned 32-bit-wide integer type.
dword	Signed 32-bit-wide integer type.
MUTEX	The type used as a reference to a mutex. If the multithreading features of NFTL are not required, you can define it as any integer type.

Selecting the Correct Device Driver

This section describes how to select the correct device driver and develop/customize the Platform Dependent Layer.

The first step in this activity is to select the correct NAND and OneNAND™ drivers to link in your image, according to your platform. According to the selected driver, you must implement some functions that are dependent on the specific features of your board.

Setting Up the NAND Platform Dependent Driver

The following instructions describe how to set up the NAND platform dependent driver.

1. In the NAND.h file, set the NAND bus size. It can be defined once in either the X8 or X16 macro. If you forget to complete this step (or you define both), your compiler will indicate that this was not correctly defined. As a result, there is no risk in forgetting this step.
2. Define the low-level functions in the HD_NANDInterface.h file, which are described in the following table:

Low-level Function	Description
NAND_BusSize	This function returns the NAND bus size (16 or 8, according to your board design).
NAND_Open	This function begins a communication session with the device.
NAND_CommandInput	This function puts on the bus the given byte as a command.
NAND_AddressInput	This function puts on the bus the given byte as an address.
NAND_DataOutput	This function reads a single data unit from the Flash device.
NAND_MultipleDataInput	This function writes multiple bytes to the Flash device. Note that with x16 devices, the data buffer will NOT be guaranteed to be word aligned. If you can take advantage of DMA to improve I/O performance, you should add the related code here.
NAND_MultipleDataOutput	This function reads multiple bytes from the Flash device. Note that with x16 devices, the data buffer will NOT be guaranteed to be word aligned. If you can take advantage of DMA to improve I/O performance, you should add the related code here.
NAND_Close	This function ends a communication session with the device.
NAND_Platform_Init()	This function can be used to initialize the NAND controller, board register, and the like (for example: map the NAND port to the virtual memory space) before any operation is performed on the device. If it is not required by your system, you can leave it empty.

3. The last step is required only if your system takes advantage of hardware ECC. In this case, you must replace the ECC code included in NFTL with your own. In specific cases, this step may require further actions, according to your ECC algorithm. Contact Micron for further details and/or support for it. The ECC functions are implemented in the ECC.c\ECC.h files, and are described in the following table:

ECC Function	Description
GetECCSize	This function returns the size of the data unit for the ECC calculation (typ. 512).
MakeDataECC	This function calculates the ECC code for the given data and stores it in a buffer.
CheckDataECC	This function compares the ECC values stored in the NAND device with the ECC calculated on the data buffer and returns the new ECC values.
MakeSpareECC	This function calculates the ECC on a 16-byte buffer, representing a portion of the spare.
CheckSpareECC	This function compares the ECC values stored in the NAND device with the ECC calculated on the spare buffer and returns the new ECC values.

Setting Up the OneNAND™ Platform Dependent Driver

The following instructions describe how to configure the OneNAND™ platform driver.

1. Define the MEMORY_CONTROLLER_BASE_ADDRESS to the base address of the OneNAND™ register area.
2. Implement the three functions described in the following table:

Function	Description
Platform_Init	This function initializes the memory controller, board register, and the like.

Function	Description
PD_RAMBufferCopy	This function copies data to/from the ONENNAND internal buffer from/to RAM memory. The typical implementation will use either a DMA or the NFTL OS_MemCpy macro.
void PD_RAMBufferSet	This function fills the ONENNAND internal buffer with a given byte. The typical implementation will use the NFTL OS_Memset macro.

Maximum Number of Erasable Blocks (OneNAND™ Devices Only)

The following define is used to set the maximum number of erasable blocks for OneNAND™ devices:

```
#define MAX_NOOF_ERASABLE_BLOCKS
```

Set the maximum number of blocks that will be erased at a time using the MultiBlockErase feature of OneNAND™. The current value for MAX_NOOF_ERASABLE_BLOCKS is set to 64, which is the maximum number allowed by available OneNAND™ technology.

Note: It is recommended that you do not change this value.

Programming Guide

Normal Use of NFTL V4.6.0

After configuring the platform and correctly initializing the header file, it is possible to use NFTL as shown in the following example.

