

XSR v1.5.1

Porting Guide

APR-07-2006, Version 5.1





Copyright notice

Copyright Samsung Electronics Co., Ltd

All rights reserved.

Trademarks

XSR is a trademark of Samsung Electronics Co., Ltd in Korea and other countries.

Restrictions on Use and Transfer

All software and documents of **XSR** are commercial. Therefore, you must install, use, redistribute, modify and purchase only in accordance with the terms of the license agreement you entered into with Samsung Electronics Co., Ltd.

All advertising materials mentioning features or use of this software must display the following acknowledgement: Technical Writing Team.

"This product includes **XSR** written by Samsung Electronics Co., Ltd."

Contact Information

Flash Software Group
Memory Division
Samsung Electronics Co., Ltd

Address: BanWol-Dong, Hwasung-City
Gyeonggi-Do, Korea, 445-701

Telephone : +82-31-208-6370
Fax : +82-31-208-6799



Preface

SEC XSR POG-001

This Document is a Porting Guide for XSR (eXtended Sector Remapper) developed by Samsung Electronics.

Purpose

This document is XSR Porting Guide. This document explains the definition, architecture, system requirement, and porting tutorial of XSR. This document also provides the features and API of each module - OAM, PAM, LLD – that a user should know well to port XSR.

Scope

This document is for Project Manager, Project Leader, Application Programmers, etc.

Definitions and Acronyms

FTL (Flash Translation Layer)	A software module which maps between logical addresses and physical addresses when accessing to flash memory
XSR	eXtended Sector Remapper
STL	Sector Translation Layer
BML	Block Management Layer
LLD	Low Level Device Driver
Initial bad block	Invalid blocks upon arrival from the manufacturers
Run-time bad block	Additional invalid blocks may occur during the life of NAND flash usage
Sector	The file system performs read/write operations in a 512-byte unit called sector.
Page	NAND flash memory is partitioned into fixed-sized pages. A page is (512+16) bytes or (2048 + 64) bytes.
Block	NAND flash memory is partitioned into fixed-sized blocks. A block is 16K bytes or 128K bytes.
Wear-Leveling algorithm	Wear-leveling algorithm is an algorithm for increasing lifetime of NAND flash memory
NAND flash device	NAND flash device is a device that contains NAND flash memory or NAND flash controller.
NAND flash memory	NAND-type flash memory
Deferred Check Operation	The method that can increase time and device operation performance. Every operation function of LLD defers the check routine to the next operation.
OneNAND	Samsung NAND flash device that includes NAND flash memory and NAND flash controller.

Related Documents

- SEC, XSR v1.5.1 Part 1. Sector Translation Layer Programmer's Guide, Samsung Electronics, Co., LTD, APR-07-2006
- SEC, XSR v1.5.1 Part 2. Block Management Layer Programmer's Guide, Samsung Electronics, Co., LTD, APR-07-2006

History

Version	Date	Comment	Author	Approval
0.1	JULY-05-2004	Initial version	Seungeun Lee	
0.2	JULY-09-2004	Updated version - Delete FIL ; Some of FIL functionalities is shifted to BML or LLD. - Add a synchronous feature - Add chapter 7.5. "Byte Alignment Restriction" - Add MRead/MWrite function - Modify the figure of XSR system architecture. XSR consists of XSR Core (STL and BML), OAM, PAM, and LLD. In the previous version, XSR has only STL and BML	Seungeun Lee	
1.0	JULY-09-2004	Released version	Seungeun Lee	Kwangyoon Lee
1.1	OCT-18-2004	Updated version - Add MErase/ EraseVerify function - Add ECC Policy	Min Young Kim Se Wook Na Seungeun Lee	
2.0	OCT-19-2004	Released version	Seungeun Lee	Kwangyoon Lee
2.1	NOV-26-2004	Updated version - Add PAM interrupt functions - Modify OAM interrupt functions and timer functions - Modify Interrupt and Timer	Min Young Kim Se Wook Na Seungeun Lee	
3.0	DEC-09-2004	XSR v1.3.1 Release	Seungeun Lee	Kwangyoon Lee
3.1	JAN-14-2005	Modify ECC Policy	Min Young Kim Se Wook Na Seungeun Lee	
4.0	JUN-24-2005	Updated version for XSR v1.4.0 - Add PAM_Memcpy - Modify ECC policy - Modify interrupt and timer	Min Young Kim Se Wook Na	Song Ho Yoon
5.0 RC5	FEB-20-2006	XSR v1.5.0 RC5 Release	Byoung Young Ahn	
5.1	APR-07-2006	XSR v1.5.1 Release	Byoung Young Ahn	

Contents

1. Introduction	9
1.1. XSR Overview	9
1.2. XSR System Architecture	10
1.3. XSR Features	11
2. Development Environment	12
2.1. System Requirement	12
2.2. Directory Structure	12
3. Porting Tutorial	14
3.1. Prerequisite	14
3.1.1. Porting Outline	14
3.1.2. Condition Check	15
3.2. Porting Example	16
3.2.1. Extract XSR File	16
3.2.2. OAM Porting	16
3.2.3. LLD Porting	22
3.2.4. PAM Porting	34
3.2.5. Application Creating	37
4. OAM (OS Adaptation Module)	44
4.1. Description & Architecture	44
4.2. API	46
OAM_Malloc	47
OAM_Free	48
OAM_Memcpy	49
OAM_Memset	51
OAM_Memcmp	52
OAM_Debug	54
OAM_CreateSM	55
OAM_DestroySM	57
OAM_AcquireSM	59
OAM_ReleaseSM	61
OAM_InitInt	63
OAM_BindInt	65
OAM_EnableInt	67
OAM_DisableInt	69
OAM_ClearInt	71
OAM_ResetTimer	73
OAM_GetTime	75
OAM_WaitNMSec	77
OAM_Pa2Va	78
5. LLD (Low Level Driver)	82
5.1. Description & Architecture	82
5.2. API	84
XXX_Init	85
XXX_Open	87
XXX_Close	90
XXX_GetDevInfo	92

XXX_Read	97
XXX_Write	102
XXX_Erase	107
XXX_MRead	110
XXX_MWrite	118
XXX_MErase	126
XXX_EraseVerify	132
XXX_CopyBack	135
XXX_ChkInitBadBlk	143
XXX_SetRWArea	145
XXX_IOCTL	147
XXX_GetPrevOpData	153
XXX_FlushOp	156
6. PAM (Platform Adaptation Module)	158
6.1. Description & Architecture	158
6.2. API	160
PAM_Init	161
PAM_GetPAParm	162
PAM_RegLFT	166
PAM_InitInt	170
PAM_BindInt	172
PAM_EnableInt	174
PAM_DisableInt	176
PAM_ClearInt	178
PAM_Memcpy	180
7. Advanced Topics	182
7.1. Semaphore	182
7.1.1. Backgrounds	182
7.1.2. Semaphore	183
7.1.3. Implementation	184
7.2. Interrupt	188
7.2.1. Interrupt of Device Driver Concept	188
7.2.2. Synchronous / Asynchronous Device Driver Model	189
7.2.3. BML and Interrupt	192
7.2.4. STL and Interrupt	192
7.2.5. Timer	195
7.2.6. Implementation	197
7.3. Power-Off Recovery	200
7.3.1. Power-Off Error Processing and Timer	200
7.3.2. Implementation	203
7.4. Deferred Check Operation	204
7.5. Byte Alignment Restrictions	206
7.6. ECC Policy	209
Appendix	217
I. OAM.h	217
II. LLD.h	219
III. PAM.h	223
Index	225

Figures

Figure 1-1. XSR System Architecture	10
Figure 2-1. XSR and Platform Directory Structure	12
Figure 3-1. XSR Porting Flowchart	14
Figure 3-2. XSR and Platform Directory	15
Figure 3-3. XSR and Platform Directory	16
Figure 3-4. OAM Function Classification	17
Figure 3-5. Simple NAND Emulator Design	22
Figure 3-6. LLD Function Classification	23
Figure 3-7. PAM Function Classification	34
Figure 3-8. XSRHello Project Structure	41
Figure 3-9. XSRHello Project Working Screen	42
Figure 4-1. OAM in XSR System Architecture	44
Figure 5-1. LLD in XSR System Architecture	83
Figure 6-1. PAM in XSR System Architecture	159
Figure 6-2. Register LLD Address to BML	166
Figure 6-3. Define Volume and Device of BML	168
Figure 7-1. Simultaneous Requests to BML	182
Figure 7-2. Semaphore Operation in BML	183
Figure 7-3. Semaphore Usage Example	184
Figure 7-4. Device Driver Concept	188
Figure 7-5. Interrupt of Hardware Device	188
Figure 7-6. Polling	189
Figure 7-7. Synchronous Device Driver Operation	190
Figure 7-8. Interrupt Usage at Device Driver	191
Figure 7-9. Asynchronous Device Driver Operation	191
Figure 7-10. BML and Interrupt	192
Figure 7-11. STL to BML Mapping Example	193
Figure 7-12. Operation Queue Created by STL_AWrite()	194
Figure 7-13. STL_Sync() Operation (nQuantum: 1)	195
Figure 7-14. Flowchart of STL_Sync()	196
Figure 7-15. Flow between upper layer and XSR in asynchronous mode	199
Figure 7-16. Power-off Slope Assumption	201
Figure 7-17. NAND Block Erase Error at Power-off	202
Figure 7-18. OAM_WaitNMSec() Usage	203
Figure 7-19. Compare General Operation with Deferred Check Operation	204

Tables

Table 2-1. System Requirement	12
Table 2-2. Component of XSR and Platform Directory	12
Table 4-1. OAM API	46
Table 5-1. LLD API	84
Table 5-2. LLDSpec in LLD.h	93
Table 5-3. LLDMEArg data structure in LLD.h	127
Table 5-4. LLDMEList data structure in LLD.h	127
Table 5-5. RndInArg data structure in XsrTypes.h	136
Table 5-6. CpBkArg data structure in XsrTypes.h	136
Table 6-1. PAM API	160
Table 7-1. XSR Multiple volume/device supporting	183
Table 7-2. 4Byte Alignment Example	206
Table 7-3. Usage of Main Buffer and Spare Buffer in Read/Write Operation	210
Table 7-4. Return values of Main area ECC error	211
Table 7-5. Return values of Spare area ECC error	211

1. Introduction

This document is a Porting Guide of XSR (eXtended Sector Remapper). This chapter first introduces the definition, system architecture, and features of XSR.

1.1. XSR Overview

□ Flash Memory

Flash memory is an electronically programmable nonvolatile memory. It is used in laptop computer, PDA, or cell phone as storage. It is divided into two types, NOR and NAND, according to its cell composition at manufacturing. NAND flash memory has higher capacity and is cheaper than NOR flash memory. It has many benefits and widely used as storage of portable device.

□ NAND Flash Memory

Unlike a hard disk, a user cannot rewrite data on NAND flash memory. It means user cannot overwrite existing data on it without first executing an erase operation. A user needs to erase the current data to write new data. Additionally, erase operation is performed by a block, which is larger than other operation unit; read and write operation is performed by a page on NAND flash memory. This prolongs working time, because unrelated data must also be erased and then rewritten to complete updating.

NAND flash memory is also limited in the number of times it can be written to and then erased (erase cycles). Flash device writes data sporadically, not in its address order. Specific sector can be written more frequently and the entire flash device becomes unusable eventually. Lastly, NAND flash memory can have an initial bad block¹ when it is manufactured and a run time bad block or bit-flipping² when it is used.

□ XSR development background

As the above reason, a user cannot manage data on NAND flash memory like block devices³. Therefore, data management of NAND flash memory is very important issue. To cover that, Samsung Electronics develops the flash management software XSR (eXtended Sector Remapper) to use NAND flash memory as a regular block device. XSR has same functionalities with well-known FTL (Flash Translation Layer). Basically, it is a software layer or device driver, that resides between the OS file system and the NAND flash memory. XSR can be used separately and is independent of Operating system as a common component itself. It provides the OS with full block-device functionality so that NAND flash memory appears to the OS as a regular hard disk drive, while it transparently manages the flash data.

¹ Bad block is a block on which data cannot be read, written, or erased. There are two types of bad blocks: Initial bad block and Run-time bad block. Initial bad block occurs from manufacturer and runtime bad block occurs during using of NAND flash memory.

² Bad block is a block on which data cannot be read, written, or erased. When 1 bit error happens, XSR corrects the error itself, called bit-flipping.

³ Block devices include all disk drives and other mass-storage devices on the computer.

1.2. XSR System Architecture

As mentioned in chapter 1.1. XSR Overview, XSR exists between the file system and NAND flash memory. It works in conjunction with an existing Operating system or in some embedded applications.

Figure 1-1 shows the system architecture of XSR.

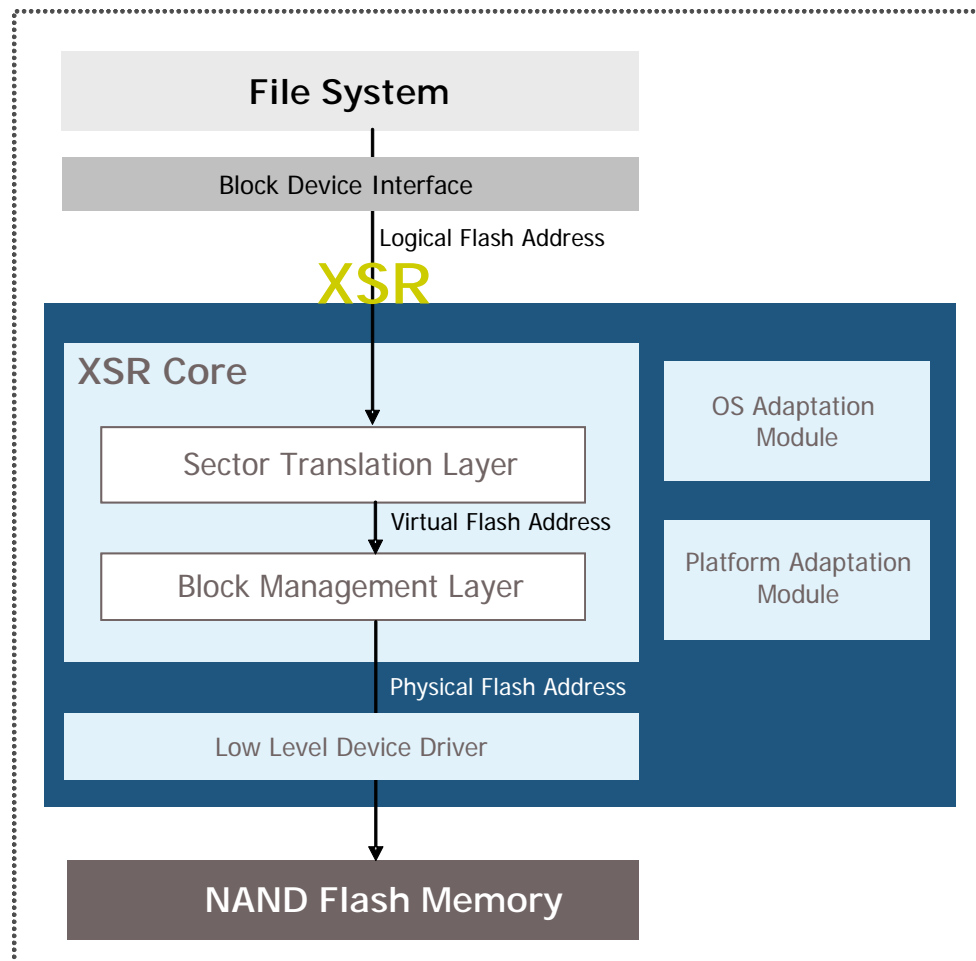


Figure 1-1. XSR System Architecture

There is a file system at the top of the figure. Block device interface exists between the file system and XSR. Block device interface is a kind of driver layer, which provides a file system with block device services. Block device interface code is ported for each OS.

There is XSR at the middle of the figure and it consists of four parts: XSR core, OAM (OS Adaptation Module), PAM (platform adaptation module), and LLD (Low Level Device Driver). The following briefly explains each of them.

□ **XSR core:** XSR core is composed of two layers: STL (Sector Translation Layer) and BML (Block Management Layer). STL is a top layer of XSR. BML is below the STL. The layers specifically have different features with each other, but all they are to perform the



basic functionalities of XSR as block device emulation and flash memory management. The main features of each layer are as follows.

- **STL** (Sector Translation Layer): translates a logical address from the file system into the virtual flash address. It internally has wear-leveling⁴ during the address translation.
- **BML** (Block Management Layer): translates the virtual address from the upper layer into the physical address. At this time, BML does the address translation in consideration of bad block and the number of NAND device in use. BML accesses LLD⁵, which actually performs read, write, or erase operation, with the physical address.

Note

Each layer of XSR can be operated separately as a module. Thus, STL can be used with other layer, which has same functionalities with BML.

☐ **OAM:** OAM is at the right of the figure. OAM connects XSR with the OS. OAM needs to be configured according to your OS environment to use NAND flash memory.

☐ **PAM:** PAM is below OAM. PAM connects XSR with the platform. PAM also needs to be configured according to your platform to use NAND flash memory.

☐ **LLD:** There is a low level device driver between BML and NAND flash memory. It reads, writes, or erases data on the physical sector address received from XSR and is controlled by BML.

1.3. XSR Features

The following describes the main benefits and features of XSR implementation.

- ☐ It emulates a block device and manages data on NAND flash memory efficiently.
- ☐ It extends the life span of NAND flash memory by enhancing Wear- leveling.
- ☐ It can be embedded in any kind of OS using NAND flash memory.
- ☐ It enhances data integrity by managing a bad block and performing error detection or correction.
- ☐ It reduces data loss in case of sudden power failure with the advanced algorithms, and guarantees data stability.

In the next chapter, XSR build and installation procedures are covered in detail.

⁴ Wear-leveling is an internal operation to use every block of NAND flash memory evenly through the algorithm. It extends NAND flash memory life span.

⁵ LLD is an abbreviation of Low Level Device Driver. It performs actual read/write/erase operation to NAND flash memory as a device driver.

2. Development Environment

This chapter describes the system requirement and the directory structure.

2.1. System Requirement

Table 2-1 shows the system requirement to install XSR and use it.

Table 2-1. System Requirement

System Requirement	
Host OS	Windows 2K/XP
Target CPU	50 mips
Source Disk Space	About 8 MB
NAND Flash Chip	Samsung NAND Chip Emulator using RAM
Target Disk Space	Minimum 50 MB

2.2. Directory Structure

Figure 2-1 shows the XSR and Platform Directory Structure. Depending on type of released package, detail structure of directories can be different from Figure 2-1.



Figure 2-1. XSR and Platform Directory Structure

Table 2-2 describes the components of XSR and Platform directory structure in Figure 2-1. The following table only refers to porting related components of XSR and Platform directory.

Table 2-2. Component of XSR and Platform Directory

Directory	Description
XSR	This folder is a base directory when XSR is installed.
Core	This folder has XSR source code. (STL and BML)

	Inc	This folder has XSR header files.
	Lib	This folder has XSR libraries (STL and BML).
	LLD	This folder has LLD source code.
	OAM	This folder has OAM source code.
Platform		This folder is a base directory for platform dependent code.
	PAM	This folder has platform source code.

3. Porting Tutorial

This chapter describes a porting example of XSR. First of all, you should read the prerequisite and check points. Then, follow the steps of the porting example.

3.1. Prerequisite

3.1.1. Porting Outline

The procedure of the porting example is as follows.

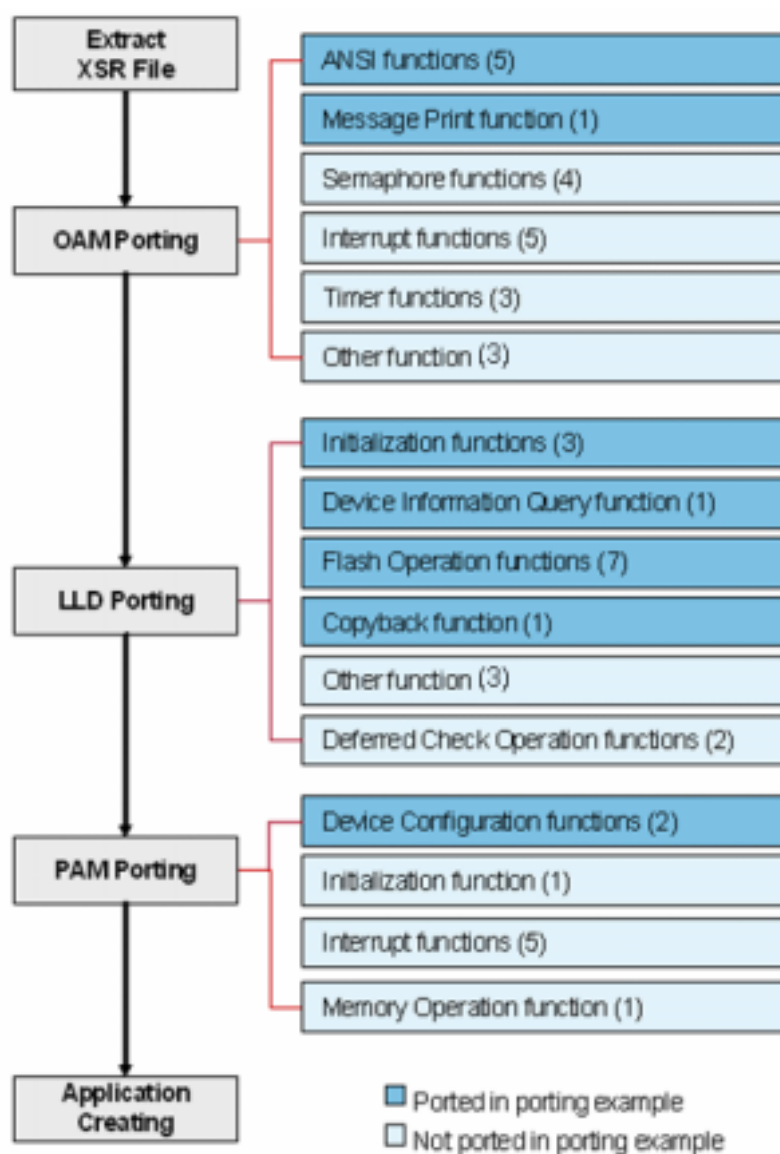


Figure 3-1. XSR Porting Flowchart

This document shows you the process to port XSR on a PC. So, you port OAM(OS Adaptation Module) to Win32. You implement ANSI functions, and Message print functions. You do not implement the semaphore functions, because you use the single process. Also, you do not implement both interrupt functions and timer functions, because you do not use real NAND flash memory.

In porting LLD, you do not use a real device or NAND flash memory. This document shows you the process to make a simple NAND emulator including small block NAND flash memory functionality. You do not implement functions related to Deferred Check Operation.

In porting PAM, you use the simple NAND emulator that is made in implementing LLD. You implement the device and driver configuration functions

After implementing OAM, LLD and PAM, you create the application in Visual Studio and check that XSR is normally operated. Therefore, XSR porting example will be completed.

3.1.2. Condition Check

Before you start the porting example, you should check the files to use in porting. The following figure shows XSR and Platform directory structure to check.

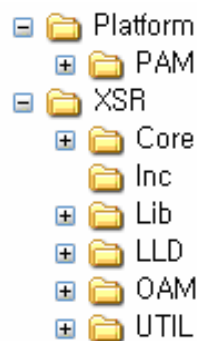


Figure 3-2. XSR and Platform Directory

The following description only refers to porting related files in XSR and Platform directory.

- ☐ In **PAM** folder, there is a template file in **Template** folder: MyPAM.cpp
- ☐ In **Inc** folder, there are XSR library files: BML.h, LLD.h, OAM.h, PAM.h, STL.h, XSR.h, and XsrTypes.h.
- ☐ In **Lib** folder, there is a XSR library file in **Generic\VS60\Retail** folder: XsrEmul.lib.
- ☐ In **LLD** folder, there are template files in **Template** folder: MyLLD.h and MyLLD.cpp.
- ☐ In **OAM** folder, there is a template file in **Template** folder: MyOAM.cpp.

Now, you fulfill all prerequisite for the porting example.

3.2. Porting Example

Follow the next porting example step by step.

3.2.1. Extract XSR File

At first, extract the provided XSR file.

- 1) Create a folder named as **XSR** in any location to port.
- 2) Extract **XSR_1.5.1_RTM.zip** or **XSR_1.5.1_xxx.zip** in a newly created folder **XSR**.

Then, you can see the directory structure as follows.