Step 1: Create the `t_partitionInfo` structure

```
typedef struct {  
    ubyte number;  
    ubyte safepartition;  
    uword startBlock;  
    uword sizeInBlocks;  
    uword numOfPageInBlock;  
    udword fillFactor;  
    ubyte blankCheckEnabled;  
    ubyte EnableBackground;  
    ubyte partition_type;  
    uword maxCacheEntries;  
} t_partitionInfo;
```

Where;

`t_partitionInfo.number`: The partition number in the whole NFTL.

`t_partitionInfo.safepartition`: Enable read verify after a write if the `SAFE_WRITE` macro is enabled.

`t_partitionInfo.startBlock`: The start blocks of this partition.

`t_partitionInfo.sizeInBlocks`: The total number of blocks in this partition.

`t_partitionInfo.numOfPageInBlock`: How many pages in one Physical Block. This information comes from the NAND Device Descriptions.

`t_partitionInfo.fillFactor`: The number of blocks used as hidden blocks.

`t_partitionInfo.blankCheckEnabled`: This value must be set to 0. Do not leave it undefined.

`t_partitionInfo.EnableBackground`: Set it to `Background_ON` if the background thread for this partition must be started as soon as the partition is mounted. Otherwise, set it to `BACKGROUND_OFF`.

`t_partitionInfo.partition_type`: The type of partition. Currently, NFTL supports two different types of partitions: Standard PL safe partitions (when `partition_type = NOOPTIONS`), and Swap partitions with no PLR or data preservation (when `partition_type = SWAP`).

`t_partitionInfo.maxCacheEntries`: The number of the block whose state must be cached in RAM. This field must be set between 1 and the number of physical blocks in the partition. Note that raising this field will increase the NFTL performance at the cost of using more RAM space. In the SWAP partition, this field is currently unused (note, this may

change in future NFTL releases) because this kind of partition always uses always a large cache as the partition.

Step 2: NFTL_Initialize

```
t_nftl_error NFTL_Initialize(ubyte partitionsNumber,t_partitionInfo * partitionInfos);
```

Step 3: NFTL_MountPartition

```
t_nftl_error NFTL_MountPartition(ubyte partitionNumber);  
t_nftl_error NFTL_FormatPartition(ubyte partition,udword FillFactor);
```

Step 4: Function operation

```
t_nftl_error NFTL_FormatPartition(ubyte partition,udword FillFactor);  
t_nftl_error NFTL_MountPartition(ubyte partitionNumber);  
t_nftl_error NFTL_UnmountPartition(ubyte partition);  
t_nftl_error NFTL_GetPartitionStatus(ubyte partition,t_partitionStatus *partitionStatus);  
t_nftl_error NFTL_GetPartitionsInfo(t_partitionInfo ** pInfo);  
t_nftl_error NFTL_IsPartitionMounted(ubyte aPartitionNumber);  
t_nftl_error NFTL_WriteSector(ubyte partition,udword VirtualAddress, ubyte *Buffer);  
t_nftl_error NFTL_WriteSectors (ubyte partition, udword VirtualAddress, ubyte numOfSectors,  
ubyte *Buffer);  
t_nftl_error NFTL_ReadSector(ubyte partition,udword VirtualAddress, ubyte *Buffer);  
t_nftl_error NFTL_DeleteSector(ubyte partition,udword VirtualAddress);  
t_nftl_error NFTL_DefragPartition(ubyte partition);  
t_nftl_error NFTL_StartBackground(ubyte aPartitionNumber);  
t_nftl_error NFTL_StopBackground(ubyte aPartitionNumber);  
t_nftl_error NFTL_IsBackgroundRunning(ubyte aPartitionNumber);
```

Step 5: NFTL_DeInit

After all operations are performed in NFTL, the NFTL_DeInit() function must be called.

Demo code for the five steps

```
#include <stdio.h>  
#include <string.h>  
#include "Common.h"  
#include "TranslationModule.h"  
  
#define N_BLOCKS 400  
#define PAGEINBLOCK 64 // 64: large page| 32: small page  
#define FillFactor 10  
  
void NFTL_Demo_Code(void)  
{  
    t_nftl_error ret;  
    ubyte partitionsNumber = 0;  
    ubyte wbuff[SECTOR_SIZE];  
    ubyte rbuff[SECTOR_SIZE];
```



```

uword * temp = NULL;
udword sectornum = 0;
udword totalsectors = 0;
udword i=0;

//First Step
ubyte uHIDE_BLOCK_NUMBER=10;
t_partitionInfo part_info[2];

//first partition
part_info[0].number = 0;
part_info[0].startBlock = 0;
part_info[0].sizeInBlocks = N_BLOCKS/2;
part_info[0].numOfPageInBlock = PAGEINBLOCK;
part_info[0].fillFactor = uHIDE_BLOCK_NUMBER;
part_info[0].safepartition =1; //safe partition enable
part_info[0].blankCheckEnabled =0;
part_info[0].EnableBackground = BACKGROUND_OFF;
part_info[0].partition_type = NOOPTIONS; /* PLR recovery*/
part_info[0].maxcacheentrines = N_BLOCKS/4; /* Half partition */
//second partition
part_info[1].number = 1;
part_info[1].startBlock = N_BLOCKS/2;
part_info[1].sizeInBlocks = N_BLOCKS/2;
part_info[1].numOfPageInBlock = PAGEINBLOCK;
part_info[1].fillFactor = uHIDE_BLOCK_NUMBER;
part_info[1].safepartition =0; //safe partition disable
part_info[1].blankCheckEnabled =0;
part_info[1].EnableBackground = BACKGROUND_OFF;
part_info[1].partition_type = SWAP; /* No data preservation between sessions, no PLR */
part_info[1].maxcacheentrines = 20; /* 20 physical blocks are cached */

printf("NFTL_Demo_Code");

//Second Step

ret = NFTL_Initialize(2,part_info);
if (ret != NFTL_SUCCESS)
{
    printf("NFTL_Initialize error!\n");
    return;
}

//Third Step
for(i = 0; i<2; i++)
{
    ret = NFTL_MountPartition(i);
    if (ret == NFTL_UNFORMATTED) //NAND VIRGIN
    {
        ret = NFTL_FormatPartition(i, uHIDE_BLOCK_NUMBER);
        if (ret == NFTL_SUCCESS)
        {
            ret = NFTL_MountPartition(i);
        } else
        {
            printf("NFTL_FormatPartition error!\n");
            return;
        }
    }
    if (ret != NFTL_SUCCESS)
    {
        printf("NFTL_Mount error!\n");
        return;
    }
}

```

```
//Forth Step
ret = NFTL_UnmountPartition(1);
if (ret != NFTL_SUCCESS)
{
    printf("NFTL_Unmount error!\n");
    return;
}
ret = NFTL_MountPartition(1);
if (ret != NFTL_SUCCESS)
{
    printf("NFTL_Mount error!\n");
    return;
}

ret = NFTL_FormatPartition(1, uHIDE_BLOCK_NUMBER);
if (ret != NFTL_SUCCESS)
{
    printf("NFTL_Format error!\n");
    return;
}

ret = NFTL_GetPartitionInfo(&part_info);
if (ret != NFTL_SUCCESS)
{
    printf("NFTL_GetPartitionInfo error!\n");
    return;
}
ret = NFTL_IsPartitionMounted(1);
if (ret == NFTL_UNMOUNTED_PARTITION)
{
    printf("NFTL_UNMOUNTED \n");
    return;
}

for(i=0;i<SECTOR_SIZE/sizeof(uword);i++)
    temp[i]=i;

ret = NFTL_WriteSector(1,0,wbuff);
if (ret != NFTL_SUCCESS)
{
    printf("NFTL_WriteSector error, SectorNum = 0!\n");
    return;
}

ret = NFTL_ReadSector(1,0,rbuff);
if (ret != NFTL_SUCCESS)
{
    printf("NFTL_ReadSector error, SectorNum = 0!\n");
    return;
}

ret = NFTL_DeleteSector(1,0);
if (ret != NFTL_SUCCESS)
{
    printf("NFTL_DeleteSector error, sectornum=0!\n");
    return;
}

ret = NFTL_IsBackgroundRunning(0);
if (ret == BACKGROUND_OFF)
{
    printf("Background is OFF\n");
} else {
```

```

    printf("Background is ON\n");
}

ret = NFTL_StartBackground(0);
if (ret != NFTL_SUCCESS)
{
    printf("NFTL_StartBackground error, partition = 0\n");
    return;
}

ret = NFTL_IsBackgroundRunning(0);
if (ret == BACKGROUND_OFF)
{
    printf("Error: Background has not been started correctly!!\n");
} else {
    printf("Background is ON\n");
}

ret = NFTL_StopBackground(0);
if (ret != NFTL_SUCCESS)
{
    printf("NFTL_StopBackground error, partition = 0\n");
    return;
}
if (ret != BACKGROUND_OFF)
{
    printf("Error: Background has not been stopped correctly!!\n");
} else {
    printf("Background is OFF\n");
}

//Fifth Step
ret = NFTL_DeInit();
if (ret != NFTL_SUCCESS)
{
    printf("NFTL_DeInit error!\n");
    return;
}
}