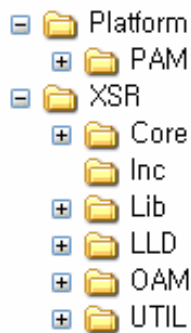


Figure 3-3. XSR and Platform Directory

3.2.2. OAM Porting

Among three modules (OAM, LLD, and PAM), you port OAM at first.

OAM, an OS-dependent module, links XSR to OS. In this porting example, you exercise OAM porting to Win32 because you port XSR on a PC.

In **\XSR\OAM** folder, there is a template file **MyOAM.cpp** in **Template** folder. This template file contains 21 functions that are classified into 6 categories as follows.

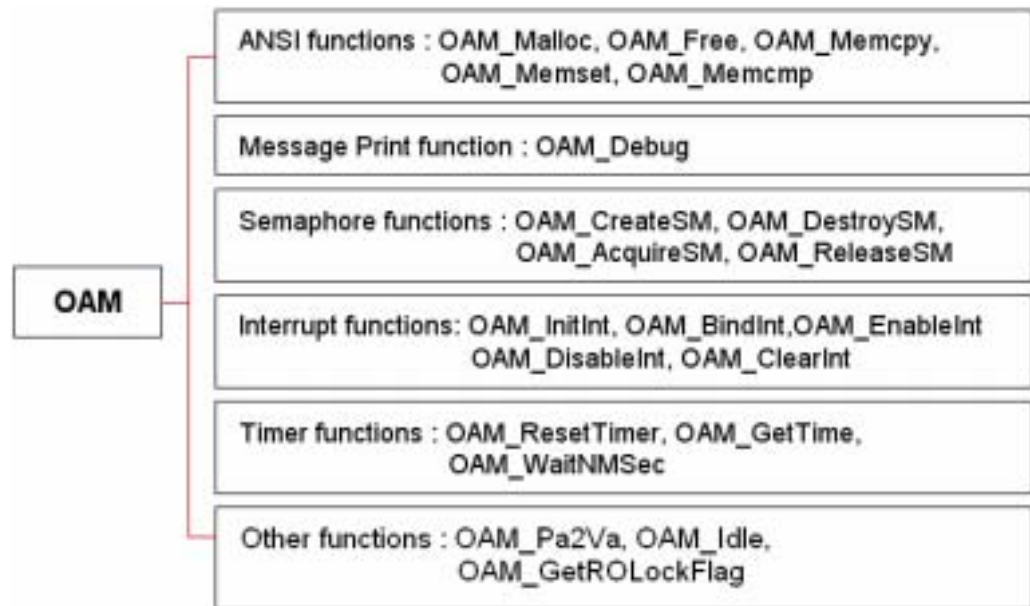


Figure 3-4. OAM Function Classification

The following explains the classification of OAM functions.

□ ANSI functions

ANSI functions are the mandatory functions that XSR library uses. You must implement these functions at all times.

```
OAM_Malloc()
OAM_Free()
OAM_Memcpy()
OAM_Memset()
OAM_Memcmp()
```

□ Message print function

Message print function is the functions to print all XSR messages, including an error or debug message. If you do not implement this function, you cannot see XSR debug messages.

```
OAM_Debug()
```

□ Semaphore functions

The semaphore functions are the functions to protect codes or devices using XSR in multi-task or multi-process environment. For more information about the semaphore functions, refer to Chapter 7.1. Semaphore.

```
OAM_CreateSM()
OAM_DestroySM()
OAM_AcquireSM()
OAM_ReleaseSM()
```

□ Interrupt functions

The interrupt functions are the functions for Asynchronous functionality of XSR. For more information about the interrupt, refer to Chapter 7.2. Interrupt.

```
OAM_InitInt()
OAM_BindInt()
OAM_EnableInt()
```

```
OAM_DisableInt()
OAM_ClearInt()
```

❑ Timer functions

Timer functions are the functions to check the time flow for Asynchronous functionality of XSR or handle the time-related errors. For more information about timer, refer to Chapter 7.2.5 Timer and 7.3 Power-Off Recovery

```
OAM_ResetTimer()
OAM_GetTime()
OAM_WaitNMSec()
```

❑ Other functions

These functions does not correspond to any category above.

```
OAM_Pa2Va()
OAM_Idle()
OAM_GetROLockFlag()
```

Now, port OAM to Win32.

- 1) Make a duplicate of the existing **Template** folder in \XSR\OAM, and name the new folder as **MyOAM**.

Check there is a file **MyOAM.cpp** in the newly named folder **MyOAM**.

- 2) Open the existing file MyOAM.cpp in an editor.
- 3) Add the followings to include the header files.

```
#include <windows.h>
#include <stdio.h>
#include <stdarg.h>
#include <string.h>
```

- 4) Add the following code in bold to implement ANSI functions.

```
VOID *
OAM_Malloc(UINT32 nSize)
{
    return malloc(nSize);
}

VOID
OAM_Free(VOID *pMem)
{
    free(pMem);
}

VOID
OAM_Memcpy(VOID *pDst, VOID *pSrc, UINT32 nLen)
{
    memcpy((void *) pDst, (void *) pSrc, nLen);
}
```

```

}

VOID
OAM_Memset(VOID *pDst, UINT8 nData, UINT32 nLen)
{
    memset((void *) pDst, (int) nData, nLen);
}

INT32
OAM_Memcmp(VOID *pCmp1, VOID *pCmp2, UINT32 nLen)
{
    return memcmp((void *) pCmp1, (void *) pCmp2, nLen);
}

```

ANSI functions are implemented to map one-to-one with the standard ANSI function. Each six functions execute the general memory operation: memory allocation, memory release, memory copy, memory set, and memory comparison. You must implement these mandatory functions.

- 5) Add the following code in bold to implement Message print function.

```

VOID
OAM_Debug(VOID *pFmt, ...)
{
    static char    aStr[4096];
    va_list        ap;

    va_start(ap, pFmt);
    vsprintf(aStr, (char *) pFmt, ap);
    printf(aStr);

    va_end(ap);
}

```

Message print function prints messages on XSR. If you want to see an error or debug message of XSR, you should implement this function.

- 6) Do not modify the semaphore functions because you use a single process, not a multi-process. The current template always returns TRUE32.

```

BOOL32
OAM_CreateSM(SM32 *pHandle)
{
    *pHandle = 1;
    return TRUE32;
}

BOOL32
OAM_DestroySM(SM32 nHandle)
{
    return TRUE32;
}

```



```
BOOL32
OAM_AcquireSM(SM32 nHandle)
{
    return TRUE32;
}

BOOL32
OAM_ReleaseSM(SM32 nHandle)
{
    return TRUE32;
}
```

For detailed information about the semaphore functions, refer to Chapter 4.2. OAM APIs.
For detailed information about the semaphore functionality, refer to Chapter 7.1. Semaphore.

- 7) Do not modify the interrupt functions because you do not use real NAND flash memory.

```
VOID
OAM_InitInt(UINT32 nLogIntId)
{
}

VOID
OAM_BindInt(UINT32 nLogIntId, UINT32 nPhyIntId)
{
}

VOID
OAM_EnableInt(UINT32 nLogIntId, UINT32 nPhyIntId)
{
}

VOID
OAM_DisableInt(UINT32 nLogIntId, UINT32 nPhyIntId)
{
}

VOID
OAM_ClearInt(UINT32 nLogIntId, UINT32 nPhyIntId)
{
}
```

For detailed information about the interrupt functions, refer to Chapter 4.2. OAM APIs.

- 8) Do not modify timer functions because you do not real NAND flash memory.

```
VOID
OAM_ResetTimer(VOID)
{
}

UINT32
```

```
OAM_GetTime(VOID)
{
    return 0;
}

VOID
OAM_WaitNMSec(UINT32 nNMSec)
{
}
```

For detailed information about timer functions, refer to Chapter 4.2. OAM APIs. For detailed information about timer functionality, refer to Chapter 7.2.5 Timer and 7.3 Power-Off Recovery.

- 9) The function OAM_Pa2Va is used for the address translation: from Physical address to Virtual address. Do not modify this function, because you do not use real NAND flash memory.

```
UINT32
OAM_Pa2Va(UINT32 nPAddr)
{
    return nPAddr;
}
```

If it is needed to the address translation for accessing to hardware, you should implement this function.

- 10) The function OAM_Idle is called when XSR is at idle time. For example, if XSR is polling on the device status, this function is called. Do not modify this function for now.

```
VOID
OAM_Idle(VOID)
{
}
```

At idle, other operations can be done by implementing this function.

- 11) The function OAM_GetROLockFlag is called when XSR determines whether certain block is in Read-Only partition or not. If XSR finds it is in range of RO partition, this function is called. If it is necessary to regard blocks in RO partition as RW under certain condition, implement this function. Do not modify this function unless you deeply understand internal implementation of XSR. Default implementation of this function just returns TRUE.

```
BOOL32
OAM_GetROLockFlag(VOID)
{
    return TRUE32;
}
```

After editing MyOAM.cpp, save the file and close it.

By far, you implement an OAM file, MyOAM.cpp, operating on Win32. Next, you implement LLD files.

3.2.3. LLD Porting

XSR sends a request to a device driver LLD, and then LLD accesses to the real device. This porting example shows you the process to make a simple NAND emulator as RAM, because you do not use real NAND flash memory.

Suppose that the simple NAND emulator is as follows;

- `pNANDArray`, a general pointer variable, points to the start address of the simple NAND emulator.
- Supposing that using large block, so 1 page = 4 sector.
- 1 page = (512 * 4) bytes main array + (16 * 4) bytes spare array .
- 1 block = 64 pages.
- The total number of blocks = 1024.
- The total memory of the simple NAND emulator
 = the number of total blocks * (the number of pages in a block * the size of a page)
 = 1024 * (64 * (512 * 4 + 16 * 4)) bytes.

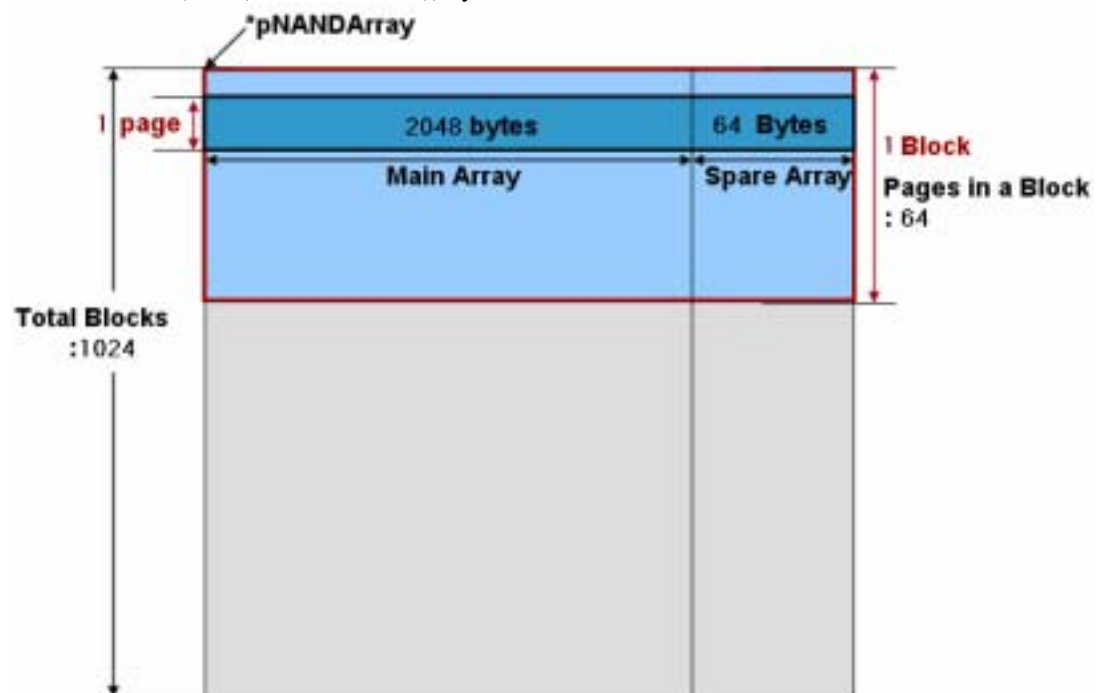


Figure 3-5. Simple NAND Emulator Design

In \XSR\LLD folder, there are two template files in **Template** folder: **MyLLD.h** and **MyLLD.cpp**. A template file **MyLLD.cpp** contains 17 functions that are classified into 6

categories.

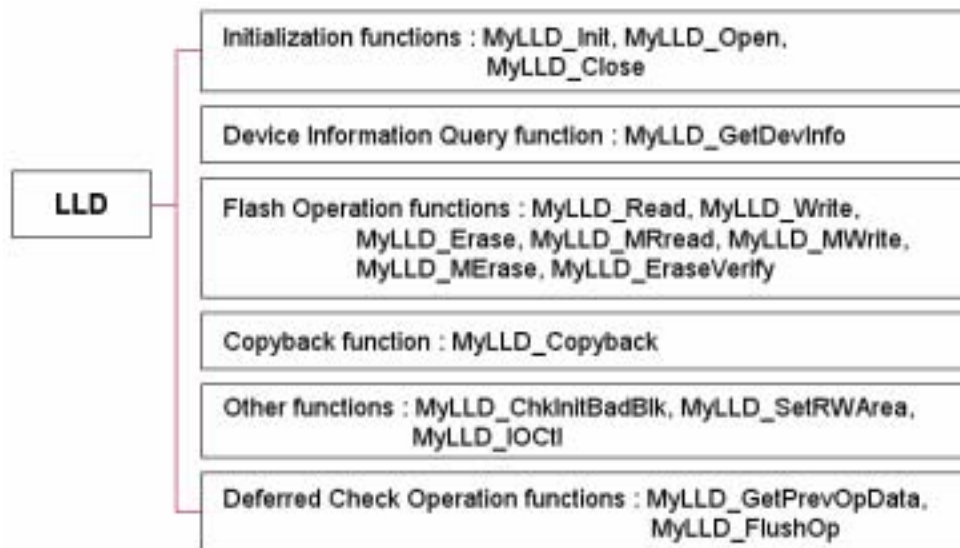


Figure 3-6. LLD Function Classification

Now, make a simple NAND emulator and port LLD to that simple NAND emulator.

- 1) Make a duplicate of the existing **Template** folder in \XSR\LLD, and name the new folder as **MyLLD**.

Check there are two files **MyLLD.cpp** and **MyLLD.h** in the newly named folder **MyLLD**.

- 2) Open the existing file MyLLD.h in an editor and check it.
Do not modify MyLLD.h and use it as it is.

```

#ifndef __MY_LLD_H__
#define __MY_LLD_H__

#ifdef __cplusplus
extern "C" {
#endif /* __cplusplus */

INT32 MyLLD_Init (VOID *pParm);
INT32 MyLLD_Open (UINT32 nDev);
INT32 MyLLD_Close (UINT32 nDev);
INT32 MyLLD_ChkInitBadBlk (UINT32 nDev, UINT32 nPbn);
INT32 MyLLD_Read (UINT32 nDev, UINT32 nPsn, UINT32 nNumOfScts,
                  UINT8 *pMbuf, UINT8 *pSbuf, UINT32 nFlag);
INT32 MyLLD_Write (UINT32 nDev, UINT32 nPsn, UINT32 nNumOfScts,
                  UINT8 *pMbuf, UINT8 *pSbuf, UINT32 nFlag);
INT32 MyLLD_Erase (UINT32 nDev, UINT32 nPbn, UINT32 nFlag);
INT32 MyLLD_CopyBack (UINT32 nDev, CpBkArg *pstCpArg,
                     UINT32 nFlag);
INT32 MyLLD_GetDevInfo (UINT32 nDev, LLDSpec* pstLLDDev);
INT32 MyLLD_GetPrevOpData (UINT32 nDev, UINT8 *pMbuf,

```

```

                                UINT8 *pSBuf);
INT32 MyLLD_FlushOp (UINT32 nDev);
INT32 MyLLD_SetRWArea (UINT32 nDev,  UINT32 n1stUB,
                        UINT32 nNumOfUBs);
INT32 MyLLD_IOCTL (UINT32 nDev,  UINT32 nCode,  UINT8 *pBufI,
                   UINT32 nLenI,  UINT8 *pBufO,  UINT32 nLenO,
                   UINT32 *pByteRet);
INT32 MyLLD_MRead (UINT32 nDev,  UINT32 nPsn,  UINT32 nNumOfScts,
                  SGL *pstSGL,  UINT8 *pSBuf,  UINT32 nFlag);
INT32 MyLLD_MWrite (UINT32 nDev,  UINT32 nPsn,  UINT32 nNumOfScts,
                   SGL *pstSGL,  UINT8 *pSBuf,   UINT32 nFlag,
                   UINT32 *pErrPsn);
INT32 MyLLD_MErase (UINT32 nDev,  LLDMEArg *pstMEArg,
                   UINT32 nFlag);
INT32 MyLLD_EraseVerify (UINT32 nDev,  LLDMEArg *pstMEArg,
                        UINT32 nFlag);

#ifdef __cplusplus
}
#endif /* __cplusplus */

#endif /* __MY_LLD_H__ */

```

If necessary, you can rename the prefix (MyLLD_) of LLD function suitable for the device.
If you change the function name, save the file MyLLD.h and close it.

- 3) Open the existing file MyLLD.cpp in an editor.
- 4) Add the following code to make a simple NAND emulator.
First, define the number of blocks and the size of total blocks to use. Then, define a general pointer variable pointing to the start address of the buffer for the simple NAND emulator.

```

#define NUM_BLK          1024
#define PGS_PER_BLK     64
#define SCTS_PER_PG      4
#define SCTS_PER_BLK    (64 * 4)
#define PG_SIZE          (512 * 4 + 16 * 4)
#define BLK_SIZE         (64 * PG_SIZE)

UINT8 *pNANDArray;

```

NUM_BLK is the number of the total blocks. It is defined as 1024.

BLK_SIZE is the size of a block. The formula to decide the size of a block is the number of pages in a block * the size of a page = 64 * (512 * 4 + 16 * 4).

pNANDArray is a general pointer variable pointing to the start address of the simple NAND emulator.

- 5) Add the following code in bold to implement Device initialization related functions: Init , Open and Close.


```

INT32
MyLLD_Init(VOID *pParm)
{
    XsrVolParm *pstParm = (XsrVolParm*)pParm;

    pNANDArray = (UINT8 *)OAM_Malloc(NUM_BLK * BLK_SIZE);
    if (pNANDArray == NULL)
    {
        LLD_RTL_PRINT((TEXT("Memory Allocation Error!!\n")));
        return LLD_INIT_FAILURE;
    }
    OAM_Memset(pNANDArray, 0xff, NUM_BLK * BLK_SIZE);

    return LLD_SUCCESS;
}

INT32
MyLLD_Open(UINT32 nDev)
{
    return LLD_SUCCESS;
}

INT32
MyLLD_Close(UINT32 nDev)
{
    return LLD_SUCCESS;
}

```

Init initializes data structure required to the device operating. Init called first and once when the driver is loaded.

This example allocates a buffer for the simple NAND emulator calling a function OAM_Malloc, and initializes the buffer as 0xff. In general, you can implement this function suitable for real NAND device.

Open initializes the device to be used.

Do not modify the template. In general, you can implement this function that makes the real device be ready to use.

Close, in opposite to **Open**, unlinks XSR and the device.

Do not modify the template. In general, you can implement this function that makes the real device be released.

- 6) Add the following code in bold to implement Device information query function GetDevInfo.

```

INT32
MyLLD_GetDevInfo(UINT32 nDev, LLDSpec* pstDevInfo)
{
    pstDevInfo->nMCode          = 0xEC;
    pstDevInfo->nDCode          = 0x11;

    pstDevInfo->nNumOfBlks      = NUM_BLK;
    pstDevInfo->nPgsPerBlk     = 64;
    pstDevInfo->nSctsPerPg     = 4;
}

```

```

pstDevInfo->nBlksInRsv      = 20;
pstDevInfo->nNumOfPlane     = 1;

pstDevInfo->nBadPos         = 0;
pstDevInfo->nLsnPos        = 2;
pstDevInfo->nEccPos         = 8;
pstDevInfo->nMEFlag         = LLD_ME_NO;
pstDevInfo->nBWidth         = 0;

pstDevInfo->aUID[ 0]        = 0;
pstDevInfo->aUID[ 1]        = 0;
pstDevInfo->aUID[ 2]        = 0;
pstDevInfo->aUID[ 3]        = 0;
pstDevInfo->aUID[ 4]        = 0;
pstDevInfo->aUID[ 5]        = 0;
pstDevInfo->aUID[ 6]        = 0;
pstDevInfo->aUID[ 7]        = 0;
pstDevInfo->aUID[ 8]        = 0;
pstDevInfo->aUID[ 9]        = 0;
pstDevInfo->aUID[10]        = 0;
pstDevInfo->aUID[11]        = 0;
pstDevInfo->aUID[12]        = 0;
pstDevInfo->aUID[13]        = 0;
pstDevInfo->aUID[14]        = 0;
pstDevInfo->aUID[15]        = 0;

pstDevInfo->nTrTime         = 500000;
pstDevInfo->nTwTime         = 3500000;
pstDevInfo->nTeTime         = 20000000;
pstDevInfo->nTfTime         = 50;

return LLD_SUCCESS;
}

```

GetDevInfo returns the device information to BML.

☐ **nMCode** and **nDCode** are Device IDs.

This example gives random code as 0xEC and 0x11 because you do not use a real device.

☐ **nNumOfBlks** is the number of blocks, **nPgsPerBlk** is the number of pages in a block,

☐ **nSctsPerPg** is the number of sectors in a page, and **nBlksInRsv** is the number of the reserved blocks.

☐ **nMEFlag** is a flag which indicates whether multi-block erase is supported or not.

☐ **aUID** is Unique ID of a device.

This example gives random code as 0 because you do not use a real device.

☐ **nTrTime** is the time for page reading, **nTwTime** is the time for page writing, **nTeTime** is the time for block erasing, **nTfTime** is the time for data transmitting (512 bytes). These values are referred as XSR mapping.

These time values are the standard to calculate the performance cost for the internal algorithm. The time values should be suitable for the real operation time approximately, although it is not exactly same. The unit of the operation time is ns.



For more information about the device information to BML, refer to LLD API, XXX_GetDevInfo.

- 7) Add the following code in bold to implement Flash operation functions: Read, Write, Erase, MRead, MWrite, MErase and EraseVerify.

```
INT32
MyLLD_Read(UINT32 nDev,  UINT32 nPsn,  UINT32 nNumOfScts,
            UINT8 *pMBuf,  UINT8 *pSBuf,  UINT32 nFlag)
{
    UINT32 nBlkOff;
    UINT32 nPgOff;
    UINT32 nSctOff;
    UINT32 nPos;

    nBlkOff = nPsn / SCTS_PER_BLK;
    nPgOff = (nPsn - nBlkOff * SCTS_PER_BLK) / SCTS_PER_PG;
    nSctOff = nPsn - (nBlkOff * SCTS_PER_BLK + nPgOff * SCTS_PER_PG);

    nPos = nBlkOff * BLK_SIZE + nPgOff * PG_SIZE;

    if (pMBuf != NULL)
    {
        PAM_Memcpy(pMBuf, pNANDArray + nPos + 512 * nSctOff,
                   512 * nNumOfScts);
    }

    if (pSBuf != NULL)
    {
        PAM_Memcpy(pSBuf, pNANDArray + nPos + 512 * 4 + 16 * nSctOff,
                   16 * nNumOfScts);
    }

    return LLD_SUCCESS;
}
```

Read is to read data as a unit of a sector from NAND flash memory.

The parameters `nPsn` and `nNumOfScts` read data as a unit of a sector. The number of sectors does not exceed the sector-included page because you can read/write as a unit of a page. That is, this function reads data as a unit of a page; else the argument of the function is received as a unit of a sector. So, the number of sectors is limited by the number of a page.