```

First Use of NFTL V4.6.0 on a Previously-used NAND Flash Memory

This section describes how to implement the current version of the NFTL software on a NAND Flash device that used software other than the current version of NFTL. In most cases another Flash software was used. In this case, the NAND Flash device must be erased before NFTL V4.6.0 can be implemented.

Note: Please ensure that the old data on this NAND device is no longer needed or is correctly saved on another device.

The following pseudo sample code is for the erasing function:

```

//startblock : the start block of this partition
//endblock : the end block of this partition
void ErasePartition (uword startblock, uword endblock)
{
    int I;
    ubyte ret;
    for ( I=startblock; I<=endblock; I++)
    {
        ret = IsBlockBAD(I);
        if (ret != BLOCK_BAD){
            ret = BlockErase(I);
            if (ret!= SUCCESS)
            {

```

```

        MarkBlockBAD(I);
    }
}
}
}

```

Then, perform the steps described in Normal Use of NFTL V4.6.0 (page 31).

Debug Mode Using NFTL V4.6.0

If the logging operation feature is enabled via the macro in the Common.h file, NFTL operations are monitored and any possible errors are tracked. The PrintLogger() API allows the user to know the module in which the error occurs and which internal operations have failed.

Demo code

```

#define N_BLOCKS 400
#define PAGEINBLOCK 64 // 64: large page| 32: small page
#define FillFactor 10

void NFTL_Demo_Code(void)
{
    t_nftl_error ret;
    ubyte partitionsNumber = 0;
    ubyte wbuff[SECTOR_SIZE];
    ubyte rbuff[SECTOR_SIZE];
    uword * temp = NULL;
    udword sectornum = 0;
    udword totalsectors = 0;
    udword i=0;

    //First Step
    ubyte uHIDE_BLOCK_NUMBER=10;
    t_partitionInfo part_info[1];
    part_info[0].number = 0;
    part_info[0].startBlock = 0;
    part_info[0].sizeInBlocks = N_BLOCKS/2;
    part_info[0].numOfPageInBlock = PAGEINBLOCK;
    part_info[0].fillFactor = uHIDE_BLOCK_NUMBER;
    part_info[0].safepartition = 0; //safe partition disable
    part_info[0].blankCheckEnabled = 0;
    part_info[0].partition_type = NOOPTIONS;
    part_info[0].maxcacheentrines = N_BLOCKS/4;
    part_info[0].EnableBackground = BACKGROUND_OFF;
    printf("NFTL_Demo_Code");

    //Second Step
    ret = NFTL_Initialize(1,part_info);
    if (ret != NFTL_SUCCESS)
    {
        printf("NFTL_Initialize error!\n");
        PrintLogger();
        return;
    }

    //Third Step
    ret = NFTL_MountPartition(0);
    if (ret != NFTL_SUCCESS)
    {
        printf("NFTL_Mount error!\n");
        PrintLogger();
        return;
    }
}

```

```
//Fourth Step
ret = NFTL_FormatPartition(0, uHIDE_BLOCK_NUMBER);
if (ret != NFTL_SUCCESS)
{
    printf("NFTL_Format error!\n");
    PrintLogger();
    return;
}

//Fifth Step
ret = NFTL_DeInit();
if (ret != NFTL_SUCCESS)
{
    printf("NFTL_DeInit error!\n");
    PrintLogger();
    return;
}
}
```

Definitions and Acronyms

Term	Definition
Age	The number of ERASE operations executed on a block.
Aging Block Table (ABT)	The set of ages of each physical block on a device.
Background Thread	A thread used to execute garbage collection while the system is idle. This is done to improve the user's perception of NFTL's performance.
Bad Block Management (BBM)	The HAL software module that manages bad blocks on the NAND device. The bad blocks can be either from the factory or generated during device use.
Bad Block Table (BBT)	The set of bad block physical numbers.
Error Correction Code (ECC)	The HAL module that calculates the error correction code (ECC) on READ/WRITE data.
Flash Abstraction Layer (FAL)	The layer that abstracts the NAND internal architecture. It consists of the following modules: <ul style="list-style-type: none"> • Translation Module (TM) • Wear Leveling (WL) • Garbage Collection (GC)
Free Blocks Chain (FBC)	The set of physical block numbers to which data has not been written.
Garbage Collection (GC)	The FAL software module that frees invalid memory space to allow further PROGRAM operations.
Hardware Adaptation Layer (HAL)	The layer that executes operations on the NAND Flash device (through the low level driver) and manages the errors due to hardware (through the Bad Block Management module and ECC).
Low Level Driver (LLD)	The HAL software module that executes the READ, WRITE, and ERASE operations directly on the NAND Flash memory device.
NAND Flash Translation Layer (NFTL)	The system that allows the file system to use the NAND Flash device as a mass storage device in place of a standard disk drive. It consists of two layers: <ul style="list-style-type: none"> • Flash Abstraction Layer (FAL) • Hardware Adaptation Layer (HAL)
Page Number	The offset of a physical page in a block.
Physical Address	The physical address calculated by the FAL module to execute sector operations directly on the NAND Flash device. It addresses both the physical block and physical page.
Physical Block	The smallest erasable area on the NAND Flash device. It can be erased in a single ERASE operation.
Physical Block Number	The physical block number in the partition chosen by the user in the user directive setting.
Physical Page	A portion of a physical block. There are two family devices supported by NFTL with page sizes of 512 bytes or 2048 bytes.
Sector	The minimal unit used for memory storage in READ and WRITE operations.
Translation Module (TM)	The FAL software module that exports the interface protocols to the user and converts virtual addresses to physical addresses.
Virtual Address	<u>The logical address that is used to locate sectors.</u> It refers to both the virtual block and sector.
Virtual Block	NFTL works with virtual blocks rather than with physical blocks.
Virtual Block Number	The block offset within the range of addressable blocks.



NAND Flash Translation Layer (NFTL) Definitions and Acronyms

Term	Definition
Virtual to Physical Mapping Table	The Translation module <u>data structure used to store information in RAM</u> to convert virtual addresses to physical addresses.
Wear Leveling (WL)	The FAL software module that monitors and spreads the number of PROGRAM/ERASE cycles across the entire NAND Flash memory device.

Source Code Organization

Table 4: Flash Abstraction Layer (FAL) Code File Names and Descriptions

Component	Code File Name	Description
Bad Block Management	BadBlockManagement.c	Manages bad blocks.
Bad Block Management	BadBlockManagemen.h	Exports the prototypes for the bad block management function.
Background Task	Background.h	Exports the function prototypes for the background functions.
Background Task	Background.c	Implements the garbage collection mechanism in the back-ground.
Cache Manager	CacheManager.h	Exports the function prototypes for the cache manager.
Cache Manager	CacheManager.c	Implements all algorithms related to the management of cache elements.
Garbage Collection	GarbageCollector.h	Manages the physical erasing of NAND Flash memory blocks.
Garbage Collection	GarbageCollector.c	Exports the prototypes for the garbage collection function.
Translation Module	TranslationModule.c	Manages all operations requested from the user through the APIs.
Translation Module	TranslationModule.h	Exports the API prototypes.
Translation Module	StructureManager.c	Manages, as an object, the data structure that allows the FAL to emulate rewriting.
Translation Module	StructureManager.h	Exports the function prototypes of object data structure.
Wear Leveling	WearLeveling.c	Implements the algorithms that propagate the PROGRAM/ERASE operations in the NAND Flash physical block.
Wear Leveling	WearLeveling.h	Exports the prototypes for the wear leveling function.