This example calculates the start address of memory using `pNANDArray`, and copies data to the using buffer. This function handles the main array and spare array separately when reading data of NAND flash memory page.

```
INT32
MyLLD_Write(UINT32 nDev,  UINT32 nPsn,  UINT32 nNumOfScts,
             UINT8 *pMBuf,  UINT8 *pSBuf,  UINT32 nFlag)
{
    UINT32 nIdx;
```

```

UINT32 nSctIdx;

UINT32 nBlkOff;
UINT32 nPgOff;
UINT32 nSctOff;
UINT32 nPos;

nBlkOff = nPsn / SCTS_PER_BLK;
nPgOff = (nPsn - nBlkOff * SCTS_PER_BLK) / SCTS_PER_PG;
nSctOff = nPsn - (nBlkOff * SCTS_PER_BLK + nPgOff * SCTS_PER_PG);

nPos = nBlkOff * BLK_SIZE + nPgOff * PG_SIZE;

if (pMBuf != NULL)
{
    for (nSctIdx = 0; nSctIdx < nNumOfScts ; nSctIdx++)
    {
        for (nIdx = 0; nIdx < 512; nIdx++)
        {
            pNANDArray[nPos + 512 * (nSctOff + nSctIdx) + nIdx]
                &= pMBuf[512 * nSctIdx + nIdx];
        }
    }
}

if (pSBuf != NULL)
{
    for (nSctIdx = 0; nSctIdx < nNumOfScts ; nSctIdx++)
    {
        for (nIdx = 0; nIdx < 16; nIdx++)
        {
            pNANDArray[nPos + 512 * 4 + 16 * (nSctOff + nSctIdx)
                + nIdx] &= pSBuf[16 * nSctIdx + nIdx];
        }
    }
}

return LLD_SUCCESS;
}

```

Write is to write data as a unit of a page or sector to NAND flash memory.

The parameters `nPsn` and `nNumOfScts` read data as a unit of a sector. The number of sectors does not exceed the sector-included page because you can read/write as a unit of a page. That is, this function writes data as a unit of a page, else the argument of the function is received as a unit of a sector. So, the number of sectors is limited by the number of a page.

This example calculates the start address of memory using `pNANDArray`, and writes data. This function always handles main array and spare array separately when it reads data of NAND flash memory page. You use an AND operator to emulate the real writing to NAND flash memory in this function.

```

INT32
MyLLD_Erase(UINT32 nDev, UINT32 nPbn, UINT32 nFlag)
{

```

```

OAM_Memset(pNANDArray + nPbn * BLK_SIZE, 0xff, BLK_SIZE);

return LLD_SUCCESS;
}

```

Erase is to erase a block of NAND flash memory.

This example calculates the start address of memory using pNANDArray, and erases data. This example handles as a unit of a block. You fill memory buffer with 0xff. In general, you can implement this function that accesses to the real NAND device.

```

INT32
MyLLD_MRead(UINT32 nDev,  UINT32 nPsn,  UINT32 nNumOfScts,
             SGL *pstSGL,  UINT8 *pSBuf,  UINT32 nFlag)
{
    UINT32 nBlkOff;
    UINT32 nPgOff;
    UINT32 nSctOff;
    UINT32 nPos;
    UINT32 nCnt;
    UINT32 nReadScts;
    UINT32 nSctIdx;
    UINT32 nCurPos;

    nBlkOff = nPsn / SCTS_PER_BLK;
    nPgOff = (nPsn - nBlkOff * SCTS_PER_BLK) / SCTS_PER_PG;
    nSctOff = nPsn - (nBlkOff * SCTS_PER_BLK + nPgOff * SCTS_PER_PG);

    nPos = nBlkOff * BLK_SIZE + nPgOff * PG_SIZE;
    nCurPos = nPos;
    nSctIdx = nSctOff;

    if (pstSGL != NULL)
    {
        for (nCnt = 0; nCnt < pstSGL->nElements; nCnt++)
        {
            nReadScts = 0;
            while (nReadScts < pstSGL->stSGLEntry[nCnt].nSectors)
            {
                PAM_Memcpy(
                    pstSGL->stSGLEntry[nCnt].pBuf + 512*nReadScts,
                    pNANDArray + nCurPos + 512 * nSctIdx, 512);

                nReadScts++;
                nSctIdx++;
                if (nSctIdx == 4)
                {
                    nCurPos += PG_SIZE;
                    nSctIdx = 0;
                }
            }
        }
    }

    if (pSBuf != NULL)

```

```

{
    PAM_Memcpy(pSBuf, pNANDArray + nPos + 512 * 4 + 16 * nSctOff,
               16 * nNumOfScts);
}

return LLD_SUCCESS;
}

```

MRead reads pages and sectors within a block boundary, while **Read** reads data within a page. This function can be optimized by not reading sectors one by one. But, for simplicity, the above scheme shows how to handle SGL argument.

```

INT32
MyLLD_MWrite(UINT32 nDev,  UINT32 nPsn,  UINT32 nNumOfScts,
             SGL *pstSGL,  UINT8 *pSBuf,  UINT32 nFlag,
             UINT32 *pErrPsn)
{
    UINT32 nIdx;
    UINT32 nBlkOff;
    UINT32 nPgOff;
    UINT32 nSctOff;
    UINT32 nPos;
    UINT32 nCnt;
    UINT32 nWriteScts;

    UINT32 nSctIdx;
    UINT32 nCurPos;
    UINT8 *pSrcBuf;

    nBlkOff = nPsn / SCTS_PER_BLK;
    nPgOff = (nPsn - nBlkOff * SCTS_PER_BLK) / SCTS_PER_PG;
    nSctOff = nPsn - (nBlkOff * SCTS_PER_BLK + nPgOff * SCTS_PER_PG);

    nPos = nBlkOff * BLK_SIZE + nPgOff * PG_SIZE;

    nCurPos = nPos;
    nSctIdx = nSctOff;

    if (pstSGL != NULL)
    {
        for (nCnt = 0; nCnt < pstSGL->nElements; nCnt++)
        {
            nWriteScts = 0;
            pSrcBuf = pstSGL->stSGLEntry[nCnt].pBuf;
            while(nWriteScts < pstSGL->stSGLEntry[nCnt].nSectors)
            {
                for (nIdx = 0; nIdx < 512; nIdx++)
                {
                    pNANDArray[nCurPos + 512*nSctIdx + nIdx]
                        &= pSrcBuf[512*nWriteScts + nIdx];
                }

                nWriteScts++;
                nSctIdx++;
                if (nSctIdx == 4)

```

```

        {
            nCurPos += PG_SIZE;
            nSctIdx = 0;
        }
    }
}

if (pSBuf != NULL)
{
    for (nSctIdx = 0; nSctIdx < nNumOfScts ; nSctIdx++)
    {
        for (nIdx = 0; nIdx < 16; nIdx++)
        {
            pNANDArray[nPos+512*4+16*(nSctOff+nSctIdx)+nIdx]
                &= pSBuf[16 * nSctIdx + nIdx];
        }
    }
}

return LLD_SUCCESS;
}

```

MWrite writes pages and sectors within a block boundary, while **Write** writes data within a page. This function can be optimized by not writing sectors one by one. But, for simplicity, the above scheme shows how to handle SGL argument.

```

INT32
MyLLD_MErase(UINT32 nDev, LLDMEArg *pstMEArg, UINT32 nFlag)
{
    return LLD_SUCCESS;
}

```

MErase erases multiple blocks of NAND flash memory simultaneously, while **Erase** erases only one block of NAND flash memory. In this example, we assume that NAND emulator does not support an erase operation for multiple blocks, thus MErase is not provided.

```

INT32
MyLLD_EraseVerify(UINT32 nDev, LLDMEArg *pstMEArg, UINT32 nFlag)
{
    return LLD_SUCCESS;
}

```

EraseVerify verifies an erase operation whether it checks blocks are properly erased. Do not modify the template. In this example, we assume that NAND emulator does not support an erase operation for multiple blocks, thus EraseVerify is not provided.

- 8) Add the following code in bold to implement CopyBack function. CopyBack function supports a page copy functionality using the internal buffer in a NAND device.

```

INT32
MyLLD_CopyBack(UINT32 nDev, CpBkArg *pstCpArg, UINT32 nFlag)
{
    UINT8  aBuf[2112];
    UINT32 nIdx;
    UINT32 nBlkOff;
    UINT32 nPgOff;
    UINT32 nSctNum;
    UINT32 nOffset;
    UINT32 nPos;

    nBlkOff = pstCpArg->nSrcSn / SCTS_PER_BLK;
    nPgOff = (pstCpArg->nSrcSn - nBlkOff * SCTS_PER_BLK)
            / SCTS_PER_PG;

    nPos = nBlkOff * BLK_SIZE + nPgOff * PG_SIZE;

    PAM_Memcpy(aBuf, pNANDArray + nPos, 2112);

    for (nIdx = 0; nIdx < pstCpArg->nRndInCnt; nIdx++)
    {
        nSctNum = pstCpArg->pstRndInArg[nIdx].nOffset / 1024;
        nOffset = pstCpArg->pstRndInArg[nIdx].nOffset % 1024;

        if (nOffset < 512)
        {
            PAM_Memcpy(aBuf + 512 * nSctNum + nOffset,
                       pstCpArg->pstRndInArg[nIdx].pBuf,
                       pstCpArg->pstRndInArg[nIdx].nNumOfBytes);
        }
        else
        {
            PAM_Memcpy(aBuf + 512 * 4 + 16 * nSctNum + nOffset,
                       pstCpArg->pstRndInArg[nIdx].pBuf,
                       pstCpArg->pstRndInArg[nIdx].nNumOfBytes);
        }
    }

    nBlkOff = pstCpArg->nDstSn / SCTS_PER_BLK;
    nPgOff = (pstCpArg->nDstSn - nBlkOff * SCTS_PER_BLK)
            / SCTS_PER_PG;

    nPos = nBlkOff * BLK_SIZE + nPgOff * PG_SIZE;

    for (nIdx = 0; nIdx < 2112; nIdx++)
    {
        pNANDArray[nPos + nIdx] &= aBuf[nIdx];
    }

    return LLD_SUCCESS;
}

```

Reference

CopyBack means the operation method to copy pages using the internal buffer in a



NAND device.

This copyback method improves the performance by cutting the transfer time and operation procedure, because this method does not use the external memory. When copying a page using the copyback method, a part of data can be brought the outside device; this is called **Random-in**.

To implement CopyBack function, you must understand a CpBkArg structure. For more information about CpBkArg, refer to LLD function XXX_CopyBack.

- 9) Do not modify ChkInitBadBlk, SetRWArea, and IOCtl functions, because you use the simple NAND emulator.

```
INT32
MyLLD_ChkInitBadBlk(UINT32 nDev, UINT32 nPbn)
{
    return LLD_INIT_GOODBLOCK;
}

INT32
MyLLD_SetRWArea(UINT32 nDev, UINT32 n1stUB, UINT32 nNumOfUBs)
{
    return LLD_SUCCESS;
}

INT32
MyLLD_IOCTL(UINT32 nDev, UINT32 nCode, UINT8 *pBufI,
             UINT32 nLenI, UINT8 *pBufO, UINT32 nLenO,
             UINT32 *pByteRet)
{
    return LLD_SUCCESS;
}
```

ChkInitBadBlk is to check whether a block is an initial bad block or not.

If the value of the bad mark position in the first or second page of a block is not 0xff(a normal statement), the block is the initial bad block.

This function is implemented to read the first or second page of a block and check it bad or not. In general, you implement this function using Read function in real NAND device.

SetRWArea is used when NAND device supports the hardware write protection.

- 10) Do not modify Deferred Check Operation functions. The current template always returns LLD_SUCCESS.

```
INT32
MyLLD_GetPrevOpData(UINT32 nDev, UINT8 *pMBuf, UINT8 *pSBuf)
{
    return LLD_SUCCESS;
}

INT32
MyLLD_FlushOp(UINT32 nDev)
{
    return LLD_SUCCESS;
}
```

```
}

```

For detailed information about the function related to Deferred Check Operation, refer to Chapter 5.2. LLD APIs. For detailed information about the functionality of Deferred Check Operation, refer to Chapter 7.4. Deferred Check Operation.

After editing MyLLD.cpp, save the file and close it.

By far, you make a simple NAND emulator and port the device driver file MyLLD.cpp operating to the simple NAND emulator. Next, you implement a PAM file.

3.2.4. PAM Porting

PAM, a platform-dependent module, links XSR to the platform. This document shows you the process to port OAM to Win32 and port LLD by making a simple NAND emulator. You implement PAM based on that porting environment.

In \Platform\PAM directory, there is a template file **MyPAM.cpp** in **Template** folder. This template file contains 9 functions as follows.

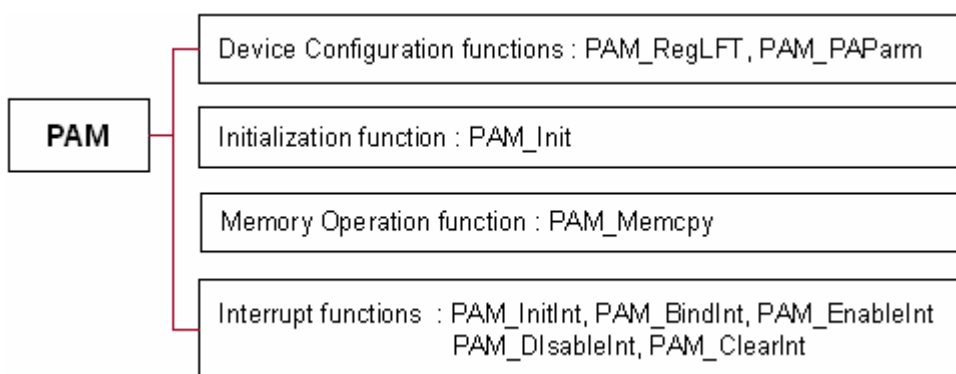


Figure 3-7. PAM Function Classification

Now, port PAM to the simple NAND emulator made by LLD.

- 1) Make a duplicate of the existing **Template** folder in \Platform\PAM, and name the new folder as **MyPAM**.

Check that there is a file **MyPAM.cpp** in the newly named folder **MyPAM**.

- 2) Open the existing file MyPAM.cpp in an editor.
- 3) Update the path of a LLD header file as follows.

☐ Before Modification

```
#include "../../XSR/LLD/Template/MyLLD.h"

```

□ After Modification

```
#include "../../XSR/LLD/MyLLD/MyLLD.h"
```

Depending on the type of the released package, the proper path of LLD header files can be different from above.

- 4) Do not modify initialization functions because you do not real NAND flash memory.

```
VOID
PAM_Init(VOID)
{
}
```

- 5) Add the following code in bold to implement a function **RegLFT**, which registers LLD to XSR.

```
VOID
PAM_RegLFT(VOID *pstFunc)
{
    LowFuncTbl *pstLFT;

    pstLFT                                = (LowFuncTbl*)pstFunc;
    pstLFT[0].Init                        = MyLLD_Init;
    pstLFT[0].Open                        = MyLLD_Open;
    pstLFT[0].Close                       = MyLLD_Close;
    pstLFT[0].Read                        = MyLLD_Read;
    pstLFT[0].Write                       = MyLLD_Write;
    pstLFT[0].CopyBack                    = MyLLD_CopyBack;
    pstLFT[0].Erase                       = MyLLD_Erase;
    pstLFT[0].GetDevInfo                  = MyLLD_GetDevInfo;
    pstLFT[0].ChkInitBadBlk               = MyLLD_ChkInitBadBlk;
    pstLFT[0].FlushOp                     = MyLLD_FlushOp;
    pstLFT[0].SetRWArea                   = MyLLD_SetRWArea;
    pstLFT[0].GetPrevOpData                = MyLLD_GetPrevOpData;
    pstLFT[0].IOctl                       = MyLLD_IOCTL;
    pstLFT[0].MRead                       = MyLLD_MRead;
    pstLFT[0].MWrite                      = MyLLD_MWrite;
    pstLFT[0].MErase                      = MyLLD_MErase;
    pstLFT[0].EraseVerify                  = MyLLD_EraseVerify;
}
```

RegLFT registers LLD to BML.

If you change LLD function name in MyLLD.cpp and MyLLD h, you must rename LLD function name in this function.

- 6) Add the following code in bold to implement a function **GetPAParm**, which configures the platform and device.

```
#define VOL0 0
#define VOL1 1
```

```
#define    DEV0    0
#define    DEV1    1
#define    DEV2    2
#define    DEV3    3

VOID*
PAM_GetPAParm(VOID)
{
    gstParm[VOL0].nBaseAddr[DEV0] = 0x20000000;
    gstParm[VOL0].nBaseAddr[DEV1] = NOT_MAPPED;
    gstParm[VOL0].nBaseAddr[DEV2] = NOT_MAPPED;
    gstParm[VOL0].nBaseAddr[DEV3] = NOT_MAPPED;

    gstParm[VOL0].nEccPol           = SW_ECC;
    gstParm[VOL0].nLSchemePol       = SW_LOCK_SCHEME;
    gstParm[VOL0].bByteAlign        = FALSE32;
    gstParm[VOL0].nDevsInVol        = 1;
    gstParm[VOL0].pExInfo           = NULL;

    return (VOID *) gstParm;
}
```

GetPAParm returns the information to XSR and LLD.

This example only implements [**VOL0**] and [**DEV0**] because this example uses 1 device. [**VOL0**] has the following functionalities: it uses software ECC, it uses the software lock scheme, and it does not use byte aligning. In general, you must implement the total setting about nBaseAddr[] as many as the device number per a volume.

- 7) Do not modify the interrupt functions because you do not use real NAND flash memory.

```
VOID
PAM_InitInt(UINT32 nLogIntId)
{
}

VOID
PAM_BindInt(UINT32 nLogIntId)
{
}

VOID
PAM_EnableInt(UINT32 nLogIntId)
{
}

VOID
PAM_DisableInt(UINT32 nLogIntId)
{
}

VOID
PAM_ClearInt(UINT32 nLogIntId)
```


$$\left\{ \begin{array}{l} \end{array} \right\}$$

For detailed information about the interrupt functionality, refer to Chapter 7.2. Interrupt.

- 8) Do not modify the memory copy function because you do not use real NAND flash memory.

```

VOID
PAM_Memcpy(VOID *pDst, VOID *pSrc, UINT32 nLen)
{
    OAM_Memcpy(pDst, pSrc, nLen);
}

```

PAM_Memcpy copies data from source to destination using system supported function. In this porting example, PAM_Memcpy just calls OAM_Memcpy because you do not use real platform.

After editing MyPAM.cpp, save the file and close it.

By far, you implement `MyPAM.cpp`. You implement all required porting files, so create the application to execute `XSR`.

3.2.5. Application Creating

Now, create the application using the implemented files and test that XSR operates well.

- 1) To make a main file of the project, add the following code in an editor and name it as **XSRHello.cpp** in **XSR** folder.

```
#include <XSR.h>
#include <stdio.h>

#define NUM_OF_VOLS 2

#define VOL0      0
#define VOL1      1

UINT8 aRWBuf[512];

INT32
SeqWrite(UINT32 nTotalLogScts)
{
    UINT32      nIdx;
    UINT32      nIdx2;
    INT32       nErr;

    printf("      ");
    for (nIdx = 0; nIdx < nTotalLogScts; nIdx++)
    {
        printf("\b\b\b\b\b\b\b\b\b%-8d", nTotalLogScts - nIdx);
        for (nIdx2 = 0; nIdx2 < 512; nIdx2++)
```

```

        {
            aRWBuf[nIdx2] = (UINT8)(nIdx + nIdx2);
        }
        nErr = STL_Write(VOL0 , PARTITION_ID_FILESYSTEM,
                        nIdx, 1, aRWBuf);
        if (nErr != STL_SUCCESS) break;
    }
    printf("\b\b\b\b\b\b\b\b\b\b");

    return nErr;
}

INT32
SeqVerify(UINT32 nTotalLogScts)
{
    UINT32      nIdx;
    UINT32      nIdx2;
    INT32      nErr;

    for (nIdx = 0; nIdx < nTotalLogScts; nIdx++)
    {
        nErr = STL_Read(VOL0 , PARTITION_ID_FILESYSTEM,
                        nIdx, 1, aRWBuf);
        if (nErr != STL_SUCCESS) break;

        for (nIdx2 = 0; nIdx2 < 512; nIdx2++)
        {
            if (aRWBuf[nIdx2] != (UINT8)(nIdx + nIdx2)) break;
        }
        if (nIdx2 < 512)
        {
            nErr = 1;
            break;
        }
    }

    return nErr;
}

VOID
main(VOID)
{
    STLInfo      stSTLinfo;
    STLConfig     stSTLconfig;
    XSRPartI     stPart[NUM_OF_VOL0];
    INT32      nErr;
    UINT32      nTotalLogScts;
    UINT32      nIdx;

    printf("### Hello XSR ###\r\n");

    do
    {
        STL_Init();

        OAM_Memcpy(&stPart[VOL0].aSig, "XSRPARTI",

```

```

        BML_MAX_PART_SIG);

    stPart[VOL0].nVer                = 0x00010000;
    stPart[VOL0].numOfPartEntry      = 1;

    stPart[VOL0].stPEntry[0].nID     =
        PARTITION_ID_FILESYSTEM;
    stPart[VOL0].stPEntry[0].nAttr    = BML_PI_ATTR_RW;
    stPart[VOL0].stPEntry[0].n1stVbn = 0;
    stPart[VOL0].stPEntry[0].numOfBlks = 100;

    nErr = BML_Format(VOL0, &stPart[VOL0],
        BML_INIT_FORMAT);
    if (nErr != BML_SUCCESS)
    {
        printf("[:ERR] BML_Format() returns ERROR : %x\r\n",
            nErr);
        break;
    }

    stSTLconfig.nFillFactor          = 100;
    stSTLconfig.numOfRsvUnits        = 2;
    stSTLconfig.nBlksPerUnit         = 1;
    stSTLconfig.nSnapshots           = 4;

    nErr = STL_Format(VOL0, PARTITION_ID_FILESYSTEM,
        &stSTLconfig, 0);
    if (nErr != STL_SUCCESS)
    {
        printf("[:ERR] STL_Format() returns ERROR : %x\r\n",
            nErr);
        break;
    }

    stSTLinfo.nSamBufFactor          = 100;
    stSTLinfo.bASyncMode             = FALSE32;

    nErr = STL_Open(VOL0, PARTITION_ID_FILESYSTEM,
        &stSTLinfo);
    if (nErr != STL_SUCCESS)
    {
        printf("[:ERR] STL_Open() returns ERROR : %x\r\n",
            nErr);
        break;
    }

    nTotalLogScts = stSTLinfo.nTotalLogScts;

    printf("* Simple Test()\r\n");
    for (nIdx = 0; nIdx < 10; nIdx++)
    {
        printf("\t(%2d) Write : ", nIdx + 1);
        if (SeqWrite(nTotalLogScts) == STL_SUCCESS)
        {
            printf("OK\r\n");
        }
    }

```

```

else
{
    printf("ERROR\r\n");
    break;
}

printf("\t    Verify : ", nIdx + 1);
if (SeqVerify(nTotalLogScts) == STL_SUCCESS)
{
    printf("OK\r\n");
}
else
{
    printf("ERROR\r\n");
    break;
}
}
} while (0);

STL_Close(VOL0, PARTITION_ID_FILESYSTEM);

printf("### Bye XSR ###\r\n");
}

```

- 2) Execute Visual Studio, and create a new Win32 Console project.
This is an example to name the new project as **XSRHello**.
- 3) Include the following files in the new project **XSRHello**.
Then, you can see the project structure as the following figure.

- ☐ XSR**XSRHello.cpp**
- ☐ XSR\OAM\MyOAM**MyOAM.cpp**
- ☐ XSR\LLD\MyLLD**MyLLD.cpp**
 MyLLD.h
- ☐ XSR\Inc**BML.h**
 LLD.h
 OAM.h
 PAM.h
 STL.h
 XSR.h
 XsrTypes.h
- ☐ XSR\lib\Generic\VS60\Retail**XSREmul.lib**
- ☐ Platform\PAM\MyPAM**MyPAM.cpp**

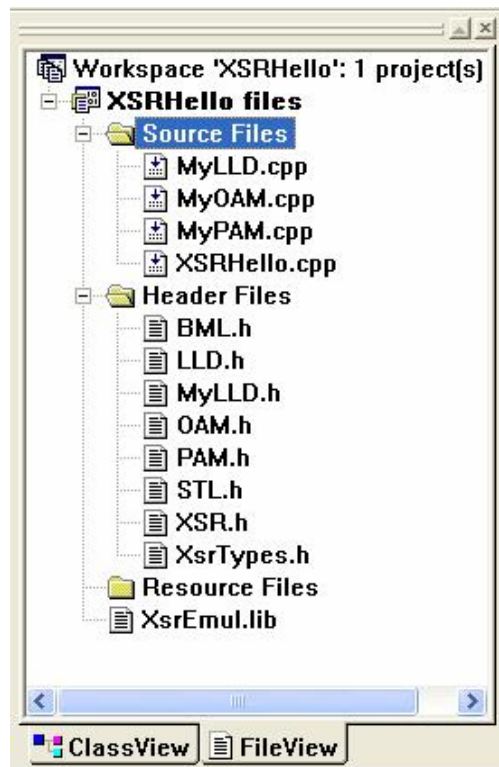


Figure 3-8. XSRHello Project Structure

- 4) On **Project** menu, click **Setting**, and then **Project Setting** pop-up window is opened. On **Project Setting** pop-up window, set as follows.

- ☐ On C/C++ tap, click General in Category list.
Add “, **OAM_RTLMSG_ENABLE**” in Preprocessor definitions text box.


Reference

If you declare **OAM_RTLMSG_ENABLE**, XSR_RTL_PRINT() function is linked to OAM_Debug() function.

If you declare **OAM_DBGMSG_ENABLE**, XSR_DBG_PRINT() function is linked to OAM_Debug() function.

- ☐ On C/C++ tap, click **Preprocessor** in **Category** list.
Add “. . . /inc” in **Additional include directories** text box.

Now, all preparation for executing XSRHello project is done.

- 5) Click **Execute XSRHello.cpp** on Build menu, or press **Ctrl + F5** key, or click  button on menu toolbar. Then, the project is executed.

The project is compiled and built if there is no error, and the project is performed. The command window is opened, so you can XSRHello project is working.

```

### Hello XSR ###
* Simple Test()
  ( 1) Write  : OK
      Verify  : OK
  ( 2) Write  : OK
      Verify  : OK
  ( 3) Write  : OK
      Verify  : OK
  ( 4) Write  : OK
      Verify  : OK
  ( 5) Write  : OK
      Verify  : OK
  ( 6) Write  : OK
      Verify  : OK
  ( 7) Write  : OK
      Verify  : OK
  ( 8) Write  : OK
      Verify  : OK
  ( 9) Write  : OK
      Verify  : OK
 (10) Write  : OK
      Verify  : OK
### Bye XSR ###
Press any key to continue

```

Figure 3-9. XSRHello Project Working Screen

XSRHello project working process is as follows;

1. First, call STL_Init,
2. Format BML
3. Format STL
4. Open STL,
5. Then, repeat Sequential Write and Sequential Verify 10 times.

Remark

The preceding API calling sequence is for the NAND device that is called for the first time. BML_Format() should be called just one time to initialize NAND device. And STL_Format() should be called for each RW partitions. If BML_Format() and STL_Format() are called for the NAND device before, STL_Init() and STL_Open() are enough to next opening the NAND device.

Reference

Sequential Write means writing data from the sector 0 to the last as many as defined sequentially.
Sequential Verify means verifying the written data from the sector 0 to the last as many as defined sequentially.

Once all working is finished, it shows “### Bye XSR ###”. Press any key to close XSRHello command window.



You follow XSR porting example and execute it. Now, you can port XSR in various environments by implementing OAM, LLD and PAM.

4. OAM (OS Adaptation Module)

This chapter describes the definition, system architecture, features, and APIs of OAM.

4.1. Description & Architecture

OAM is an abbreviation of OS Adaptation Module. OAM is to receive the services that OS provides. OAM is responsible for OS-dependent part of XSR layers (STL and BML). If OS is changed, a user only changes OAM.

For example, a layer of XSR wants to write memory. The memory request service is dependent on OS. Each layer calls an adaptation module OAM to use OS functionalities. Therefore a user must implement OAM suitable for the OS when a user ports XSR.

Figure 4-1 shows OAM in XSR system architecture.

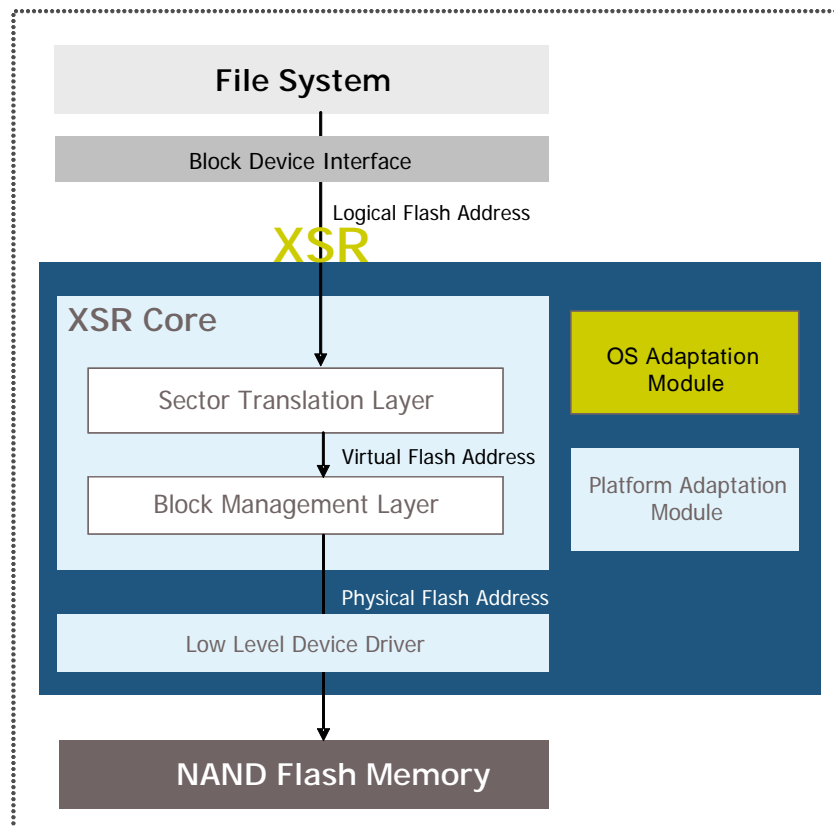


Figure 4-1. OAM in XSR System Architecture

OAM has 21 functions that are classified into 6 categories as follows.

□ **ANSI functions** : memory allocation, memory free, memory copy, memory setting, and memory comparison



- ☐ **Message print function :** message print
- ☐ **Semaphore functions:** semaphore creation, semaphore destroy, semaphore acquire, and semaphore release
- ☐ **Interrupt functions:** interrupt initialization, interrupt bind, interrupt enable, interrupt disable, and interrupt clear
- ☐ **Timer functions :** timer setting, timer return, and time waiting
- ☐ **Other functions :** address translation, idle operation, RO partition check

In the next chapter, OAM APIs are covered in detail.

4.2. API

This chapter describes OAM APIs. .

Reference

All the OAM function has a prefix “OAM_” on each function name.

Table 4-1 shows the lists of OAM APIs.

The right row in table shows that the function is **M**andatory or **O**ptional or **R**ecommended. Optional functions should be existed, but contents of the functions does not need to be implemented.

Table 4-1. OAM API

Function	Description	
OAM_Malloc	This function allocates memory for XSR.	M
OAM_Free	This function frees memory that XSR allocates.	M
OAM_Memcpy	This function copies from the source data to the destination data.	M
OAM_Memset	This function sets data of a specific buffer.	M
OAM_Memcmp	This function compares data of two buffers.	M
OAM_Debug	This function is called when XSR prints message.	R
OAM_CreateSM	This function creates the semaphore.	O
OAM_DestroySM	This function destroys the semaphore.	O
OAM_AcquireSM	This function acquires the semaphore.	O
OAM_ReleaseSM	This function releases the semaphore.	O
OAM_InitInt	This function initializes the specified logical interrupt for NAND device.	O
OAM_BindInt	This function binds the specified interrupt for NAND device.	O
OAM_EnableInt	This function enables the specified interrupt for NAND device.	O
OAM_DisableInt	This function disables the specified interrupt for NAND device.	O
OAM_ClearInt	This function clears the specified interrupt for NAND device.	O
OAM_ResetTimer	This function resets timer.	O
OAM_GetTime	This function returns the current time value.	O
OAM_WaitNMSec	This function is called for delaying as a unit of milliseconds.	R
OAM_Pa2Va	This function is called for the address translation to access to the hardware from the system using virtual memory.	R
OAM_Idle	This function is called at idle time.	O
OAM_GetROLockFlag	This function is called when XSR determines whether Partition is RO or not.	O

OAM_Malloc

DESCRIPTION

This function allocates memory for XSR.

SYNTAX

```
VOID *
OAM_Malloc(UINT32 nSize)
```

PARAMETERS

Parameter	Type	In/Out	Description
nSize	UINT32	In	Size to be allocated

RETURN VALUE

Return Value	Description
VOID	allocated memory buffer pointer If this function fails, the return value is NULL.

REMARKS

This function is a mandatory ANSI function. ANSI function is implemented to map one-to-one with the standard ANSI function.

This function is called by the function that wants to use memory.

EXAMPLE

(1) Example to call the function

```
#include <OAM.h>
#include <XSRTypes.h>

VOID
Example(VOID)
{
    UINT8 *MainBuf = (UINT8 *)OAM_Malloc(512);

    OAM_Free(MainBuf);
}
```

(2) Example to implement the function in Symbian

```
void *
OAM_Malloc(UINT32 nSize)
{
    return Kern::Alloc(nSize);
}
```

SEE ALSO

OAM_Free

OAM_Free

DESCRIPTION

This function frees memory that XSR allocates.

SYNTAX

```
VOID  
OAM_Free(VOID *pMem)
```

PARAMETERS

Parameter	Type	In/Out	Description
pMem	VOID *	In	Pointer to be free

RETURN VALUE

None

REMARKS

This function is a mandatory ANSI function. ANSI function is implemented to map one-to-one with the standard ANSI function.

This function is called by the function that wants to free memory.

EXAMPLE

(1) Example to call the function

```
#include <OAM.h>  
#include <XSRTypes.h>  
  
VOID  
Example(VOID)  
{  
    UINT8 *MainBuf = (UINT8 *)OAM_Malloc(512);  
  
    OAM_Free(MainBuf);  
}
```

(2) Example to implement the function in Symbian

```
VOID  
OAM_Free(VOID *pMem)  
{  
    Kern::Free(pMem);  
    return ;  
}
```

SEE ALSO

OAM_Malloc

OAM_Memcpy

DESCRIPTION

This function copies from the source data to the destination data.

SYNTAX

```
VOID
OAM_Memcpy(VOID *pDst, VOID *pSrc, UINT32 nLen)
```

PARAMETERS

Parameter	Type	In/Out	Description
pDst	VOID *	Out	Array Pointer of destination data to be copied
pSrc	VOID *	In	Array Pointer of source data to be copied
nLen	UINT32	In	Length to be copied

RETURN VALUE

None

REMARKS

This function is a mandatory ANSI function. ANSI function is implemented to map one-to-one with the standard ANSI function.

This function is called by the function that wants to copy data in the source buffer to data in the destination buffer.

EXAMPLE

(1) Example to call the function

```
#include <OAM.h>
#include <XSRTypes.h>

VOID
Example(VOID)
{
    UINT8 *SrcBuf = (UINT8 *)OAM_Malloc(512);
    UINT8 *DstBuf = (UINT8 *)OAM_Malloc(512);

    OAM_Memcpy(DstBuf, SrcBuf, (512));
}
```

(2) Example to implement the function in Symbian

```
VOID
OAM_Memcpy(VOID *pDst, VOID *pSrc, UINT32 nLen)
{
    memcpy((TAny*)(pDst), (const TAny*)(pSrc),
        (unsigned int)nLen);
}
```



SEE ALSO

OAM_Memset, OAM_Memcmp

OAM_Memset

DESCRIPTION

This function sets data of a specific buffer.

SYNTAX

```
VOID
OAM_Memset(VOID *pDst, UINT8 nV, UINT32 nLen)
```

PARAMETERS

Parameter	Type	In/Out	Description
pDst	VOID *	Out	Array Pointer of destination data to be copied
nV	UINT8	In	Value to be set
nLen	UINT32	In	Length to be set

RETURN VALUE

None

REMARKS

This function is a mandatory ANSI function. ANSI function is implemented to map one-to-one with the standard ANSI function.

This function is called by the function that wants to set the source buffer's own data.

EXAMPLE

(1) Example to call the function

```
#include <OAM.h>
#include <XSRTypes.h>

VOID
Example(VOID)
{
    UINT8 *pBuf = (UINT8 *)OAM_Malloc(512);

    OAM_Memset(pBuf, 0xFF, (512));
}
```

(2) Example to implement the function in Symbian

```
VOID
OAM_Memset(VOID *pSrc, UINT8 nV, UINT32 nLen)
{
    memset((TAny*)(pSrc), (TInt)(nV), (unsigned int)(nLen));
}
```

SEE ALSO

OAM_Memcpy, OAM_Memcmp

OAM_Memcmp

DESCRIPTION

This function compares data of two buffers.

SYNTAX

```
INT32
OAM_Memcmp(VOID *pBuf1, VOID *pBuf2, UINT32 nLen)
```

PARAMETERS

Parameter	Type	In/Out	Description
pBuf1	VOID *	In	Pointer of Buffer1
pBuf2	VOID *	In	Pointer of Buffer2
nLen	UINT32	In	Length to be compared

RETURN VALUE

Return Value	Description
Negative number	When pBuf1 is smaller than pBuf2.
0	When pBuf1 is the same as pBuf2.
Positive number	When pBuf1 is bigger than pBuf2.

REMARKS

This function is a mandatory ANSI function. ANSI function is implemented to map one-to-one with the standard ANSI function.

This function is called by the function that wants to compare data of two buffers.

EXAMPLE

(1) Example to call the function

```
#include <OAM.h>
#include <XSRTypes.h>

VOID
Example(VOID)
{
    UINT8 Re;
    UINT8 *pBuf1 = (UINT8 *)OAM_Malloc(512);
    UINT8 *pBuf2 = (UINT8 *)OAM_Malloc(512);

    Re = OAM_Memcmp(pBuf1, pBuf2);

    if (Re != 0)
    {
        printf(" Compare fail(%x)\n", Re);
    }
}
```

(2) Example to implement the function in Symbian

```
INT32  
OAM_Memcmp(VOID *pSrc, VOID *pDst, UINT32 nLen)  
{  
    INT8 *pS1 = (UINT8 *)pSrc;  
    INT8 *pD1 = (UINT8 *)pDst;  
  
    while (nLen--)  
    {  
        if (*pS1 != *pD1)  
            return (*pS1 - *pD1);  
        pS1++;  
        pD1++;  
    }  
  
    return 0;  
}
```

SEE ALSO

OAM_Memcpy, OAM_Memset



OAM_Debug

DESCRIPTION

This function is called when XSR prints messages such as error or debug or etc.

SYNTAX

```
VOID  
OAM_Debug(VOID *pFmt, ...)
```

PARAMETERS

Parameter	Type	In/Out	Description
pFmt	VOID *	In	Data to be printed

RETURN VALUE

None

REMARKS

This function is recommended.

EXAMPLE

(1) Example to call the function

```
#include <OAM.h>  
#include <XSRTypes.h>  
  
VOID  
Example(VOID)  
{  
    OAM_Debug("Print debug message");  
}
```

(2) Example to implement the function in Win32

```
VOID  
OAM_Debug(VOID *pFmt, ...)  
{  
    static char  aStr[4096];  
    va_list      ap;  
  
    va_start(ap, pFmt);  
    vsprintf(aStr, (char *) pFmt, ap);  
    printf(aStr);  
  
    va_end(ap);  
}
```

SEE ALSO

OAM_CreateSM

DESCRIPTION

This function creates the semaphore.

SYNTAX

```
BOOL32
OAM_CreateSM(SM32 *pHandle)
```

PARAMETERS

Parameter	Type	In/Out	Description
pHandle	SM32 *	Out	Handle of the semaphore to be created

RETURN VALUE

Return Value	Description
TRUE32	If this function creates the semaphore successfully, it returns TRUE32.
FALSE32	Else it returns FALSE32.

REMARKS

This function is an optional semaphore function.

This function is called by the function that wants to create the semaphore.

XSR regards the number of the semaphore token as 0 after calling OAM_CreateSM(). Thus, when a user implements OAM_CreateSM(), a user should set the initial value of the semaphore token as 0.

EXAMPLE

(1) Example to call the function

```
#include <OAM.h>
#include <XSRTypes.h>

VOID
Example(VOID)
{
    SM32      nSm;

    if (OAM_CreateSM (&(nSm)) == FALSE32)
    {
        FVM_DBG_PRINT((TEXT("OAM_CreateSM Error\r\n")));
    }
}
```

(2) Example to implement the function in pSOS

```
BOOL32
OAM_CreateSM(SM32 *pHandle)
```

```
{  
    BOOL32 bRet = TRUE32;  
  
    if (sm_create("XSRS",  
                 0,  
                 SM_LOCAL | SM_FIFO | SM_UNBOUNDED,  
                 (ULONG *)pHandle) != 0)  
    {  
        bRet = FALSE32;  
    }  
  
    return bRet;  
}
```

SEE ALSO

OAM_DestroySM, OAM_AcquireSM, OAM_ReleaseSM

OAM_DestroySM

DESCRIPTION

This function destroys the semaphore.

SYNTAX

```
BOOL32
OAM_DestroySM(SM32 nHandle)
```

PARAMETERS

Parameter	Type	In/Out	Description
nHandle	SM32	In	Handle of the semaphore to be destroyed

RETURN VALUE

Return Value	Description
TRUE32	If this function destroys the semaphore successfully, it returns TRUE32.
FALSE32	Else it returns FALSE32.

REMARKS

This function is an optional semaphore function.

This function is called by the function that wants to destroy the semaphore.

EXAMPLE

(1) Example to call the function

```
#include <OAM.h>
#include <XSRTypes.h>

VOID
Example(VOID)
{
    SM32      nSm;

    if (OAM_CreateSM (&(nSm)) == FALSE32)
    {
        FVM_DBG_PRINT((TEXT("OAM_CreateSM Error\r\n")));
    }

    OAM_DestroySM(nSm);
}
```

(2) Example to implement the function in pSOS

```
BOOL32
OAM_DestroySM(SM32 nHandle)
{
    BOOL32 bRet = TRUE32;
```



```
    if (sm_delete(nHandle) != 0)
    {
        bRet = FALSE32;
    }

    return bRet;
}
```

SEE ALSO

OAM_CreateSM, OAM_AcquireSM, OAM_ReleaseSM

OAM_AcquireSM

DESCRIPTION

This function acquires the semaphore.

SYNTAX

```
BOOL32
OAM_AcquireSM(SM32 nHandle)
```

PARAMETERS

Parameter	Type	In/Out	Description
nHandle	SM32	In	Handle of the semaphore to be acquired

RETURN VALUE

Return Value	Description
TRUE32	If this function acquires the semaphore successfully, it returns TRUE32.
FALSE3	Else it returns FALSE32.

REMARKS

This function is an optional semaphore function.

This function is called by the function that wants to acquire the semaphore.

EXAMPLE

(1) Example to call the function

```
#include <OAM.h>
#include <XSRTypes.h>

VOID
Example(VOID)
{
    SM32      nSm;

    if (OAM_CreateSM (&nSm) == FALSE32)
    {
        FVM_DBG_PRINT((TEXT("OAM_CreateSM Error\r\n")));
    }
    OAM_ReleaseSM(nSm);

    OAM_AcquireSM(nSm);
    OAM_ReleaseSM(nSm);
}
```

(2) Example to implement the function in pSOS

```
BOOL32
OAM_AcquireSM(SM32 nHandle)
```

```
{  
    BOOL32 bRet = TRUE32;  
  
    if (sm_p(nHandle, SM_WAIT, 0) != 0)  
    {  
        bRet = FALSE32;  
    }  
  
    return bRet;  
}
```

SEE ALSO

OAM_CreateSM, OAM_DestroySM, OAM_ReleaseSM

OAM_ReleaseSM

DESCRIPTION

This function releases the semaphore.

SYNTAX

```
BOOL32
OAM_ReleaseSM(SM32 nHandle)
```

PARAMETERS

Parameter	Type	In/Out	Description
nHandle	SM32	In	Handle of the semaphore to be released

RETURN VALUE

Return Value	Description
TRUE32	If this function releases the semaphore successfully, it returns TRUE32.
FALSE32	Else it returns FALSE32.

REMARKS

This function is an optional semaphore function.

This function is called by the function that wants to release the semaphore.

EXAMPLE

(1) Example to call the function

```
#include <OAM.h>
#include <XSRTypes.h>

VOID
Example(VOID)
{
    SM32      nSm;

    if (OAM_CreateSM (&(nSm)) == FALSE32)
    {
        FVM_DBG_PRINT((TEXT("OAM_CreateSM Error\r\n")));
    }
    OAM_ReleaseSM(nSm);

    OAM_AcquireSM(nSm);
    OAM_ReleaseSM(nSm);
}
```

(2) Example to implement the function in pSOS

```
BOOL32
OAM_ReleaseSM(SM32 nHandle)
```



```
{  
    BOOL32 bRet = TRUE32;  
  
    if (sm_v(nHandle) != 0)  
    {  
        bRet = FALSE32;  
    }  
  
    return bRet;  
}
```

SEE ALSO

OAM_CreateSM, OAM_DestroySM, OAM_AcquireSM

OAM_InitInt

DESCRIPTION

This function initializes the specified logical interrupt for NAND device.

SYNTAX

```
VOID
OAM_InitInt(UINT32 nLogIntId)
```

PARAMETERS

Parameter	Type	In/Out	Description
nLogIntId	UINT32	In	Logical interrupt ID number

RETURN VALUE

None

REMARKS

This function is an optional interrupt function.
Currently, this function is used for asynchronous operation.

EXAMPLE

(1) Example to call the function

```
#include <OAM.h>
#include <XSRTypes.h>

#define INT_ID_NAND_0 (0) /* Interrupt ID : 1st NAND */

VOID
Example(VOID)
{
    OAM_InitInt(INT_ID_NAND_0);
}
```

(2) Example to implement the function in Symbian

```
VOID
OAM_InitInt(UINT32 nLogIntId)
{
    switch (nLogIntId)
    {
        case XSR_INT_ID_NAND_0:
            iNANDInt = new TNandInterrupt;
            break;

        default:
            break;
    }
}
```

**SEE ALSO**

OAM_BindInt, OAM_EnableInt, OAM_DisableInt, OAM_ClearInt

OAM_BindInt

DESCRIPTION

This function binds the specified interrupt for NAND device.

SYNTAX

```
VOID
OAM_BindInt(UINT32 nLogIntId, UINT32 nPhyIntId)
```

PARAMETERS

Parameter	Type	In/Out	Description
nLogIntId	UINT32	In	Logical interrupt ID number
nPhyIntId	UINT32	In	Physical interrupt ID number

RETURN VALUE

None

REMARKS

This function is an optional interrupt function.
Currently, this function is used for asynchronous operation.

EXAMPLE

(1) Example to call the function

```
#include <OAM.h>
#include <XSRTypes.h>

#define INT_ID_NAND_0 (0) /* Interrupt ID : 1st NAND */

enum TPlatformInterruptId
{
    EIrqExt0,          // IRQ 0
    EIrqExt1,          // IRQ 1
    EIrqExt2,          // IRQ 2
    EIrqExt3,          // IRQ 3
    EIrqExt4_7,        // IRQ 4
    EIrqExt8_23,       // IRQ 5
    EIrqCAM,           // IRQ 6
    EIrqBatteryFault,  // IRQ 7
    EIrqTick,          // IRQ 8
    EIrqWatchdog,      // IRQ 9
    EIrqTimer0,        // IRQ 10
    EIrqTimer1,        // IRQ 11
    EIrqTimer2,        // IRQ 12
};

VOID
Example(VOID)
{
```

```
OAM_BindInt(INT_ID_NAND_0, EIrqExt4_7);  
}
```

(2) Example to implement the function in Symbian

```
VOID  
OAM_BindInt(UINT32 nLogIntId, UINT32 nPhyIntId)  
{  
    switch (nLogIntId)  
    {  
        case XSR_INT_ID_NAND_0:  
            iNANDInt->Bind((TInt)nPhyIntId, NandIsr, iNANDInt);  
            break;  
  
        default:  
            break;  
    }  
}
```

SEE ALSO

OAM_InitInt, OAM_EnableInt, OAM_DisableInt, OAM_ClearInt

OAM_EnableInt

DESCRIPTION

This function enables the specified interrupt for NAND device.

SYNTAX

```
VOID
OAM_EnableInt(UINT32 nLogIntId, UINT32 nPhyIntId)
```

PARAMETERS

Parameter	Type	In/Out	Description
nLogIntId	UINT32	In	Logical interrupt ID number
nPhyIntId	UINT32	In	Physical interrupt ID number

RETURN VALUE

None

REMARKS

This function is an optional interrupt function.
Currently, this function is used for asynchronous operation.