Table 5: Hardware Adaptation Layer (HAL) Code File Names and Descriptions

Component	Code File Name	Description
Error Correction Code	ECC.h	Interface to Software ECC functions.
Error Correction Code	ECC.c	Software ECC function implementation.
Low Level Driver	Common.h	A header file that contains the spare area configuration, return code and general definitions that are useful for each NFTL module.
Low Level Driver	DriverInterface.h	Generic interface to low-level driver functions.
NAND Driver	NAND.h	Interface to the NAND driver.
NAND Driver	NAND.c	NAND driver implementation.
OneNAND Driver	ONENAND.h	Interface to the OneNAND™ driver.
OneNAND Driver	ONENAND.c	OneNAND™ driver implementation.

Table 6: Operating System (OS) Specific Code File Names and Descriptions

Component	Code File Name	Description
Debug	NFTL_Debug.c	Provides user debug capability/features.

Table 6: Operating System (OS) Specific Code File Names and Descriptions (Continued)

Component	Code File Name	Description
Debug	NFTL_Debug.h	Exports the prototypes for the debug function.
Logger	NFTL_Logger.c	Provides user logging capability/features.
Logger	NFTL_Logger.h	Exports the prototypes for the logger function.
OS Adaptation Module	OS_Adaptation_Module.h	The header for the OS_Adaptation_Module.c. It exports the function's prototype, macros, and device settings.
OS Adaptation Module	OS_Adaptation_Module.c	Contains OS-specific implementation of functions related to dynamic memory management, thread, and mutex management. It also contains system-dependent data type definitions.

NFTL Frequently Asked Questions

What are the advantages of using NFTL?

NFTL is the best solution for systems that already have an operating system including a file system that manages data storage on other devices. In this case, NFTL can be seen simply as a driver to READ and WRITE sectors from/to the NAND Flash memory device.

The advantage of using an NFTL is having dedicated software to manage the NAND Flash behavior, which improves system performance and does not involve modifications of existing systems.

Does NFTL support large page (2112-byte) and small page (528-byte) NAND Flash memory devices?

NFTL supports both NAND and OneNAND™ Flash devices with page sizes equal to 528 bytes and 2112 bytes, depending on the devices supported.

How does NFTL manage bad blocks?

NFTL has a software module called Bad Block Management (BBM) that manages bad blocks. The bad blocks are marked with a Bad flag, which is created during the normal operation and is saved in the first physical page of each block. At each power-up, NFTL reads the bad block flag and skips the bad blocks.

Bad bLock information is also stored in a bad block table at the end of each partition.

How many bad blocks can be managed by NFTL?

For each partition, NFTL can manage a maximum of 2% of the partition size (in blocks) of bad blocks.

There are no limits to the number of bad blocks managed by NFTL.

What is the sector size managed by NFTL? Is it possible to change it?

NFTL performs READ and WRITE operations on 512/1024/2048/4096 byte sector sizes, depending on the underlying device and the compile time NFTL configuration. The sector size can be changed at compile time through the NFTL configuration files.

When does NFTL perform garbage collection?

- When requested by the user (through the NFTL interface API called “NFTL_DefragPartition”)
- When requested by the NFTL system. In this case, the garbage collection process starts when a user-defined threshold is reached.
- In the background from a specific thread if it has been started either implicitly at mounting or explicitly with a call to the NFTL_StartBackground() API.

How much RAM is required to perform NFTL data management correctly?

The amount of RAM necessary to perform NFTL data management on NAND Flash memory devices can range from about 30KB to 80KB, depending on enabled features, configuration and the quantity of data written in the NAND Flash device.

This RAM is used as shown in the following list:

- ~ 20KB are used to execute the NFTL code
- ~ 20KB are used to contain system static variables
- ~ 26KB to 65KB are used for system dynamic variables. The exact space used depends on the NFTL configuration. Note that increasing RAM usage has a positive affect on NFTL READ/WRITE performance.

If the NAND Flash device is unformatted, is it possible to manage data storage on it through NFTL?