EXAMPLE

(1) Example to call the function

```
#include <OAM.h>
#include <XSRTypes.h>

#define INT_ID_NAND_0 (0) /* Interrupt ID : 1st NAND */

enum TPlatformInterruptId
{
    EIrqExt0,          // IRQ 0
    EIrqExt1,          // IRQ 1
    EIrqExt2,          // IRQ 2
    EIrqExt3,          // IRQ 3
    EIrqExt4_7,        // IRQ 4
    EIrqExt8_23,       // IRQ 5
    EIrqCAM,           // IRQ 6
    EIrqBatteryFault,  // IRQ 7
    EIrqTick,          // IRQ 8
    EIrqWatchdog,      // IRQ 9
    EIrqTimer0,        // IRQ 10
    EIrqTimer1,        // IRQ 11
    EIrqTimer2,        // IRQ 12
};

VOID
Example(VOID)
{
```

```
OAM_EnableInt(INT_ID_NAND_0, EIrqExt4_7);  
}
```

(2) Example to implement the function in Symbian

```
VOID  
OAM_EnableInt(UINT32 nLogIntId, UINT32 nPhyIntId)  
{  
    switch (nLogIntId)  
    {  
        case XSR_INT_ID_NAND_0:  
            iNANDInt->Enable((TInt)nPhyIntId);  
            break;  
  
        default:  
            break;  
    }  
}
```

SEE ALSO

OAM_InitInt, OAM_BindInt, OAM_DisableInt, OAM_ClearInt

OAM_DisableInt

DESCRIPTION

This function disables the specified interrupt for NAND device.

SYNTAX

```
VOID
OAM_DisableInt(UINT32 nLogIntId, UINT32 nPhyIntId)
```

PARAMETERS

Parameter	Type	In/Out	Description
nLogIntId	UINT32	In	Logical interrupt ID number
nPhyIntId	UINT32	In	Physical interrupt ID number

RETURN VALUE

None

REMARKS

This function is an optional interrupt function.
Currently, this function is used for asynchronous operation.

EXAMPLE

(1) Example to call the function

```
#include <OAM.h>
#include <XSRTypes.h>

#define INT_ID_NAND_0 (0) /* Interrupt ID : 1st NAND */

enum TPlatformInterruptId
{
    EIrqExt0,          // IRQ 0
    EIrqExt1,          // IRQ 1
    EIrqExt2,          // IRQ 2
    EIrqExt3,          // IRQ 3
    EIrqExt4_7,        // IRQ 4
    EIrqExt8_23,       // IRQ 5
    EIrqCAM,           // IRQ 6
    EIrqBatteryFault,  // IRQ 7
    EIrqTick,          // IRQ 8
    EIrqWatchdog,      // IRQ 9
    EIrqTimer0,        // IRQ 10
    EIrqTimer1,        // IRQ 11
    EIrqTimer2,        // IRQ 12
};

VOID
Example(VOID)
{
```

```
OAM_DisableInt(INT_ID_NAND_0, EIrqExt4_7);  
}
```

(2) Example to implement the function in Symbian

```
VOID  
OAM_DisableInt(UINT32 nLogIntId, UINT32 nPhyIntId)  
{  
    switch (nLogIntId)  
    {  
        case XSR_INT_ID_NAND_0:  
            iNANDInt->Disable((TInt)nPhyIntId);  
            break;  
  
        default:  
            break;  
    }  
}
```

SEE ALSO

OAM_InitInt, OAM_BindInt, OAM_EnableInt, OAM_ClearInt

OAM_ClearInt

DESCRIPTION

This function clears the specified interrupt for NAND device.

SYNTAX

```
VOID
OAM_ClearInt(UINT32 nLogIntId, UINT32 nPhyIntId)
```

PARAMETERS

Parameter	Type	In/Out	Description
nLogIntId	UINT32	In	Logical interrupt ID number
nPhyIntId	UINT32	In	Physical interrupt ID number

RETURN VALUE

None

REMARKS

This function is an optional interrupt function.
Currently, this function is used for asynchronous operation.

EXAMPLE

(1) Example to call the function

```
#include <OAM.h>
#include <XSRTypes.h>

#define INT_ID_NAND_0 (0) /* Interrupt ID : 1st NAND */

enum TPlatformInterruptId
{
    EIrqExt0,          // IRQ 0
    EIrqExt1,          // IRQ 1
    EIrqExt2,          // IRQ 2
    EIrqExt3,          // IRQ 3
    EIrqExt4_7,        // IRQ 4
    EIrqExt8_23,       // IRQ 5
    EIrqCAM,           // IRQ 6
    EIrqBatteryFault,  // IRQ 7
    EIrqTick,          // IRQ 8
    EIrqWatchdog,      // IRQ 9
    EIrqTimer0,        // IRQ 10
    EIrqTimer1,        // IRQ 11
    EIrqTimer2,        // IRQ 12
};

VOID
Example(VOID)
```

```
{  
    OAM_ClearInt(INT_ID_NAND_0, EIrqExt4_7);  
}
```

(2) Example to implement the function in Symbian

```
VOID  
OAM_ClearInt(UINT32 nLogIntId, UINT32 nPhyIntId)  
{  
    switch (nLogIntId)  
    {  
        case XSR_INT_ID_NAND_0:  
            iNANDInt->Clear((TInt)nPhyIntId);  
            break;  
  
        default:  
            break;  
    }  
}
```

SEE ALSO

OAM_InitInt, OAM_BindInt, OAM_EnableInt, OAM_DisableInt

OAM_ResetTimer

DESCRIPTION

This function resets the timer.

SYNTAX

```
VOID  
OAM_ResetTimer(VOID)
```

PARAMETERS

None

RETURN VALUE

None

REMARKS

This function is a recommended timer function.
Currently, this function is used for asynchronous operation.

In current implementation, this function does not use a timer of operating system but uses global counter variables. However, if user wants to implement the asynchronous feature based on execution time of operation, this function should be changed to use the real OS timer instead of the counter.

For more information about timer functions for asynchronous feature, refer to 7.2.5 Timer.

EXAMPLE

(1) Example to call the function

```
#include <OAM.h>  
#include <XSRTypes.h>  
  
VOID  
Example(VOID)  
{  
    OAM_ResetTimer();  
}
```

(2) Example to implement the function in Symbian

```
static UINT32 nTimerCounter = 0;  
static UINT16 nTimerPrevCnt = 0;  
  
VOID  
OAM_ResetTimer(VOID)  
{  
    nTimerPrevCnt = 0;  
    nTimerCounter = 0;  
}
```

**SEE ALSO**

OAM_GetTime, OAM_WaitNMSec

OAM_GetTime

DESCRIPTION

This function returns the current time value.

SYNTAX

```
UINT32
OAM_GetTime(VOID)
```

PARAMETERS

None

RETURN TYPE

Return Type	Description
UINT32	Current time value

REMARKS

This function is a recommended timer function.
Currently, this function is used for asynchronous operation.

In current implementation, this function does not use a timer of operating system but uses global counter variables. However, if user wants to implement the asynchronous feature based on execution time of operation, this function should be changed to use the real OS timer instead of the counter.

For more information about timer functions for asynchronous feature, refer to 7.2.5 Timer.

EXAMPLE

(1) Example to call the function

```
#include <OAM.h>
#include <XSRTypes.h>

VOID
Example(VOID)
{
    UINT32    nStartTime;

    nStartTime = OAM_GetTime();
}
```

(2) Example to implement the function in Symbian

```
UINT32
OAM_GetTime(VOID)
{
    return nTimerCounter++;
}
```



SEE ALSO

OAM_ResetTimer, OAM_WaitNMSec

OAM_WaitNMSec

DESCRIPTION

This function is called for waiting as a unit of milliseconds.

SYNTAX

```
VOID
OAM_WaitNMSec(UINT32 nNMSec)
```

PARAMETERS

Parameter	Type	In/Out	Description
nNMSec	UINT32	In	μsec time for waiting

RETURN TYPE

None

REMARKS

This function is a recommended timer function.
Currently, this function is used to wait the time in handling Power-off error.

For more information about timer functions for power-off recovery, refer to 7.3 Power-Off Recovery.

EXAMPLE

(1) Example to implement the function in Win32

```
VOID
OAM_WaitNMSec(UINT32 nNMSec)
{
    Sleep(nNMSec);
}
```

SEE ALSO

OAM_ResetTimer, OAM_GetTime

OAM_Pa2Va

DESCRIPTION

This function is called for the address translation to access to the hardware from the system using virtual memory.

SYNTAX

```
UINT32
OAM_Pa2Va(UINT32 nPAddr)
```

PARAMETERS

Parameter	Type	In/Out	Description
nPAddr	UINT32	In	Physical address of NAND device

RETURN TYPE

Return Type	Description
UINT32	Virtual address of NAND device that Symbian OS uses

REMARKS

This function is recommended.

EXAMPLE

(1) Example to call the function

```
#include <OAM.h>
#include <XSRTypes.h>

VOID
Example(VOID)
{
    INT32 nBaseAddr = 0x30000000;
    UINT32 nVirtualAddr;

    nVirtualAddr = OAM_Pa2Va(nBaseAddr);
}
```

(2) Example to implement the function in Symbian

```
UINT32
OAM_Pa2Va(UINT32 nPAddr)
{
    DPlatChunkHw *NandChunk;
    TLinAddr NandVirtAddr;

    XSR_DBG_PRINT( (TEXT( ">Nand::Open -> OneNandPhyAddr = 0x%x" ),
                    nPAddr) );

    TInt r=DPlatChunkHw::New(NandChunk,nPAddr,0x80000,
                             EMapAttrSupRw|EMapAttrFullyBlocking);
```



```
if (r != KErrNone)
{
    XSR_DBG_PRINT((TEXT(">Nand::Open -> NandChunk->
                        DoCreateL failed = 0x%x"), r));
    return r;
}

NandVirtAddr = NandChunk->LinearAddress();
XSR_DBG_PRINT((TEXT(">Nand::Open -> OneNandVirtAddr
                    = 0x%x"), NandVirtAddr));

return (UINT32)NandVirtAddr;
}
```

SEE ALSO

OAM_Idle

DESCRIPTION

This function is called when XSR is at idle time.

SYNTAX

```
VOID  
OAM_Idle(VOID)
```

PARAMETERS

None

RETURN TYPE

None

REMARKS

This function is optional.

If XSR is polling on the device status, this function is called. XSR usually keep polling on device status register to perform next operation. This polling takes little time, so ,usually, this function is not needed.to implement. But, under certain condition, it is better to yield CPU to other task instead of just waiting. XSR performance will be decreased, but other task can be performed with XSR simultaneously.

Caution : Do not perform any flash memory operation (XSR operation) in this function.

EXAMPLE

(1) Example to implement the function in Symbian

```
VOID  
OAM_Idle(VOID)  
{  
    /* Default : Do nothing */  
}
```

SEE ALSO

OAM_GetROLockFlag

DESCRIPTION

This function is called when XSR determines Read Only attribute.

SYNTAX

```
BOOL32  
OAM_GetROLockFlag(VOID)
```

PARAMETERS

None

RETURN TYPE

Return Type	Description
BOOL32	TRUE32 with RO partition.

REMARKS

This function is optional.

The function OAM_GetROLockFlag is called when XSR determines whether certain block is in Read-Only partition or not. If XSR finds it is in range of RO partition, this function is called. If it is necessary to regard blocks in RO partition as RW under certain condition, implement this function to return FALSE. Do not modify this function unless you deeply understand internal implementation of XSR. Default implementation of this function just returns TRUE.

When the RO partition should be regarded as RW, this function should return FALSE.

EXAMPLE

(1) Example to implement the function in Symbian

```
BOOL32  
OAM_GetROLockFlag(VOID)  
{  
    return TRUE32;  
}
```

SEE ALSO

5. LLD (Low Level Driver)

This chapter describes the definition, system architecture, features, and APIs of LLD.

5.1. Description & Architecture

LLD is an abbreviation of Low Level Device Driver. LLD is adaptation module of XSR. LLD accesses the real device.

XSR cannot send a command directly to the device. When XSR sends a command to the device, it is needed a kind of translator, converting XSR command to the device-understandable message. The translator is a device driver LLD. LLD directly access to the device.

Reference

Generally, a device is a machine or hardware designed for a purpose. For example, it can include keyboards, mouse, display monitors, hard disk drives, CD-ROM players, printers, audio speakers, and etc. In this document, a device means NAND flash memory.

Generally, a device driver is a program that controls a particular type of a device. That is, a device driver converts the general input/output instruction of the Operation system to messages that the device type can understand.

For example, if XSR orders “read” command to the device, the device cannot receive “read” command of XSR itself. LLD translates “read” command for the device. Thus, the device receives “read” command not by XSR but by LLD, and executes it. Therefore, a user must implement LLD suitable for the device when a user ports XSR.

Figure 5-1 shows LLD in XSR system architecture.

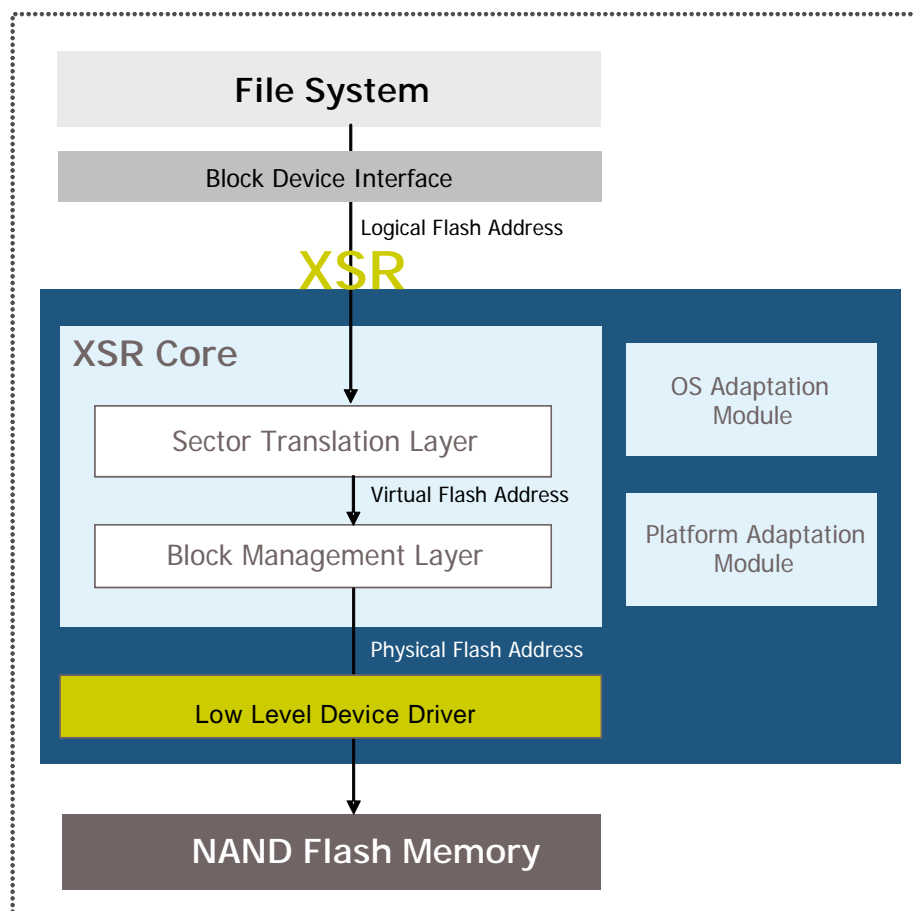


Figure 5-1. LLD in XSR System Architecture

LLD has 17 functions that are classified into 6 categories as follows.

- ☐ **Initialization functions** : initialization, open, and close
- ☐ **Device information query function** : device information return
- ☐ **Flash operation function** : read, write, and erase
- ☐ **Copyback function** : copyback
- ☐ **Other functions** : initial bad block check, writable area setting
- ☐ **Deferred Check Operation functions**

In the next chapter, LLD APIs are covered in detail.

5.2. API

This chapter describes LLD APIs.

Note

In Table 5-1, a user can rename XXX in the function name. It is recommended to name XXX with the device name and three or four capital letters.

Table 5-1 shows the lists of LLD APIs.

The right row in table shows that the function is **Mandatory** or **Optional** or **Recommended**. Optional functions should be existed, but contents of the functions does not need to be implemented.

Table 5-1. LLD API

Function	Description	
XXX_Init	This function initializes the device driver.	M
XXX_Open	This function opens the device and makes it be ready.	M
XXX_Close	This function closes the device and closes linking.	M
XXX_GetDevInfo	This function reports the device information to upper layer.	M
XXX_Read	This function reads data from a page of NAND flash memory within page boundary.	M
XXX_Write	This function writes data into a page of NAND flash memory page boundary.	M
XXX_Erase	This function erases a block of NAND flash memory.	M
XXX_MRead	This function reads data from a page of NAND flash memory within block boundary.	M
XXX_MWrite	This function writes data into a page of NAND flash memory within block boundary.	M
XXX_MErase	This function erases multiple blocks of NAND flash memory	M
XXX_EraseVerify	This function verifies an erase operation whether success or not	M
XXX_CopyBack	This function copies data by using internal buffer in the device.	M
XXX_ChkInitBadBlk	This function checks whether a block is an initial bad block or not.	M
XXX_SetRWArea	This function is called when NAND device provides write/erase protection functionality in hardware.	O
XXX_IOCTL	This function is called to extend LLD functionality.	O
XXX_GetPrevOpData	This function copies data of previous write operation to the given buffer.	O
XXX_FlushOp	This function completes the current working operation.	O

XXX_Init

DESCRIPTION

This function initializes XXX device driver.

SYNTAX

```
INT32
XXX_Init(VOID *pParm)
```

PARAMETERS

Parameter	Type	In/Out	Description
pParm	VOID *	In	Pointer to XsrVolParm data structure

RETURN VALUE

Return Value	Description
LLD_SUCCESS	Initialize Success
LLD_INIT_FAILURE	Initialize Failure

REMARKS

This function is a mandatory initialization function.

For more information about XsrVolParm data structure, refer to the API page of PAM_GetPAParm.

EXAMPLE

(1) Example to call the function

```
#include <XSRTypes.h>
#include <PAM.h>
#include <LLD.h>
#include <XXX.h> /* Header File of Low Level Device Driver */

BOOL32
Example(VOID)
{
    INT32    nErr;
    UINT32    nVol = 0;
    UINT32    nBaseAddr = 0x20000000;
    LowFuncTbl stLFT[MAX_VOL];

    PAM_RegLFT((VOID *)stLFT);
    nErr = stLFT[nVol].Init((VOID *) PAM_GetPAParm());

    if (nErr != LLD_SUCCESS)
    {
        printf("XXX_Init(%d) fail. ErrCode = %x\n", nDev, nErr);
        return(FALSE32);
    }
}
```

```

    return(TRUE32);
}

```

(2) Example to implement the function in OneNAND

```

INT32
ONLD_Init(VOID *pParm)
{
    UINT32      nCnt;
    UINT32      nIdx;
    UINT32      nVol;
    static BOOL32 nInitFlg = FALSE32;
    XsrVolParm  *pstParm = (XsrVolParm*)pParm;

    ONLD_DBG_PRINT((TEXT("[ONLD : IN] ++ONLD_Init()\r\n")));

    if (nInitFlg == TRUE32)
    {
        return(ONLD_SUCCESS);
    }

    for (nCnt = 0; nCnt < MAX_SUPPORT_DEVS; nCnt++)
    {
        pstPrevOpInfo[nCnt]      = (PrevOpInfo *) NULL;
        astONLDDev[nCnt].bOpen    = FALSE32;
        astONLDDev[nCnt].bPower   = FALSE32;
        astONLDDev[nCnt].pstDevSpec = NULL;
    }

    for (nVol = 0; nVol < XSR_MAX_VOL; nVol++)
    {
        for (nIdx = 0; nIdx < (XSR_MAX_DEV / XSR_MAX_VOL); nIdx++)
        {
            nCnt = nVol * (XSR_MAX_DEV / XSR_MAX_VOL) + nIdx;

            if (pstParm[nVol].nBaseAddr[nIdx] != NOT_MAPPED)
                astONLDDev[nCnt].BaseAddr
                    = OAM_Pa2Va(pstParm[nVol].nBaseAddr[nIdx]);
            else
                astONLDDev[nCnt].BaseAddr = NOT_MAPPED;
        }
    }

    nInitFlg = TRUE32;

    ONLD_DBG_PRINT((TEXT("[ONLD : OUT] --ONLD_Init()\r\n")));

    return(ONLD_SUCCESS);
}

```

SEE ALSO

XXX_Open, XXX_Close

XXX_Open

DESCRIPTION

This function opens the device and makes it be ready.

SYNTAX

```
INT32
XXX_Open(UINT32 nDev)
```

PARAMETERS

Parameter	Type	In/Out	Description
nDev	UINT32	In	Physical Device Number (0 ~ 7)

RETURN VALUE

Return Value	Description
LLD_SUCCESS	Open Success
LLD_OPEN_FAILURE	Open Failure

REMARKS

This function is a mandatory initialization function.

EXAMPLE

(1) Example to call the function

```
#include <XSRTypes.h>
#include <PAM.h>
#include <LLD.h>
#include <XXX.h> /* Header File of Low Level Device Driver */

BOOL32
Example(VOID)
{
    INT32    nErr;
    UINT32   nDev = 0;
    UINT32   nVol = 0;
    LowFuncTbl stLFT[MAX_VOL];

    PAM_RegLFT((VOID *)stLFT);

    nErr = stLFT[nVol].Open(nDev);

    if (nErr != LLD_SUCCESS)
    {
        printf("XXX_Open(%d) fail. ErrCode = %x\n", nDev, nErr);
        return (FALSE32);
    }
    return (TRUE32);
}
```

(2) Example to implement the function in OneNAND

```

INT32
ONLD_Open(UINT32 nDev)
{
    INT32    nCnt;
    UINT16   nMID, nDID, nVID;
    UINT32    nBAddr;
    LLDMEArg  *pstMEArg;

    SET_PWR_FLAG(nDev);

    _GetDevID(nDev, &nMID, &nDID, &nVID);

    for (nCnt = 0; astNandSpec[nCnt].nMID != 0; nCnt++)
    {
        if (nDID == astNandSpec[nCnt].nDID)
        {
            if ((nVID >> 8) == astNandSpec[nCnt].nGEN)
            {
                astONLDDDev[nDev].bOpen      = TRUE32;
                astONLDDDev[nDev].pstDevSpec = &astNandSpec[nCnt];
                break;
            }
            else
            {
                continue;
            }
        }
    }

    _InitDevInfo(nDev, &nDID, &nVID);

    if (astONLDDDev[nDev].bOpen != TRUE32)
    {
        CLEAR_PWR_FLAG(nDev);
        return (ONLD_OPEN_FAILURE);
    }

    if (pstPrevOpInfo[nDev] == (PrevOpInfo*)NULL)
    {
        pstPrevOpInfo[nDev]
            = (PrevOpInfo*)MALLOC(sizeof(PrevOpInfo));

        if (pstPrevOpInfo[nDev] == (PrevOpInfo*)NULL)
        {
            return (ONLD_OPEN_FAILURE);
        }
    }

    pstPrevOpInfo[nDev]->ePreOp      = NONE;
    pstPrevOpInfo[nDev]->nCmd         = 0;
    pstPrevOpInfo[nDev]->nPsn        = 0;
    pstPrevOpInfo[nDev]->nScts       = 0;
    pstPrevOpInfo[nDev]->nFlag       = 0;
    pstPrevOpInfo[nDev]->pstPreMEArg
        = (LLDMEArg*)MALLOC(sizeof(LLDMEArg));

```



```

pstMEArg = pstPrevOpInfo[nDev]->pstPreMEArg;
if (pstMEArg == NULL)
{
    return (ONLD_OPEN_FAILURE);
}

memset(pstMEArg, 0x00, sizeof(LLDMEArg));
pstMEArg->pstMEList
    = (LLDMEList*)MALLOC(sizeof(LLDMEList) * XSR_MAX_MEBLK);
if (pstMEArg->pstMEList == NULL)
{
    return (ONLD_OPEN_FAILURE);
}

pstMEArg->nNumOfMList = (UINT16)0x00;
pstMEArg->nBitMapErr  = (UINT16)0x00;
pstMEArg->bFlag        = FALSE32;

pstPrevOpInfo[nDev]->nBufSel = DATA_BUF0;

nBAddr = GET_DEV_BADDR(nDev);

/* IO Buffer Enable for NAND interrupt enable/disable */
ONLD_REG_SYS_CONF1(nBAddr) &= MASK_SYNC_BURST_READ;
ONLD_REG_SYS_CONF1(nBAddr) |=
    (IOBE_ENABLE | BST_RD_LATENCY_4 | BST_LENGTH_8WD);

ONLD_REG_SYS_CONF1(nBAddr) |= (SYNC_READ_MODE);

LLD_RTL_PRINT((TEXT("[ONLD:MSG]   nNumOfBlks   = %d\r\n"),
astONLDDev[nDev].pstDevSpec->nNumOfBlks));
LLD_RTL_PRINT((TEXT("[ONLD:MSG]   nNumOfPlanes = %d\r\n"),
astONLDDev[nDev].pstDevSpec->nNumOfPlanes));
LLD_RTL_PRINT((TEXT("[ONLD:MSG]   nBlksInRsv   = %d\r\n"),
astONLDDev[nDev].pstDevSpec->nBlksInRsv));
LLD_RTL_PRINT((TEXT("[ONLD:MSG]   nBadPos      = %d\r\n"),
astONLDDev[nDev].pstDevSpec->nBadPos));
LLD_RTL_PRINT((TEXT("[ONLD:MSG]   nLsnPos      = %d\r\n"),
astONLDDev[nDev].pstDevSpec->nLsnPos));
LLD_RTL_PRINT((TEXT("[ONLD:MSG]   nECCPos      = %d\r\n"),
astONLDDev[nDev].pstDevSpec->nEccPos));
LLD_RTL_PRINT((TEXT("[ONLD:MSG]   nBWidth     = %d\r\n"),
astONLDDev[nDev].pstDevSpec->nBWidth));
LLD_RTL_PRINT((TEXT("[ONLD:MSG]   nMEFlag     = %d\r\n"),
astONLDDev[nDev].pstDevSpec->nMEFlag));
LLD_RTL_PRINT((TEXT("[ONLD:MSG]   nLKFlag     = %d\r\n"),
astONLDDev[nDev].pstDevSpec->nLKFlag));

return (ONLD_SUCCESS);
}

```

SEE ALSO

XXX_Init, XXX_Close

XXX_Close

DESCRIPTION

This function closes XXX device and closes linking.