It is not necessary to format the device before using NFTL. In fact, NFTL recognizes if the device can be used by finding the ABT table in NAND Flash. If this table does not exist, NFTL will create one. It is also possible to format a single partition at execution time using the dedicated API.

If the NAND Flash device is unformatted, is it possible to manage data storage on it through NFTL?

It is not necessary to format the device before using NFTL. In fact, NFTL recognizes if the device can be used by finding the ABT table in NAND Flash memory device. If this table does not exist, NFTL will create one. It is also possible to format a single partition at execution time using the dedicated API.

Does NFTL use error correction code (ECC)?

Yes. NFTL implements an ECC algorithm that enables NFTL to correct 1-bit ECC errors and to detect 2-bit ECC errors on 512 bytes of data.

Can custom ECC hardware be used with the NFTL software?

Yes. The NFTL software allows the user to disable its ECC algorithm and use a hardware ECC implemented by the host platform (if any).

Does NFTL support wear leveling?

Yes. NFTL implements a wear leveling algorithm that uniformly spans block ERASE operation on each block of Flash device. In addition, NFTL enables static wear leveling that guarantees the implementation of the wear leveling algorithm in the read-only area of a device.

Does NFTL support power loss recovery (PLR)?

Yes. The NFTL software is tested against sudden power loss.

What are the operations required to mount NFTL to a file system?

NFTL exports simple interfaces that perform sector operations on the NAND Flash device. As a result, it is only necessary to port NFTL interfaces into the file system software layer and adapt the NFTL data types and error returns.

How much NAND physical space is accessible by the user to manage data storage (through NFTL)?

- The blocks containing information on the number of PROGRAM/ERASE cycles performed on each block (the block age)
- The blocks dedicated to manage re-writing operations
- Bad blocks, up to 2% of the partition space

References

The following application notes and datasheets are available from either www.micron.com or your Micron distributor:

- Application Notes
 - AN1821, Garbage Collection in NAND Flash Memory
 - AN1819, Bad Block Management in NAND Flash Memory
 - AN1822, Wear Leveling in NAND Flash Memory
 - AN1823, Error Correction Code in Single-Level Cell NAND Flash Memory
- Datasheets
 - NAND128-A, NAND256-A, NAND512-A, NAND01G-A 528 Byte/264 Word Page
 - NAND01G-B, NAND02G-B, NAND04G-B, NAND08G-B 2112 Byte/1056 Word Page
 - NAND04G-C, 2112 Byte Page

Revision History

Rev. L, 02/11

- Updated software version number to 4.6.0.
- Revised the design architecture diagram.
- Added "Cache Management" and "Background Task" to the FAL description.
- Updated the Wear Leveling Module description.
- Added a section describing the Background Task module.
- Renamed the Low Level Driver (LLD) Module to "Platform Dependent Layer".
- Updated the Module Interface Information table.
- Revised list of NFTL features.
- Revised system requirements list.
- Added three new APIs to the list of exported NFTL functions.
- Revised the Enumerations, Constants, and Structures section.
- Added sections for the NFTL_StartBackground, NFTL_StopBackground, NFTL_IsBackgroundRunning and functions.
- Revised the Other Customization section.
- Revised the Porting Guide section.
- Revised the Programming Guide section.
- Updated the debug mode demo code.
- Added Background Thread to the Definitions and Acronyms table.
- Added new files to the Source Code Organization section.
- Updated the FAQs section.
- Updated the References section.
- Added a section describing the Cache Manager module.
- Added sections describing how to integrate NFTL into a system.

Rev. K, 08/10

- Updated the Unsupported Configuration for Common.h section.

Rev. J, 07/10

- Revised the design architecture diagram.
- Updated the description of the HAL.
- Updated the description of the LLD module.
- Added description of the Device dependent module.
- Updated the description of the ECC module.
- Added description of support for multiple partition types.
- Updated the code size in the System Requirements section.
- Updated the Interface Description section.
- Added NFTL_WriteSectors to the exported functions description.
- Updated the Enumerations, Constants, and Structures section.
- Added comments to description of the NFTL_WriteSector function.
- Added NFTL_WriteSectors function.