SYNTAX

```
INT32
XXX_Close(UINT32 nDev)
```

PARAMETERS

Parameter	Type	In/Out	Description
nDev	UINT32	In	Physical Device Number (0 ~ 7)

RETURN VALUE

Return Value	Description
LLD_SUCCESS	Close Success
LLD_CLOSE_FAILURE	Close Failure

REMARKS

This function is a mandatory initialization function.

EXAMPLE

(1) Example to call the function

```
#include <XSRTypes.h>
#include <PAM.h>
#include <LLD.h>
#include <XXX.h> /* Header File of Low Level Device Driver */

BOOL32
Example(VOID)
{
    INT32    nErr;
    UINT32   nDev = 0;
    UINT32   nVol = 0;
    LowFuncTbl stLFT[MAX_VOL];

    PAM_RegLFT((VOID *)stLFT);

    nErr = stLFT[nVol].Close(nDev);

    if (nErr != LLD_SUCCESS)
    {
        printf("XXX_Close(%d) fail. ErrCode = %x\n", nDev, nErr);
        return (FALSE32);
    }
    return (TRUE32);
}
```

(2) Example to implement the function in OneNAND

```

INT32
ONLD_Close(UINT32 nDev)
{
    UINT32  nBAddr;
    INT32   nRes = ONLD_SUCCESS;

    nBAddr = GET_DEV_BADDR(nDev);

    if ((nRes =ONLD_FlushOp(nDev)) != ONLD_SUCCESS)
    {
        return nRes;
    }

    if (pstPrevOpInfo[nDev] != (PrevOpInfo*)NULL)
    {
        FREE(pstPrevOpInfo[nDev]->pstPreMEArg->pstMEList);
        pstPrevOpInfo[nDev]->pstPreMEArg->pstMEList
            = (LLDMEList*)NULL;
        FREE(pstPrevOpInfo[nDev]->pstPreMEArg);
        pstPrevOpInfo[nDev]->pstPreMEArg = (LLDMEArg*)NULL;

        FREE(pstPrevOpInfo[nDev]);
        pstPrevOpInfo[nDev] = (PrevOpInfo*)NULL;
    }

    astONLDDev[nDev].bOpen      = FALSE32;
    astONLDDev[nDev].bPower     = FALSE32;
    astONLDDev[nDev].pstDevSpec = NULL;

    return (ONLD_SUCCESS);
}

```

SEE ALSO

XXX_Init, XXX_Open

XXX_GetDevInfo

DESCRIPTION

This function reports the device information to upper layer.

SYNTAX

```
INT32
XXX_GetDevInfo(UINT32 nDev, LLDSpec* pstDevInfo)
```

PARAMETERS

Parameter	Type	In/Out	Description
nDev	UINT32	In	Physical Device Number (0 ~ 7)
pstDevInfo	LLDSpec *	Out	Data structure of device information

RETURN VALUE

Return Value	Description
LLD_SUCCESS	Sending the requested information
LLD_ILLEGAL_ACCESS	Illegal Access

REMARKS

This function is a mandatory device information query function.

LLDSpec data structure is declared in LLD.h.

□ **LLDSpec** data structure

```
typedef struct {
    /* offset size */
    UINT8  nMCode;      /* Manufacturer Code 0x00 1 */
    UINT8  nDCode;      /* Device Code 0x01 1 */
    UINT16 nNumOfBlks;   /* The Number of Blocks 0x02 2 */
    UINT16 nPgsPerBlk;   /* The Number of Pages per Block
                        0x04 2 */
    UINT16 nBlksInRsv;   /* The Number of Blocks 0x06 2
                        in Reserved Block Pool */
    UINT8  nSctsPerPg;   /* The Number of Sectors per Page
                        0x08 1 */
    UINT8  nNumOfPlane; /* The Number of Plane 0x09 1 */
    UINT8  nMEFlag;      /* Multi-block Erase Flag 0x0A 1 */

    UINT8  nBWidth;      /* Device Bus Width 0x0C 1 */
    UINT8  nBadPos;      /* BadBlock Information Position
                        0x0E 1 */
    UINT8  nLsnPos;      /* LSN Position 0x0E 1 */
    UINT8  nEccPos;      /* ECC Value Position 0x0F 1 */

    UINT8  aUID[XSR_UID_SIZE]; /* UID 0x10 16,
```

```

                                0xFF(absence case of UID)  */
UINT16  nTrTime; /* Physical Read Operation Time  0x20  2 */
UINT16  nTwTime; /* Physical Write Operation Time  0x22  2 */
UINT16  nTeTime; /* Physical Erase Operation Time  0x24  2 */
UINT16  nTfTime; /* Data Transfer Time            0x26  2 */
} LLDSpec;

```

Table 5-2 explains LLDSpec data structure.

Table 5-2. LLDSpec in LLD.h

Member Variable	Member Variable
nMCode	Manufacturer code of a device
nDCode	Device code
nNumOfBlks	Total number of blocks of a device
nPgsPerBlk	The number of pages per one block of a device
nBlksInRsv	The number of blocks in reserved block pool
nSctsPerPg	The number of sectors per one page of device
nNumOfPlane	The number of planes of a device
nMEFlag	Flag for multi-block erase
nBWidth	Device bus width
nBadPos	Bad block information position
nLsnPos	Position to store LSN
nECCPos	Position to store ECC value
aUID [XSR_UID_SIZE]	Unique ID of device
nTrTime	Physical read operation time of a device (nsec)
nTwTime	Physical write operation time of a device (nsec)
nTeTime	Physical erase operation time of a device (nsec)
nTfTime	Data transfer time of a device (nsec)

Here is more detailed explanation about LLDSpec.

□ nBlksInRsv

In general, NAND flash memory can contain bad block. nBlksInRsv is the number of the reserved block in NAND flash memory. The related information about the number of the reserved block is described in “VALID BLOCK” chapter inside data sheet of flash memory. In VALID BLOCK chapter, the maximum valid block number and the minimum valid block number is marked. nBlksInRsv is the remainder between the maximum valid block number and the minimum valid block number.

□ nNumOfPlane

nNumOfPlane is the number of planes in NAND flash memory. NAND flash memory can consists of one or more planes internally.

□ nMEFlag

nMEFlag means whether NAND flash memory support multi-block erase functionality or not. If NAND flash memory supports multi-block erase, nMEFlag can have LLD_ME_OK or LLD_ME_NO. However, NAND flash memory does not support multi-block erase, nMEFlag must have LLD_ME_NO. (For more information about which device supports multi-block erase, refer to the Specification of NAND flash memory.)

□ nBWidth

nBWidth is the data bus width in NAND flash memory. The data bus of NAND flash memory can be 8bit or 16bit. If the data bus is 8bit, nBWidth becomes LLD_BW_X08. If the data bus is 16bit, nBWidth becomes LLD_BW_X16.

This value is used at software ECC module in BML. If this value is abnormal, the return value of software ECC can be abnormal.

□ nBadPos

To show a bad block, the blocks of NAND flash memory records the specific value at the specific position of spare array. nBadPos is offset of the bad block position of spare array. nBadPos depends on the kind of NAND flash memory. A user can know nBadPos in the data sheet of NAND flash memory or “Samsung NAND flash Spare Area Assignment Standard (21.Feb.2003)”.

□ nLsnPos

nLsnPos is offset that the first byte of LSN(Logical Sector Number) of spare array is located.

nLsnPos depends on the kind of NAND flash memory. A user can know nLsnPos in the data sheet of NAND flash memory or “Samsung NAND flash Spare Area Assignment Standard (21.Feb.2003)”.

□ nEccPos

nEccPos is offset that the first byte of ECC of spare array is located.

nEccPos depends on the kind of NAND flash memory. A user can know nEccPos in the data sheet of NAND flash memory or “Samsung NAND flash Spare Area Assignment Standard (21.Feb.2003)”.

EXAMPLE

(1) Example to call the function

```
#include <XSRTypes.h>
#include <PAM.h>
#include <LLD.h>
#include <XXX.h> /* Header File of Low Level Device Driver */

BOOL32
Example(VOID)
{
    INT32    nErr;
    UINT32    nDev = 0;
    UINT32    nVol = 0;
    LLDSpec    stDevInfo;
    LowFuncTbl stLFT[MAX_VOL];

    PAM_RegLFT((VOID *)stLFT);

    nErr = stLFT[nVol].GetDevInfo(nDev, &stDevInfo);

    if (nErr != LLD_SUCCESS)
    {
        printf("XXX_ GetDevInfo()fail. ErrCode = %x\n", nErr);
        return (FALSE32);
    }
    return (TRUE32);
}
```

```
}
```

(2) Example to implement the function in OneNAND

```
INT32
ONLD_GetDevInfo(UINT32 nDev, LLDSpec *pstFILDev)
{
    UINT32 nBAddr;

    if (astONLDDev[nDev].pstDevSpec == NULL)
    {
        return(ONLD_ILLEGAL_ACCESS);
    }

    nBAddr = GET_DEV_BADDR(nDev);

    pstFILDev->nMCode
        = (UINT8)astONLDDev[nDev].pstDevSpec->nMID;
    pstFILDev->nDCode
        = (UINT8)astONLDDev[nDev].pstDevSpec->nDID;
    pstFILDev->nNumOfBlks
        = (UINT16)astONLDDev[nDev].pstDevSpec->nNumOfBlks;
    pstFILDev->nPgsPerBlk = (UINT16)GET_PGS_PER_BLK(nDev);
    pstFILDev->nBlksInRsv
        = (UINT16)astONLDDev[nDev].pstDevSpec->nBlksInRsv;

    pstFILDev->nSctsPerPg = (UINT8)GET_SCTS_PER_PG(nDev);
    pstFILDev->nNumOfPlane
        = (UINT8)astONLDDev[nDev].pstDevSpec->nNumOfPlanes;

    pstFILDev->nBadPos
        = (UINT8)astONLDDev[nDev].pstDevSpec->nBadPos;
    pstFILDev->nLsnPos
        = (UINT8)astONLDDev[nDev].pstDevSpec->nLsnPos;
    pstFILDev->nEccPos
        = (UINT8)astONLDDev[nDev].pstDevSpec->nEccPos;
    pstFILDev->nBWidth
        = (UINT8)astONLDDev[nDev].pstDevSpec->nBWidth;
    pstFILDev->nMEFlag
        = (UINT8)astONLDDev[nDev].pstDevSpec->nMEFlag;

    pstFILDev->aUID[0]    = 0xEC;
    pstFILDev->aUID[1]    = 0x00;
    pstFILDev->aUID[2]    = 0xC5;
    pstFILDev->aUID[3]    = 0x5F;
    pstFILDev->aUID[4]    = 0x3D;
    pstFILDev->aUID[5]    = 0x12;
    pstFILDev->aUID[6]    = 0x34;
    pstFILDev->aUID[7]    = 0x37;
    pstFILDev->aUID[8]    = 0x34;
    pstFILDev->aUID[9]    = 0x33;
    pstFILDev->aUID[10]   = 0x30;
    pstFILDev->aUID[11]   = 0x30;
    pstFILDev->aUID[12]   = 0x03;
    pstFILDev->aUID[13]   = 0x03;
    pstFILDev->aUID[14]   = 0x06;
    pstFILDev->aUID[15]   = 0x02;
```



```
pstFILDev->nTrTime
    = (UINT32)astONLDDev[nDev].pstDevSpec->nTrTime;
pstFILDev->nTwTime
    = (UINT32)astONLDDev[nDev].pstDevSpec->nTwTime;
pstFILDev->nTeTime
    = (UINT32)astONLDDev[nDev].pstDevSpec->nTeTime;
pstFILDev->nTfTime
    = (UINT32)astONLDDev[nDev].pstDevSpec->nTfTime;

return (ONLD_SUCCESS);
}
```

SEE ALSO

XXX_Read

DESCRIPTION

This function reads data from NAND flash memory. XXX_Read, unlikely XXX_MRead, only read sectors within a page boundry. The sector number of the large block and small block are different.

SYNTAX

```
INT32
XXX_Read(UINT32 nDev, UINT32 nPsn, UINT32 nNumOfScts, UINT8
*pMBuf, UINT8 *pSBuf, UINT32 nFlag)
```

PARAMETERS

Parameter	Type	In/Out	Description
nDev	UINT32	In	Physical Device Number (0 ~ 7)
nPsn	UINT32	In	Physical Sector Number
nNumOfScts	UINT32	In	Number of sectors
pMBuf	UINT8 *	Out	Memory buffer for main array of NAND flash memory
pSBuf	UINT8 *	Out	Memory buffer for spare array of NAND flash memory
nFlag	UINT32	In	Operation options (ECC on/off)

nFlag has the the operation options as follows .

Flag	Value	Description
LLD_FLAG_ECC_ON	(1 << 1)	ECC on (Read Operation with ECC)
LLD_FLAG_ECC_OFF	(0 << 1)	ECC off (Read Operation without ECC)

RETURN VALUE

Return Value	Description
LLD_SUCCESS	Read Success
LLD_READ_ERROR ECC result code	Data Integrity Fault (1 or 2bit ECC error) This return value is available only in case of using Hardware ECC.
LLD_PREV_WRITE_ERROR LLD_PREV_OP_RESULT	In DCOP (Deferred Check Operation), previous write error happens.
LLD_PREV_ERASE_ERROR LLD_PREV_OP_RESULT	In DCOP (Deferred Check Operation), previous erase error happens.
LLD_ILLEGAL_ACCESS	Illegal Read Operation
LLD_READ_DISTURBANCE	1bit ECC error by read disturbance happens.

REMARKS

This function is a mandatory flash operation function.

DCOP (Deferred Check Operation) : a method optimizing the operation performance. The DCOP method is a procedure to terminate the function (write or erase) right after the command issue. Then it checks the result of the operation at next function call.



The parameters `nPsn` and `nNumOfScts` read data as a unit of a sector.
For more information about the byte alignment, refer to 7.5 Byte Alignment Restrictions.

EXAMPLE

(1) Example to call the function

```
#include <XSRTypes.h>
#include <PAM.h>
#include <LLD.h>
#include <XXX.h> /* Header File of Low Level Device Driver */

BOOL32
Example(VOID)
{
    INT32    nErr;
    UINT32   nDev = 0;
    UINT32   nVol = 0;
    UINT32   nPSN = 0;
    UINT32   nNumOfScts = 1;
    UINT8    aMBuf[LLD_MAIN_SIZE];
    UINT8    aSBuf[LLD_SPARE_SIZE];
    UINT32   nFlag = LLD_FLAG_ECC_ON;
    LowFuncTbl stLFT[MAX_VOL];

    PAM_RegLFT((VOID *)stLFT);

    nErr = stLFT[nVol].Read(nDev, nPSN, nNumOfScts, aMBuf,
                          aSBuf, nFlag);

    if (nErr != LLD_SUCCESS)
    {
        printf("XXX_Read() fail. ErrCode = %x\n", nErr);
        return (FALSE32);
    }
    return (TRUE32);
}
```

(2) Example to implement the function in OneNAND

```
INT32
ONLD_Read(UINT32 nDev, UINT32 nPsn, UINT32 nScts,
          UINT8 *pMBuf, UINT8 *pSBuf, UINT32 nFlag)
{
    UINT32   nCnt;
    UINT16   nBSA;
    UINT32   *pONLDMBuf;
    UINT32   *pONLDSBuf;
    UINT32   nBAddr;
    UINT16   nPbn;
    UINT32   nPpn;
    UINT16   nEccRes    = (UINT16)0x0;
    UINT16   nEccMask   = (UINT16)0x0;
    UINT32   nPsnO      = nPsn;
    UINT32   nSctsO     = nScts;
    INT32    nRes;
```

```

nBAddr = GET_DEV_BADDR(nDev);

if ((pMBuf == NULL) && (pSBuf == NULL))
{
    return (ONLD_ILLEGAL_ACCESS);
}

nPbn = GET_PBN(nDev, nPsn);
nPpn = GET_PPN(nDev, nPsn);

/* ECC Value Bit Mask Setting */
for (nCnt = 0; nCnt < nScts; nCnt++)
{
    if (pMBuf != NULL)
    {
        nEccMask |= (0x08 << (nCnt * 4));
    }
    if (pSBuf != NULL)
    {
        nEccMask |= (0x02 << (nCnt * 4));
    }
}

if ((nRes = _ChkPrevOpResult(nDev)) != ONLD_SUCCESS)
{
    pstPrevOpInfo[nDev]->ePreOp = NONE;
    return nRes;
}

ONLD_REG_START_ADDR2(nBAddr) = (UINT16)MAKE_DBS(nPbn);

/* Block Number Set */
ONLD_REG_START_ADDR1(nBAddr) = (UINT16)MAKE_FBA(nPbn);

/* Sector Number Set */
ONLD_REG_START_ADDR8(nBAddr)
    = (UINT16)(
        ((nPsn << astONLDInfo[nDev].nFPASelSft) & MASK_FPA)
        | (nPsn & astONLDInfo[nDev].nFSAMask));

nBSA = (UINT16)MAKE_BSA(nPsn, GET_NXT_BUF_SEL(nDev));

/* Start Buffer Selection */
ONLD_REG_START_BUF(nBAddr)
    = (UINT16)((nBSA & MASK_BSA) | (nScts & MASK_BSC));

if (nFlag & ONLD_FLAG_ECC_ON)
{
    /* System Configuration Reg set (ECC On)*/
    ONLD_DBG_PRINT((TEXT("[ONLD : MSG] ECC ON\r\n")));
    ONLD_REG_SYS_CONF1(nBAddr) &= CONF1_ECC_ON;
}
else
{
    /* System Configuration Reg set (ECC Off)*/
    ONLD_DBG_PRINT((TEXT("[ONLD : MSG] ECC OFF\r\n")));
}

```

```

        ONLD_REG_SYS_CONF1(nBAddr) |= CONF1_ECC_OFF;
    }

    /* INT Stat Reg Clear */
    ONLD_REG_INT(nBAddr)          = (UINT16)INT_CLEAR;

    /*-----*/
    /*ONLD Read CMD is issued          */
    /*-----*/
    if (pMBuf != NULL)
    {
        /* Page Read Command issue */
        ONLD_DBG_PRINT((TEXT("[ONLD : MSG]  Read Page\r\n")));
        ONLD_REG_CMD(nBAddr)
            = (UINT16)ONLD_CMD_READ_PAGE;
    }
    else
    {
        /* Spare Read Command issue */
        ONLD_DBG_PRINT((TEXT("[ONLD : MSG]  Read Spare\r\n")));
        ONLD_REG_CMD(nBAddr)
            = (UINT16)ONLD_CMD_READ_SPARE;
    }

    while (GET_ONLD_INT_STAT(nBAddr, PEND_READ)
           != (UINT16)PEND_READ)
    {
        /* Wait until device ready */
    }

    pONLDMBuf = (UINT32*)GET_ONLD_MBUF_ADDR(
        nBAddr, nPsnO, GET_NXT_BUF_SEL(nDev));
    pONLDSBuf = (UINT32*)GET_ONLD_SBUF_ADDR(nBAddr,
        nPsnO, GET_NXT_BUF_SEL(nDev));

    if (pMBuf != NULL)
    {
        /* Memcopy for main data */
        _ReadMain(pMBuf, pONLDMBuf, nSctsO);
    }
    if (pSBuf != NULL)
    {
        /* Memcopy for spare data */
        _ReadSpare(pSBuf, pONLDSBuf, nSctsO);
    }

    pstPrevOpInfo[nDev]->nBufSel    = GET_NXT_BUF_SEL(nDev);

    if (nFlag & ONLD_FLAG_ECC_ON)
    {
        nEccRes = (ONLD_REG_ECC_STAT(nBAddr) & nEccMask);

        if (nEccRes != 0)
        {

```



```
        return (ONLD_READ_ERROR | nEccRes);  
    }  
}  
  
return (ONLD_SUCCESS);  
}
```

SEE ALSO

XXX_Write, XXX_Erase, XXX_Copyback, XXX_MRead, XXX_MWrite,
XXX_MErase, XXX_EraseVerify

XXX_Write

DESCRIPTION

This function writes data into NAND flash memory. XXX_Write, unlikely XXX_MWrite, only write sectors within a page boundry. The sector number of the large block and small block are different.

SYNTAX

```
INT32
XXX_Write(UINT32 nDev, UINT32 nPsn, UINT32 nNumOfScts, UINT8
*pMBuf, UINT8 *pSBuf, UINT32 nFlag)
```

PARAMETERS

Parameter	Type	In/Out	Description
nDev	UINT32	In	Physical Device Number (0 ~ 7)
nPsn	UINT32	In	Physical Sector Number
nNumOfScts	UINT32	In	Number of sectors
pMBuf	UINT8 *	In	Memory buffer for main array of NAND flash memory
pSBuf	UINT8 *	In	Memory buffer for spare array of NAND flash memory
nFlag	UINT32	In	Operation options (ECC on/off, Sync/Async)

nFlag has the the operation options as follows .

Flag	Value	Description
LLD_FLAG_ASYNC_OP	(1 << 0)	Asynchronous Operation
LLD_FLAG_SYNC_OP	(0 << 0)	Synchronous Operation
LLD_FLAG_ECC_ON	(1 << 1)	ECC on (Read Operation with ECC)
LLD_FLAG_ECC_OFF	(0 << 1)	ECC off (Read Operation without ECC)

Flag value is divided into Sync/Async Operation flag and ECC on/off flag. These two flags can be merged as follow.

```
nFlag = LLD_FLAG_SYNC_OP | LLD_FLAG_ECC_ON;
```

RETURN VALUE

Return Value	Description
LLD_SUCCESS	Write Success
LLD_WRITE_ERROR	Write Failure
LLD_PREV_WRITE_ERROR LLD_PREV_OP_RESULT	In DCOP (Deferred Check Operation), previous write error happens
LLD_PREV_ERASE_ERROR LLD_PREV_OP_RESULT	In DCOP (Deferred Check Operation), previous erase error happens
LLD_WR_PROTECT_ERROR	Write Operation at Locked Area
LLD_ILLEGAL_ACCESS	Illegal Write Operation

REMARKS



This function is a mandatory flash operation function.

DCOP(Deferred Check Operation) : a method optimizing the operation performance. The DCOP method is a procedure to terminate the function (write or erase) right after the command issue. Then it checks the result of the operation at next function call.

The parameters `nPsn` and `nNumOfScts` write data as a unit of a sector.

For more information about the byte alignment, refer to Chapter 7.5 Byte Alignment Restrictions.

In order to support asynchronous mode, codes which execute next steps must be added.

First, check the value of a flag. And then, clear an interrupt. Finally, enable the interrupt.

For more information about the interrupt, refer to Chapter 7.2 Interrupt.

EXAMPLE

(1) Example to call the function

```
#include <XSRTypes.h>
#include <PAM.h>
#include <LLD.h>
#include <XXX.h> /* Header File of Low Level Device Driver */

BOOL32
Example(VOID)
{
    INT32    nErr;
    UINT32   nDev = 0;
    UINT32   nVol = 0;
    UINT32   nPSN = 0;
    UINT32   nNumOfScts = 4;
    UINT8    aMBuf[LLD_MAIN_SIZE * 4];
    UINT8    aSBuf[LLD_SPARE_SIZE * 4];
    UINT32   nFlag = LLD_FLAG_ASYNC_OP | LLD_FLAG_ECC_ON;
    LowFuncTbl stLFT[MAX_VOL];

    PAM_RegLFT((VOID *)stLFT);

    nErr = stLFT[nVol].Write(nDev, nPSN, nNumOfScts, aMBuf,
                           aSBuf, nFlag);

    if (nErr != LLD_SUCCESS)
    {
        printf("XXX_Write() fail. ErrCode = %x\n", nErr);
        return (FALSE32);
    }
    return (TRUE32);
}
```

(2) Example to implement the function in OneNAND

```
INT32
ONLD_Write(UINT32 nDev, UINT32 nPsn, UINT32 nScts, UINT8 *pMBuf,
           UINT8 *pSBuf,
           UINT32 nFlag)
{
    UINT16   nBSA;
```

```

UINT32 *pONLDMBuf, *pONLDSBuf;
UINT32  nBAddr;
UINT32  nPbn;
INT32   nRes;

nBAddr = GET_DEV_BADDR(nDev);

if ((pMBuf == NULL) && (pSBuf == NULL))
{
    return (ONLD_ILLEGAL_ACCESS);
}

nPbn = GET_PBN(nDev, nPsn);

if ((GET_DEV_DID(nDev) & 0x0008) == DDP_CHIP)
{
    if ((nRes = _ChkPrevOpResult(nDev)) != ONLD_SUCCESS)
    {
        return nRes;
    }
}
/* Device BufferRam Select */
ONLD_REG_START_ADDR2(nBAddr) = (UINT16)MAKE_DBS(nPbn);

pONLDMBuf = (UINT32*)GET_ONLD_MBUF_ADDR(
                nBAddr, nPsn, GET_NXT_BUF_SEL(nDev));
pONLDSBuf = (UINT32*)GET_ONLD_SBUF_ADDR(
                nBAddr, nPsn, GET_NXT_BUF_SEL(nDev));

if (pMBuf != NULL)
{
    /* Memcopy for main data */
    _WriteMain(pMBuf, pONLDMBuf, nScts);
}

if (pSBuf != NULL)
{
    /* Memcopy for spare data */
    _WriteSpare(pSBuf, pONLDSBuf, nScts);
}
else
{
    /* Memset for spare data 0xff*/
    MEMSET(pONLDSBuf, 0xFF, (ONLD_SPARE_SIZE * nScts));
}

if ((GET_DEV_DID(nDev) & 0x0008) != DDP_CHIP)
{
    if ((nRes = _ChkPrevOpResult(nDev)) != ONLD_SUCCESS)
    {
        return nRes;
    }
}

/* Block Number Set */
ONLD_REG_START_ADDR1(nBAddr) = (UINT16)MAKE_FBA(nPbn);

```



```

/* Sector Number Set */
ONLD_REG_START_ADDR8(nBAddr) = (UINT16)(
    ((nPsn << astONLDInfo[nDev].nFPASelSft) & MASK_FPA)
    | (nPsn & astONLDInfo[nDev].nFSAMask));

nBSA = (UINT16)MAKE_BSA(nPsn, GET_NXT_BUF_SEL(nDev));

ONLD_REG_START_BUF(nBAddr) =
    (UINT16)((nBSA & MASK_BSA) | (nScts & MASK_BSC));

if (nFlag & ONLD_FLAG_ECC_ON)
{
    /* System Configuration Reg set (ECC On)*/
    ONLD_DBG_PRINT((TEXT("[ONLD : MSG] ECC ON\r\n")));
    ONLD_REG_SYS_CONFL(nBAddr) &= CONFL_ECC_ON;
}
else
{
    /* System Configuration Reg set (ECC Off)*/
    ONLD_DBG_PRINT((TEXT("[ONLD : MSG] ECC OFF\r\n")));
    ONLD_REG_SYS_CONFL(nBAddr) |= CONFL_ECC_OFF;
}

/* in case of async mode, interrupt should be enabled */
if (nFlag & ONLD_FLAG_ASYNC_OP)
{
    PAM_ClearInt((UINT32)XSR_INT_ID_NAND_0);
    PAM_EnableInt((UINT32)XSR_INT_ID_NAND_0);
}

/* INT Stat Reg Clear */
ONLD_REG_INT(nBAddr) = (UINT16)INT_CLEAR;

/*-----*/
/* ONLD Write CMD is issued */
/*-----*/
if (pMBuf != NULL)
{
    /* Main Write Command issue */
    ONLD_DBG_PRINT((TEXT("[ONLD : MSG] Write Page\r\n")));
    ONLD_REG_CMD(nBAddr) = (UINT16)ONLD_CMD_WRITE_PAGE;
}
else
{
    /* Spare Write Command issue */
    ONLD_DBG_PRINT((TEXT("[ONLD : MSG] Write Spare\r\n")));
    ONLD_REG_CMD(nBAddr) = (UINT16)ONLD_CMD_WRITE_SPARE;
}
pstPrevOpInfo[nDev]->ePreOp = WRITE;
pstPrevOpInfo[nDev]->nPsn = nPsn;
pstPrevOpInfo[nDev]->nScts = nScts;
pstPrevOpInfo[nDev]->nFlag = nFlag;
pstPrevOpInfo[nDev]->nBufSel = GET_NXT_BUF_SEL(nDev);

return (ONLD_SUCCESS);
}

```

**SEE ALSO**

XXX_Read, XXX_Erase, XXX_Copyback, XXX_MRead, XXX_MWrite,
XXX_MErase, XXX_EraseVerify

XXX_Erase

DESCRIPTION

This function erases a block of NAND flash memory.

SYNTAX

```
INT32
XXX_Erase(UINT32 nDev, UINT32 nPbn, UINT32 nFlag)
```

PARAMETERS

Parameter	Type	In/Out	Description
nDev	UINT32	In	Physical Device Number (0 ~ 7)
nPbn	UINT32	In	Physical Block Number
nFlag	UINT32	In	Operation options (Sync/Async)

nFlag has the the operation options as follows .

Flag	Value	Description
LLD_FLAG_ASYNC_OP	(1 << 0)	Asynchronous Operation
LLD_FLAG_SYNC_OP	(0 << 0)	Synchronous Operation

RETURN VALUE

Return Value	Description
LLD_SUCCESS	Erase Success
LLD_ERASE_ERROR	Erase Failure
LLD_WR_PROTECT_ERROR	Erase Operation Error at Locked Area
LLD_PREV_WRITE_ERROR LLD_PREV_OP_RESULT	In DCOP (Deferred Check Operation), previous write error happens
LLD_PREV_ERASE_ERROR LLD_PREV_OP_RESULT	In DCOP (Deferred Check Operation), previous erase error happens
LLD_ILLEGAL_ACCESS	Illegal Erase Operation

REMARKS

This function is a mandatory flash operation function.

DCOP(Deferred Check Operation) : a method optimizing the operation performance. The DCOP method is a procedure to terminate the function (write or erase) right after the command issue. Then it checks the result of the operation at next function call.

In order to support asynchronous mode, codes which execute next steps must be added. First, check the value of a flag. And then, clear an interrupt. Finally, enable the interrupt. For more information about the interrupt, refer to Chapter 7.2 Interrupt.

EXAMPLE

(1) Example to call the function

```
#include <XSRTypes.h>
#include <PAM.h>
```

```
#include <LLD.h>
#include <XXX.h> /* Header File of Low Level Device Driver */

BOOL32
Example(VOID)
{
    INT32      nErr;
    UINT32     nDev = 0;
    UINT32     nVol = 0;
    UINT32     nPbn = 0;
    UINT32     nFlag = LLD_FLAG_ASYNC_OP;
    LowFuncTbl stLFT[MAX_VOL];

    PAM_RegLFT((VOID *)stLFT);

    nErr = stLFT[nVol].Erase(nDev, nPbn, nFlag)

    if (nErr != LLD_SUCCESS)
    {
        printf("XXX_Erase()fail. ErrCode = %x\n", nErr);
        return (FALSE32);
    }
    return (TRUE32);
}
```

(2) Example to implement the function in OneNAND

```
INT32
ONLD_Erase(UINT32 nDev, UINT32 nPbn, UINT32 nFlag)
{
    UINT32 nBAddr;
    UINT32 nRes;

    nBAddr = GET_DEV_BADDR(nDev);

    if ((nRes = _ChkPrevOpResult(nDev)) != ONLD_SUCCESS)
    {
        return nRes;
    }

    /* Block Number Set */
    ONLD_REG_START_ADDR1(nBAddr) = (UINT16)MAKE_FBA(nPbn);
    ONLD_REG_START_ADDR2(nBAddr) = (UINT16)MAKE_DBS(nPbn);

    /* INT Stat Reg Clear */
    ONLD_REG_INT(nBAddr) = (UINT16)INT_CLEAR;

    /*-----*/
    /* ONLD Erase CMD is issued */
    /*-----*/
    ONLD_REG_CMD(nBAddr) = (UINT16)ONLD_CMD_ERASE_BLK;

    pstPrevOpInfo[nDev]->ePreOp = ERASE;
    pstPrevOpInfo[nDev]->nPsn = nPbn * GET_SCTS_PER_BLK(nDev);
    pstPrevOpInfo[nDev]->nFlag = nFlag;
    /* in case of async mode, interrupt should be enabled */
}
```



```
if (nFlag & ONLD_FLAG_ASYNC_OP)
{
    PAM_ClearInt((UINT32)XSR_INT_ID_NAND_0);
    PAM_EnableInt((UINT32)XSR_INT_ID_NAND_0);
}

return (ONLD_SUCCESS);
}
```

SEE ALSO

XXX_Read, XXX_Write, XXX_Copyback, XXX_MRead, XXX_MWrite,
XXX_MErase, XXX_EraseVerify

XXX_MRead

DESCRIPTION

This function reads data from NAND flash memory. XXX_MRead, unlikely XXX_Read, only read multiple sectors within a block boundary.

When an error occurs, LLD can return ECC error. If 1bit error occurs, this function returns SUCCESS. If 2bit error occurs, this function performs the read operation of the remaining sector and then returns READ ERROR.

SYNTAX

```
INT32
XXX_MRead(UINT32 nDev, UINT32 nPsn, UINT32 nNumOfScts, SGL
*pstSGL, UINT8 *pSBuf, UINT32 nFlag)
```

PARAMETERS

Parameter	Type	In/Out	Description
nDev	UINT32	In	Physical Device Number (0 ~ 7)
nPsn	UINT32	In	Physical Sector Number
nNumOfScts	UINT32	In	Number of sectors
pstSGL	SGL *	Out	Scatter gather list structure for main array of NAND flash memory
pSBuf	UINT8 *	Out	Memory buffer for spare array of NAND flash memory
nFlag	UINT32	In	Operation options (ECC on/off)

nFlag has the the operation options as follows .

Flag	Value	Description
LLD_FLAG_ECC_ON	(1 << 1)	ECC on (Read Operation with ECC)
LLD_FLAG_ECC_OFF	(0 << 1)	ECC off (Read Operation without ECC)

RETURN VALUE

Return Value	Description
LLD_SUCCESS	Read Success
LLD_READ_ERROR ECC result code	Data Integrity Fault (1 or 2bit ECC error) This return value is available only in case of using Hardware ECC
LLD_WRITE_ERROR LLD_PREV_OP_RESULT	In DCOP (Deferred Check Operation), previous write error happens.
LLD_ERASE_ERROR LLD_PREV_OP_RESULT	In DCOP (Deferred Check Operation), previous erase error happens.
LLD_ILLEGAL_ACCESS	Illegal Read Operation
LLD_READ_DISTURBANCE	1bit ECC error by read disturbance happens.

REMARKS

This function is a mandatory flash operation function.

DCOP (Deferred Check Operation) : a method optimizing the operation performance. The



DCOP method is a procedure to terminate the function (write or erase) right after the command issue. Then it checks the result of the operation at next function call.

The parameters `nPsn` and `nNumOfScts` read data as a unit of a sector.

For more information about the byte alignment, refer to 7.5 Byte Alignment Restrictions.

SGL data structure is declared in `XsrTypes.h`.

□ **SGL** data structure

```
typedef struct
{
    UINT8      nElements;
    SGLEntry    stSGLEntry[XSR_MAX_SGL_ENTRIES];
} SGL;
```

SGLEntry data structure is declared in `XsrTypes.h`.

□ **SGLEntry** data structure

```
typedef struct
{
    UINT8*  pBuf;      /* Buffer for data      */
    UINT16  nSectors;
                /* Number of sectors this entry represents */
    UINT8  nFlag;      /* user data or meta data */
} SGLEntry;
```

SGL is abbreviation for Scatter Gather List. To read whole requested data and store it into several buffers within one function call, we use SGL.

EXAMPLE

(1) Example to call the function

```
#include <XSRTypes.h>
#include <PAM.h>
#include <LLD.h>
#include <XXX.h> /* Header File of Low Level Device Driver */

BOOL32
Example(VOID)
{
    INT32      nErr;
    UINT32     nDev = 0;
    UINT32     nVol = 0;
    UINT32     nPSN = 0;
    UINT32     nNumOfScts = 1;
    SGL        stSGL;
    UINT8      aMBuf[LLD_MAIN_SIZE * nNumOfScts];
    UINT8      aSBuf[LLD_SPARE_SIZE * nNumOfScts];
    UINT32     nFlag = LLD_FLAG_ECC_ON;
    LowFuncTbl stLFT[MAX_VOL];

    PAM_RegLFT((VOID *)stLFT);
```

```

stSGL.stSGLEntry[0].pBuf = aMBuf;
stSGL.stSGLEntry[0].nSectors = nNumOfScts;
stSGL.stSGLEntry[0].nFlag = SGL_ENTRY_USER_DATA;

stSGL.nElements = 1;

nErr = stLFT[nVol].MRead(nDev, nPSN, nNumOfScts, &stSGL,
                        aSBuf, nFlag);

if (nErr != LLD_SUCCESS)
{
    printf("XXX_Read() fail. ErrCode = %x\n", nErr);
    return (FALSE32);
}
return (TRUE32);
}

```

(2) Example to implement the function in OneNAND

```

INT32
ONLD_MRead(UINT32 nDev, UINT32 nPsn, UINT32 nScts, SGL *pstSGL,
UINT8 *pSBuf,
        UINT32 nFlag)
{
    UINT32  nCnt;
    UINT16  nBSA;
    UINT32  *pONLDMBuf;
    UINT32  *pONLDSBuf;
    UINT32  nBAddr;
    UINT16  nPbn;
    UINT16  nEccRes    = (UINT16)0x0;
    UINT16  nEccMask   = (UINT16)0x0;
    UINT32  nFlagO     = nFlag;
    UINT32  nReadPsn;
    UINT32  nCurReadPsn;
    UINT32  nReadScts = 0;
    UINT32  nCurReadScts;
    UINT32  nRemainScts;
    UINT32  nRet = ONLD_SUCCESS;
    UINT8   nSGLIdx = 0;
    UINT32  nSctCount = 0;
    UINT8   *pMBuf;
    INT32   nRes;

    if (pstSGL == NULL)
        return (ONLD_ILLEGAL_ACCESS);

    nBAddr = GET_DEV_BADDR(nDev);

    if ((nRes = _ChkPrevOpResult(nDev)) != ONLD_SUCCESS)
    {
        pstPrevOpInfo[nDev]->ePreOp = NONE;
        return nRes;
    }

    pMBuf = pstSGL->stSGLEntry[nSGLIdx].pBuf;

```



```

nRemainScts = nScts;
nReadPsn    = nPsn;
switch (nReadPsn & astONLDInfo[nDev].nSctSelSft)
{
    case 0:
        nReadScts =
            (astONLDInfo[nDev].nSctsPerPG > nRemainScts) ?
            nRemainScts : astONLDInfo[nDev].nSctsPerPG;
        break;
    case 1:
        nReadScts =
            (astONLDInfo[nDev].nSctSelSft > nRemainScts) ?
            nRemainScts : astONLDInfo[nDev].nSctSelSft;
        break;
    case 2:
        nReadScts = (2 > nRemainScts) ?
            nRemainScts : 2;
        break;
    case 3:
        nReadScts = (1 > nRemainScts) ?
            nRemainScts : 1;
        break;
}

{
    nPbn = GET_PBN(nDev, nReadPsn);

    /* Block Number Set */
    ONLD_REG_START_ADDR1(nBAddr) = (UINT16)MAKE_FBA(nPbn);
    ONLD_REG_START_ADDR2(nBAddr) = (UINT16)MAKE_DBS(nPbn);

    /* Sector Number Set */
    ONLD_REG_START_ADDR8(nBAddr) = (UINT16)(
        ((nReadPsn << astONLDInfo[nDev].nFPASelSft) & MASK_FPA)
        | (nReadPsn & astONLDInfo[nDev].nFSAMask));

    nBSA =
        (UINT16)MAKE_BSA(nReadPsn, GET_CUR_BUF_SEL(nDev));

    /* Start Buffer Selection */
    ONLD_REG_START_BUF(nBAddr) =
        (UINT16)((nBSA & MASK_BSA) | (nReadScts & MASK_BSC));

    if (nFlag & ONLD_FLAG_ECC_ON)
    {
        /* System Configuration Reg set (ECC On)*/
        ONLD_DBG_PRINT((TEXT("[ONLD : MSG] ECC ON\r\n")));
        ONLD_REG_SYS_CONF1(nBAddr) &= CONF1_ECC_ON;
    }
    else
    {
        /* System Configuration Reg set (ECC Off)*/
        ONLD_DBG_PRINT((TEXT("[ONLD : MSG] ECC OFF\r\n")));
        ONLD_REG_SYS_CONF1(nBAddr) |= CONF1_ECC_OFF;
    }
}

```

```

/* INT Stat Reg Clear */
ONLD_REG_INT(nBAddr)          = (UINT16)INT_CLEAR;
/* ----- */
/* ONLD Read CMD is issued */
/* ----- */
if (pMBuf != NULL)
{
    /* Page Read Command issue */
    ONLD_REG_CMD(nBAddr) = (UINT16)ONLD_CMD_READ_PAGE;
}
else
{
    /* Spare Read Command issue */
    ONLD_REG_CMD(nBAddr) = (UINT16)ONLD_CMD_READ_SPARE;
}

if (nFlag0 & ONLD_FLAG_ECC_ON)
{
    /* ECC Value Bit Mask Setting */
    for (nCnt = 0; nCnt < nReadScts; nCnt++)
    {
        if (pMBuf != NULL)
        {
            nEccMask |= (0x08 << (nCnt * 4));
        }
        if (pSBuf != NULL)
        {
            nEccMask |= (0x02 << (nCnt * 4));
        }
    }
}

nCurReadPsn  = nReadPsn;
nCurReadScts = nReadScts;

nRemainScts -= nReadScts;
nReadPsn    += nReadScts;
nReadScts   =
    (astONLDInfo[nDev].nSctsPerPG > nRemainScts) ?
    nRemainScts : astONLDInfo[nDev].nSctsPerPG;

while (GET_ONLD_INT_STAT(nBAddr, PEND_READ)
      != (UINT16)PEND_READ)
{
    /* Wait until device ready */
}

if (nFlag0 & ONLD_FLAG_ECC_ON)
{
    /* No Cache Read */
    nEccRes = (ONLD_REG_ECC_STAT(nBAddr) & nEccMask);

    if (nEccRes != 0)
    {

```

```

        nRet = ONLD_READ_ERROR | nEccRes;
    }
}

while (nRemainScts > 0)
{
    nPbn = GET_PBN(nDev, nReadPsn);

    nSctCount += nCurReadScts;

    if (nSctCount == pstSGL->stSGLEntry[nSGLIdx].nSectors)
    {
        nSctCount = 0;
        nSGLIdx++;
    }
    if (pstSGL->stSGLEntry[nSGLIdx].nFlag
        == SGL_ENTRY_META_DATA)
    {
        nReadPsn += pstSGL->stSGLEntry[nSGLIdx].nSectors;
        nReadScts -= pstSGL->stSGLEntry[nSGLIdx].nSectors;
        nRemainScts -= pstSGL->stSGLEntry[nSGLIdx].nSectors;
        nSctCount = 0;
        nSGLIdx++;

        if (GET_SCTS_PER_PG(nDev) == 2)
        {
            nCurReadPsn = nReadPsn;
            nCurReadScts = nReadScts;
            nReadScts =
                (astONLDInfo[nDev].nSctsPerPG > nRemainScts) ?
                nRemainScts : astONLDInfo[nDev].nSctsPerPG;
            continue;
        }
    }

    /* Block Number Set */
    ONLD_REG_START_ADDR1(nBAddr) = (UINT16)MAKE_FBA(nPbn);
    ONLD_REG_START_ADDR2(nBAddr) = (UINT16)MAKE_DBS(nPbn);

    /* Sector Number Set */
    ONLD_REG_START_ADDR8(nBAddr) = (UINT16)(
        ((nReadPsn << astONLDInfo[nDev].nFPASelSft) & MASK_FPA)
        | (nReadPsn & astONLDInfo[nDev].nFSAMask));

    nBSA =
        (UINT16)MAKE_BSA(nReadPsn, GET_NXT_BUF_SEL(nDev));

    /* Start Buffer Selection */
    ONLD_REG_START_BUF(nBAddr) =
        (UINT16)((nBSA & MASK_BSA) | (nReadScts & MASK_BSC));

    /* INT Stat Reg Clear */
    ONLD_REG_INT(nBAddr) = (UINT16)INT_CLEAR;

    /*-----*/

```

```

/* ONLD Read CMD is issued */
/*-----*/
if (pMBuf != NULL)
{
    ONLD_REG_CMD(nBAddr) = (UINT16)ONLD_CMD_READ_PAGE;
}
else
{
    ONLD_REG_CMD(nBAddr) = (UINT16)ONLD_CMD_READ_SPARE;
}

    pONLDMBuf = (UINT32*)GET_ONLD_MBUF_ADDR(
        nBAddr, nCurReadPsn, GET_CUR_BUF_SEL(nDev));
    pONLDSBuf = (UINT32*)GET_ONLD_SBUF_ADDR(
        nBAddr, nCurReadPsn, GET_CUR_BUF_SEL(nDev));

if (pMBuf != NULL)
{
    /* Memcopy for main data */
    _ReadMain(pMBuf, pONLDMBuf, nCurReadScts);
    pMBuf += (ONLD_MAIN_SIZE * nCurReadScts);
}

if (pSBuf != NULL)
{
    /* Memcopy for main data */
    _ReadSpare(pSBuf, pONLDSBuf, nCurReadScts);
    pSBuf += (ONLD_SPARE_SIZE * nCurReadScts);
}

if (nFlag0 & ONLD_FLAG_ECC_ON)
{
    /* ECC Value Bit Mask Setting */
    for (nCnt = 0; nCnt < nReadScts; nCnt++)
    {
        if (pMBuf != NULL)
        {
            nEccMask |= (0x08 << (nCnt * 4));
        }
        if (pSBuf != NULL)
        {
            nEccMask |= (0x02 << (nCnt * 4));
        }
    }
}

pstPrevOpInfo[nDev]->nBufSel = GET_NXT_BUF_SEL(nDev);

nCurReadPsn = nReadPsn;
nCurReadScts = nReadScts;

nRemainScts -= nReadScts;
nReadPsn += nReadScts;
nReadScts =
    (astONLDInfo[nDev].nSctsPerPG > nRemainScts) ?
    nRemainScts : astONLDInfo[nDev].nSctsPerPG;

```

```

while (GET_ONLD_INT_STAT(nBAddr, PEND_READ)
      != (UINT16)PEND_READ)
{
    /* Wait until device ready */
}

if (nFlag0 & ONLD_FLAG_ECC_ON)
{
    /* No Cache Read */
    nEccRes = (ONLD_REG_ECC_STAT(nBAddr) & nEccMask);

    if (nEccRes != 0)
    {
        nRet = ONLD_READ_ERROR | nEccRes;
    }
}

{
    pONLDMBuf = (UINT32*)GET_ONLD_MBUF_ADDR(
        nBAddr, nCurReadPsn, GET_CUR_BUF_SEL(nDev));
    pONLDSBuf = (UINT32*)GET_ONLD_SBUF_ADDR(
        nBAddr, nCurReadPsn, GET_CUR_BUF_SEL(nDev));

    if (pMBuf != NULL)
    {
        /* Memcopy for main data */
        _ReadMain(pMBuf, pONLDMBuf, nCurReadScts);
        pMBuf += (ONLD_MAIN_SIZE * nCurReadScts);
    }

    if (pSBuf != NULL)
    {
        /* Memcopy for spare data */
        _ReadSpare(pSBuf, pONLDSBuf, nCurReadScts);
        pSBuf += (ONLD_SPARE_SIZE * nCurReadScts);
    }

    pstPrevOpInfo[nDev]->nBufSel    =
        GET_NXT_BUF_SEL(nDev);
}

return (nRet);
}

```

SEE ALSO

XXX_Read, XXX_Write, XXX_Erase, XXX_Copyback, XXX_MWrite,
XXX_MErase, XXX_EraseVerify

XXX_MWrite

DESCRIPTION

This function writes data into NAND flash memory. XXX_MWrite, unlikely XXX_Write, only write sectors within a block boundry.