- Replaced the Unsupported Configurations for Common.h section with NFTL Allowed Configurations.
- Removed the Device Selection section.
- Removed the Sector Size Selection section.
- Removed the ECC Type Selection section.
- Added the SWAP_ERASE define to the Special Features section.
- Removed the MAX_ENTRY_CACHE define from the Other Customization section.
- Removed the ECC Enable section.
- Updated the steps and information in the Normal Use of NFTL section.
- Updated the first step in the Debug Mode Using NFTL section.
- Updated the answer to the question regarding sector sizes in the FAQs section.

Rev. 9, 03/10

- Removed the section First-used NAND.
- Removed references to MLC NAND.
- Updated the following sections:
 - Normal use of NFTL V4.x
 - First use of NFTL V4.x on a previously-used NAND flash memory
 - Debug mode using NFTL V4.x
 - Interface description

Rev. 8, 11/09

- Updated the following sections for NFTL version 4.4:
 - NFTL software features
 - NFTL exported functions
 - Enumerations, constants and structures
 - NFTL_Initialize function
 - NFTL_FormatPartition function
 - NFTL_UnmountPartition function
 - NFTL_WriteSector function
 - NFTL_ReadSector function
 - NFTL_DeleteSector function
 - NFTL_DefragPartition function
 - NFTL_GetPartitionStatus function
 - NFTL_GetPartitionsInfo function
 - NFTL_DeInit function
 - Special features
 - Other customization
 - First-used NAND
 - Normal use of NFTL V4.x
 - Debug mode using NFTL V4.x

Rev. 7, 07/09

- Updated the following sections:
 - System requirements
 - NFTL_Initialize function
 - Other customization
 - First-used NAND
 - Normal use of NFTL V4.x
 - Debug mode using NFTL V4.3

Rev. 6, 05/09

- Removed the "MultiBlockErase (OneNAND devices only)" section.
- Added paragraph to the "Getting started" and "User configuration" sections.
- Added the "Unsupported configurations for common.h." section.

Rev. 5, 04/09

- Updated formatting.
- Updated the following sections:
 - PL Features
 - Safe Write

Rev. 4, 04/09

- Added "Re-read on 2 bit ECC errors" to list of main features.
- Updated the list of defines in the "OTHER CUSTOMIZATION" section.
- Updated the introduction on the cover page.
- Revised the NFTL Design Architecture diagram to use Numonyx gradients.
- Edited document for spelling and grammar.
- Updated the list of Main Features.
- Added subsections to the Getting Started section.

Rev. 3, 12/08

- Change of NFTL API name.

Rev. 2, 10/08

- Deleted specific features that are end-of-life.
- Clarified the bad block information.
- Revised the "Source code organization" section.
- Updated the "NFTL frequently asked questions" section with the most current information.

Rev. 1, 03/07

- Initial release.

Disclaimer

Please Read Carefully:

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH MICRON PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN MICRON'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, MICRON ASSUMES NO LIABILITY WHATSOEVER, AND MICRON DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF MICRON PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

Micron products are not intended for use in medical, life saving, life sustaining, critical control or safety systems, or in nuclear facility applications.

Micron may make changes to specifications and product descriptions at any time, without notice.

Micron Technology, Inc. may have patents or pending patent applications, trademarks, copyrights, or other intellectual property rights that relate to the presented subject matter. The furnishing of documents and other materials and information does not provide any license, express or implied, by estoppel or otherwise, to any such patents, trademarks, copyrights, or other intellectual property rights.

Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined." Micron reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them.

Contact your local Micron sales office or your distributor to obtain the latest specifications and before placing your product order.

Copies of documents which have an order number and are referenced in this document, or other Micron literature may be obtained by visiting Micron's website at <http://www.micron.com>.

Other names and brands may be claimed as the property of others.

Copyright © 2011, Micron Technology, Inc., All Rights Reserved.

8000 S. Federal Way, P.O. Box 6, Boise, ID 83707-0006, Tel: 208-368-3900
www.micron.com/productsupport Customer Comment Line: 800-932-4992
Micron and the Micron logo are trademarks of Micron Technology, Inc.

All other trademarks are the property of their respective owners.

This data sheet contains minimum and maximum limits specified over the power supply and temperature range set forth herein. Although considered final, these specifications are subject to change, as further product development and data characterization sometimes occur.