SYNTAX

```
INT32
XXX_MWrite(UINT32 nDev,  UINT32 nPsn,  UINT32 nNumOfScts,
           SGL *pstSGL,  UINT8 *pSBuf,  UINT32 nFlag,  UINT32 *pErrPsn)
```

PARAMETERS

Parameter	Type	In/Out	Description
nDev	UINT32	In	Physical Device Number (0 ~ 7)
nPsn	UINT32	In	Physical Sector Number
nNumOfScts	UINT32	In	Number of sectors
pstSGL	SGL *	In	Scatter gather list structure for main array of NAND flash memory
pSBuf	UINT8 *	In	Memory buffer for spare array of NAND flash memory
nFlag	UINT32	In	Operation options (ECC on/off, Sync/Async)
pErrPsn	UINT32 *	Out	Pointer of the sector number where the error occurs. When an error occurs, this function stops the write operation. The errored physical sector number is put on this parameter.

nFlag has the the operation options as follows .

Flag	Value	Description
LLD_FLAG_ASYNC_OP	(1 << 0)	Asynchronous Operation
LLD_FLAG_SYNC_OP	(0 << 0)	Synchronous Operation
LLD_FLAG_ECC_ON	(1 << 1)	ECC on (Read Operation with ECC)
LLD_FLAG_ECC_OFF	(0 << 1)	ECC off (Read Operation without ECC)

Flag value is divided into Sync/Async Operation flag and ECC on/off flag. These two flags can be merged as follow.

```
nFlag = LLD_FLAG_SYNC_OP | LLD_FLAG_ECC_ON;
```

RETURN VALUE

Return Value	Description
LLD_SUCCESS	Write Success
LLD_WRITE_ERROR	Write Failure
LLD_WRITE_ERROR LLD_PREV_OP_RESULT	In DCOP (Deferred Check Operation), previous write error happens
LLD_ERASE_ERROR LLD_PREV_OP_RESULT	In DCOP (Deferred Check Operation), previous erase error happens

LLD_WR_PROTECT_ERROR	Write Operation at Locked Area
LLD_ILLEGAL_ACCESS	Illegal Write Operation

REMARKS

This function is a mandatory flash operation function.

DCOP(Deferred Check Operation) : a method optimizing the operation performance. The DCOP method is a procedure to terminate the function (write or erase) right after the command issue. Then it checks the result of the operation at next function call.

The parameters nPsn and nNumOfScts write data as a unit of a sector.
For more information about the byte alignment, refer to 7.5 Byte Alignment Restrictions.

SGL data structure is declared in XsrTypes.h.

□ **SGL** data structure

```
typedef struct
{
    UINT8      nElements;
    SGLEntry   stSGLEntry[XSR_MAX_SGL_ENTRIES];
} SGL;
```

SGLEntry data structure is also declared in XsrTypes.h.

□ **SGLEntry** data structure

```
typedef struct
{
    UINT8* pBuf;      /* Buffer for data */
    UINT16 nSectors;
    /* Number of sectors this entry represents */
    UINT8 nFlag;      /* user data or meta data */
} SGLEntry;
```

SGL is abbreviation for Scatter Gather List. To write whole requested data which exists in several different buffers within one function call, we use SGL.

This function does not support asynchronous mode, whereas XXX_Write supports asynchronous mode.

EXAMPLE

(1) Example to call the function

```
#include <XSRTypes.h>
#include <PAM.h>
#include <LLD.h>
#include <XXX.h> /* Header File of Low Level Device Driver */

BOOL32
Example(VOID)
{
    INT32      nErr;
```

```

UINT32    nDev = 0;
UINT32    nVol = 0;
UINT32    nPSN = 0;
SGL        stSGL;
UINT8     aMBuf[LLD_MAIN_SIZE * nNumOfScts];
UINT8     aSBuf[LLD_SPARE_SIZE * nNumOfScts];
UINT32     nFlag = LLD_FLAG_ECC_ON | LLD_FLAG_SYNC_OP;
LowFuncTbl stLFT[MAX_VOL];

PAM_RegLFT((VOID *)stLFT);

stSGL.stSGLEntry[0].pBuf = aMBuf;
stSGL.stSGLEntry[0].nSectors = nNumOfScts;
stSGL.stSGLEntry[0].nFlag = SGL_ENTRY_USER_DATA;

stSGL.nElements = 1;

nErr = stLFT[nVol].MWrite(nDev, nPSN, nNumOfScts, &stSGL,
                        aSBuf, nFlag);

if (nErr != LLD_SUCCESS)
{
    printf("XXX_Write() fail. ErrCode = %x\n", nErr);
    return (FALSE32);
}
return (TRUE32);
}

```

(2) Example to implement the function in OneNAND

```

INT32
ONLD_MWrite(UINT32 nDev, UINT32 nPsn, UINT32 nScts, SGL *pstSGL,
            UINT8 *pSBuf, UINT32 nFlag, UINT32 *pErrPsn)
{
    UINT16    nBSA;
    UINT32    *pONLDMBuf;
    UINT32    *pONLDSBuf;
    UINT32    nBAAddr;
    UINT32    nPbn;
    UINT32    nWritePsn;
    UINT32    nWriteScts = 0;
    UINT32    nTmpWriteScts;
    UINT8     nBiWriteScts;
    UINT8     nSnapWriteScts;
    UINT32    nRemainScts;
    INT32     nRes;
    UINT8     nSGLIdx;
    UINT8     nSGLCount;
    UINT32    nSctCount = 0;

    UINT8     *pMBuf;
    UINT8     *pSGLBuf = NULL;
    UINT8     nIdx;
    UINT8     nPreSGLIdx = 0;
    UINT8     nTmpRemainScts = 0;
    SGLEntry  *pSGLEnt;

```



```

if (pstSGL == NULL)
    return (ONLD_ILLEGAL_ACCESS);

nBAddr = GET_DEV_BADDR(nDev);

if ((GET_DEV_DID(nDev) & 0x0008) == DDP_CHIP)
{
    if ((nRes = _ChkPrevOpResult(nDev)) != ONLD_SUCCESS)
    {
        return nRes;
    }
}
/* Previous Operation should be flushed at a upper layer */

if (nFlag & ONLD_FLAG_ECC_ON)
{
    /* System Configuration Reg set (ECC On)*/
    ONLD_REG_SYS_CONF1(nBAddr) &= CONF1_ECC_ON;
}
else
{
    /* System Configuration Reg set (ECC Off)*/
    ONLD_REG_SYS_CONF1(nBAddr) |= CONF1_ECC_OFF;
}

/* Set user Data Buffer point */
for (nSGLIdx = 0; nSGLIdx < pstSGL->nElements; nSGLIdx++)
{
    if (pstSGL->stSGLEntry[nSGLIdx].nFlag ==
        SGL_ENTRY_USER_DATA)
    {
        pMBuf = pstSGL->stSGLEntry[nSGLIdx].pBuf;
        break;
    }
}

nSGLIdx = 0;
nRemainScts = nScts;
nWritePsn = nPsn;

switch (nWritePsn & astONLDInfo[nDev].nSctSelSft)
{
    case 0:
        nWriteScts =
            (astONLDInfo[nDev].nSctsPerPG > nRemainScts) ?
            nRemainScts : astONLDInfo[nDev].nSctsPerPG;
        break;
    case 1:
        nWriteScts =
            (astONLDInfo[nDev].nSctSelSft > nRemainScts) ?
            nRemainScts : astONLDInfo[nDev].nSctSelSft;
        break;
    case 2:
        nWriteScts = (2 > nRemainScts) ? nRemainScts : 2;
        break;
    case 3:

```

```

        nWriteScts = (1 > nRemainScts) ? nRemainScts : 1;
        break;
    }

    while (nRemainScts > 0)
    {
        nPbn = GET_PBN(nDev, nWritePsn);

        nTmpWriteScts = nWriteScts;
        nSGLCount = 0;

        if (nSctCount != 0 &&
            pstSGL->stSGLEntry[nSGLIdx].nSectors - nSctCount <
            nTmpWriteScts)
        {
            nTmpRemainScts =
                pstSGL->stSGLEntry[nSGLIdx].nSectors - nSctCount;
        }

        do
        {
            nSctCount += 1;
            if (nSctCount >=
                pstSGL->stSGLEntry[nSGLIdx].nSectors)
            {
                nSctCount = 0;
                nSGLIdx++;
                nSGLCount++;
            }
            else if (nWriteScts == 1)
            {
                nSGLCount++;
            }
        } while(--nWriteScts);

        nBiWriteScts = 0;
        nSnapWriteScts = 0;

        ONLD_REG_START_ADDR2(nBAddr) = (UINT16)MAKE_DBS(nPbn);

        pONLDMBuf = (UINT32*)GET_ONLD_MBUF_ADDR(
            nBAddr, nWritePsn, GET_NXT_BUF_SEL(nDev));
        pONLDSBuf = (UINT32*)GET_ONLD_SBUF_ADDR(
            nBAddr, nWritePsn, GET_NXT_BUF_SEL(nDev));

        for (nIdx = 0; nIdx < nSGLCount; nIdx++)
        {
            pSGLEnt = pstSGL->stSGLEntry[nPreSGLIdx + nIdx];

            switch(pSGLEnt->nFlag)
            {
            case SGL_ENTRY_BISCT_VALID_DATA:
                pSGLBuf = pSGLEnt->pBuf;
                nBiWriteScts += pSGLEnt->nSectors;
                nWriteScts = pSGLEnt->nSectors;
                break;
            }
        }
    }

```

```

case SGL_ENTRY_BISCT_INVALID_DATA:
    pSGLBuf          = pSGLEnt->pBuf;
    nBiWriteScts     += pSGLEnt->nSectors;
    nWriteScts       = pSGLEnt->nSectors;
    break;
case SGL_ENTRY_META_DATA:
    pSGLBuf          = pSGLEnt->pBuf;
    nSnapWriteScts   += pSGLEnt->nSectors;
    nWriteScts       = pSGLEnt->nSectors;
    break;
case SGL_ENTRY_USER_DATA:
    if (nTmpRemainScts != 0)
    {
        nWriteScts     = nTmpRemainScts;
        nTmpRemainScts = 0;
    }
    else
    {
        nWriteScts     =
            nTmpWriteScts - nBiWriteScts - nSnapWriteScts;
        if (nWriteScts > pSGLEnt->nSectors)
        {
            nWriteScts = pSGLEnt->nSectors;
        }
    }
    break;
default:
    return (ONLD_ILLEGAL_ACCESS);
}
if (pSGLEnt->nFlag != SGL_ENTRY_USER_DATA
    && pSGLBuf != NULL)
{
    _WriteMain(pSGLBuf, pONLDMBuf, nWriteScts);
    pONLDMBuf += (BUFF_MAIN_SIZE * nWriteScts);
}
else if (pMBuf != NULL)
{
    /* Memcopy for main data */
    _WriteMain(pMBuf, pONLDMBuf, nWriteScts);
    pMBuf      += (ONLD_MAIN_SIZE * nWriteScts);
    pONLDMBuf += (BUFF_MAIN_SIZE * nWriteScts);
}
}

nPreSGLIdx = nSGLIdx;
nWriteScts = nTmpWriteScts;

if (pSBuf != NULL)
{
    /* Memcopy for spare data */
    _WriteSpare(pSBuf, pONLDSBuf, nWriteScts);
    pSBuf += (ONLD_SPARE_SIZE * nWriteScts);
}
else
{

```

```

        /* Memset for spare data 0xff*/
        MEMSET(pONLDSBuf, 0xFF,
               (ONLD_SPARE_SIZE * nWriteScts));
    }

    if ((nRes = _ChkPrevOpResult(nDev)) != ONLD_SUCCESS)
    {
        *pErrPsn = pstPrevOpInfo[nDev]->nPsn;
        return nRes;
    }

    /* Block Number Set */
    ONLD_REG_START_ADDR1(nBAddr) = (UINT16)MAKE_FBA(nPbn);

    /* Sector Number Set */
    ONLD_REG_START_ADDR8(nBAddr) =
    (UINT16)((nWritePsn << astONLDInfo[nDev].nFPASelSft)
            & MASK_FPA)
        | (nWritePsn & astONLDInfo[nDev].nFSAMask));

    nBSA =
        (UINT16)MAKE_BSA(nWritePsn, GET_NXT_BUF_SEL(nDev));

    ONLD_REG_START_BUF(nBAddr) =
        (UINT16)((nBSA & MASK_BSA) | (nWriteScts & MASK_BSC));

    /* INT Stat Reg Clear */
    ONLD_REG_INT(nBAddr) = (UINT16)INT_CLEAR;

    /*-----*/
    /* ONLD Write CMD is issued */
    /*-----*/
    if (pMBuf != NULL || pSGLBuf != NULL)
    {
        ONLD_REG_CMD(nBAddr) = (UINT16)ONLD_CMD_WRITE_PAGE;
    }
    else
    {
        ONLD_REG_CMD(nBAddr) = (UINT16)ONLD_CMD_WRITE_SPARE;
    }

    pstPrevOpInfo[nDev]->ePreOp = WRITE;
    pstPrevOpInfo[nDev]->nPsn = nWritePsn;
    pstPrevOpInfo[nDev]->nScts = nWriteScts;
    pstPrevOpInfo[nDev]->nFlag = nFlag;

    nRemainScts -= nWriteScts;
    nWritePsn += nWriteScts;
    nWriteScts =
        (astONLDInfo[nDev].nSctsPerPG > nRemainScts) ?
        nRemainScts : astONLDInfo[nDev].nSctsPerPG;

    pstPrevOpInfo[nDev]->nBufSel =
        GET_NXT_BUF_SEL(nDev);
}

```



```
    return (ONLD_SUCCESS);  
}
```

SEE ALSO

XXX_Read, XXX_Write, XXX_Erase, XXX_Copyback, XXX_MRead,
XXX_MErase, XXX_EraseVerify

XXX_MErase

DESCRIPTION

This function erases blocks of NAND flash memory. XXX_MErase, unlikely XXX_Erase, erases multiple blocks simultaneously. When a device supports multi-block erase operation, XXX_MErase can be used.

SYNTAX

```
INT32
XXX_MErase(UINT32 nDev, LLDMEArg *pstMEArg, UINT32 nFlag)
```

PARAMETERS

Parameter	Type	In/Out	Description
nDev	UINT32	In	Physical Device Number (0 ~ 7)
pstMEArg	LLDMEArg	In	Pointer to LLDMEArg data structure
nFlag	UINT32	In	Operation options (Sync/Async)

nFlag has the the operation options as follows .

Flag	Value	Description
LLD_FLAG_ASYNC_OP	(1 << 0)	Asynchronous Operation
LLD_FLAG_SYNC_OP	(0 << 0)	Synchronous Operation

RETURN VALUE

Return Value	Description
LLD_SUCCESS	Erase Success
LLD_WR_PROTECT_ERROR LLD_PREV_OP_RESULT	In DCOP (Deferred Check Operation), previous write Protection error happens
LLD_PREV_WRITE_ERROR LLD_PREV_OP_RESULT	In DCOP (Deferred Check Operation), previous write error happens
LLD_PREV_ERASE_ERROR LLD_PREV_OP_RESULT	In DCOP (Deferred Check Operation), previous erase error happens
LLD_ILLEGAL_ACCESS	Illegal Erase Operation

REMARKS

This function is a mandatory flash operation function.

DCOP(Deferred Check Operation) : a method optimizing the operation performance. The DCOP method is a procedure to terminate the function (write or erase) right after the command issue. Then it checks the result of the operation at next function call.

In order to support asynchronous mode, codes which execute next steps must be added. First, check the value of a flag. And then, clear an interrupt. Finally, enable the interrupt. For more information about the interrupt, refer to Chapter 7.2 Interrupt.

XXX_MErase can be used by only devices which support multi-block erase feature (for example OneNAND256). Multi-block erase operation erases multiple blocks simultaneously. The unit of erase operation is 16 blocks at maximum. After XXX_MErase,

the blocks must be verified by XXX_EraseVerify.

XXX_MEraser must have information about blocks to be erased. **LLDMEArg** data structure and **LLDMEList** data structure that are required for additional information are as follows

LLDMEArg data structure is declared in LLD.h.

□ **LLDMEArg** data structure

```
typedef struct {
    LLDMEList *pstMEList; /* Pointer to LLDMEList */
    UINT16     nNumOfMList; /* Number of Entries of LLDMEList */
    UINT16     nBitMapErr; /* Error Bitmap Position of MEList */
    BOOL32     bFlag;      /* Valid Flag */
} LLDMEArg;
```

Table 5-3 describes LLDMEArg data structure.

Table 5-3. LLDMEArg data structure in LLD.h

Member Variable	Description
pstMEList	Pointer to LLDMEList data structure
nNumOfMList	Number of blocks to be erased simultaneously. The maximum number of blocks is 16.
nBitMapErr	Each bit indicates whether error occurs or not for each block
bFlag	Valid flag for a block list in LLDMEList

LLDMEList data structure is also declared in LLD.h.

□ **LLDMEList** data structure

```
typedef struct {
    UINT16 nMEListSbn; /* MEList Semi-physical Block Number */
    UINT16 nMEListPbn; /* MEList Physical Block Number */
} LLDMEList;
```

Table 5-4 describes LLDMEList data structure.

Table 5-4. LLDMEList data structure in LLD.h

Member Variable	Description
nMEListSbn	Semi-physical block number of a block in block list
nMEListPbn	Physical block number of a block in block list

EXAMPLE

(1) Example to call the function

```
#include <XSRTypes.h>
#include <PAM.h>
#include <LLD.h>
#include <XXX.h> /* Header File of Low Level Device Driver */
```

```

BOOL32
Example(VOID)
{
    INT32      nErr;
    UINT32     nDev = 0;
    UINT32     nVol = 0;
    UINT16     nPbn = 13;
    UINT16     nSbn = 13;
    UINT16     nNum = 0;
    UINT16     nNumOfPbn = 1;
    UINT32     nFlag = LLD_FLAG_ASYNC_OP;
    LowFuncTbl stLFT[MAX_VOL];
    LLDMEArg   *pstLLDMEArg[XSR_MAX_DEV];
    LLDMEList  *pstLLDMEList;

    pstLLDMEArg[nDev]->nBitMapErr = (UINT16)0x0;
    pstLLDMEArg[nDev]->nNumOfMList = nNumOfPbn;
    pstLLDMEArg[nDev]->bFlag      = TRUE32;

    pstLLDMEList = pstLLDMEArg[nDev]->pstMEList;

    pstLLDMEList[nNum].nMEListSbn = nSbn;
    pstLLDMEList[nNum].nMEListPbn = nPbn;

    PAM_RegLFT((VOID *)stLFT);

    nErr = stLFT[nVol].MErase(nDev, pstLLDMEArg[nDev], nFlag)

    if (nErr != LLD_SUCCESS)
    {
        printf("XXX_MEraser()fail. ErrCode = %x\n", nErr);
        return (FALSE32);
    }
    return (TRUE32);
}

```

(2) Example to implement the function in OneNAND

```

INT32
ONLD_MEraser(UINT32 nDev, LLDMEArg *pstMEArg, UINT32 nFlag)
{
    UINT32     nBAddr, nPbn;
    UINT32     nCurPbn;
    UINT32     nFirstchipPbn = 0xffff;
    UINT32     nSecondchipPbn = 0xffff;
    UINT16     nCnt;
    INT32      nRes;
    INT32      nRet = ONLD_SUCCESS;
    LLDMEList  *pstMEPList;

    nBAddr = GET_DEV_BADDR(nDev);

    if ((nRes = _ChkPrevOpResult(nDev)) != ONLD_SUCCESS)
        return nRes;

    pstMEPList = pstMEArg->pstMEList;

```



```

for(nCnt = 0; nCnt < pstMEArg->nNumOfMList; nCnt++)
{
    nCurPbn = pstMEPList[nCnt].nMEListPbn;

    if (((nCurPbn << 5) & MASK_DBS) == 0x0000)
        && (nFirstchipPbn == 0xffff))
    {
        nFirstchipPbn = nCurPbn;
        continue;
    }
    else if (((nCurPbn << 5) & MASK_DBS) == 0x8000)
        && (nSecondchipPbn == 0xffff))
    {
        nSecondchipPbn = nCurPbn;
        continue;
    }

    /* Block Number Set */
    ONLD_REG_START_ADDR1(nBAddr) = (UINT16)MAKE_FBA(nCurPbn);
    ONLD_REG_START_ADDR2(nBAddr) = (UINT16)MAKE_DBS(nCurPbn);
    if (ONLD_REG_WR_PROTECT_STAT(nBAddr) !=
        (UINT16)(UNLOCKED_STAT))
    {
        return (ONLD_WR_PROTECT_ERROR);
    }
    /* INT Stat Reg Clear */
    ONLD_REG_INT(nBAddr) = (UINT16)INT_CLEAR;
    /*-----*/
    /* ONLD Erase CMD is issued */
    /*-----*/
    ONLD_REG_CMD(nBAddr) = (UINT16)ONLD_CMD_ERASE_MBLK;

    while (GET_ONLD_INT_STAT(nBAddr, PEND_INT) !=
        (UINT16)PEND_INT)
    {
        /* Wait until device ready */
    }
}

if ((nFirstchipPbn != 0xffff) && (nSecondchipPbn != 0xffff))
{
    /* Block Number Set */
    ONLD_REG_START_ADDR1(nBAddr) =
        (UINT16)MAKE_FBA(nFirstchipPbn);
    ONLD_REG_START_ADDR2(nBAddr) =
        (UINT16)MAKE_DBS(nFirstchipPbn);

    if (ONLD_REG_WR_PROTECT_STAT(nBAddr) !=
        (UINT16)(UNLOCKED_STAT))
    {
        return (ONLD_WR_PROTECT_ERROR);
    }
    /* INT Stat Reg Clear */
    ONLD_REG_INT(nBAddr) = (UINT16)INT_CLEAR;

```

```

        ONLD_REG_CMD(nBAddr) = (UINT16)ONLD_CMD_ERASE_BLK;

        while (GET_ONLD_INT_STAT(nBAddr, PEND_ERASE) !=
                (UINT16)PEND_ERASE)
        {
            /* Wait until device ready */
        }

        /* Block Number Set */
        ONLD_REG_START_ADDR1(nBAddr) =
            (UINT16)MAKE_FBA(nSecondchipPbn);
        ONLD_REG_START_ADDR2(nBAddr) =
            (UINT16)MAKE_DBS(nSecondchipPbn);

        if (ONLD_REG_WR_PROTECT_STAT(nBAddr) !=
            (UINT16)(UNLOCKED_STAT))
        {
            return (ONLD_WR_PROTECT_ERROR);
        }

        /* INT Stat Reg Clear */
        ONLD_REG_INT(nBAddr) = (UINT16)INT_CLEAR;
        ONLD_REG_CMD(nBAddr) = (UINT16)ONLD_CMD_ERASE_BLK;

        nPbn = nSecondchipPbn;
    }
    else if (nFirstchipPbn != 0xffff)
    {
        /* Block Number Set */
        ONLD_REG_START_ADDR1(nBAddr) =
            (UINT16)MAKE_FBA(nFirstchipPbn);
        ONLD_REG_START_ADDR2(nBAddr) =
            (UINT16)MAKE_DBS(nFirstchipPbn);

        if (ONLD_REG_WR_PROTECT_STAT(nBAddr) !=
            (UINT16)(UNLOCKED_STAT))
        {
            return (ONLD_WR_PROTECT_ERROR);
        }

        /* INT Stat Reg Clear */
        ONLD_REG_INT(nBAddr) = (UINT16)INT_CLEAR;
        ONLD_REG_CMD(nBAddr) = (UINT16)ONLD_CMD_ERASE_BLK;

        nPbn = nFirstchipPbn;
    }
    else
    {
        /* Block Number Set */
        ONLD_REG_START_ADDR1(nBAddr) =
            (UINT16)MAKE_FBA(nSecondchipPbn);
        ONLD_REG_START_ADDR2(nBAddr) =
            (UINT16)MAKE_DBS(nSecondchipPbn);
        if (ONLD_REG_WR_PROTECT_STAT(nBAddr) !=
            (UINT16)(UNLOCKED_STAT))

```

```

    {
        return (ONLD_WR_PROTECT_ERROR);
    }
    /* INT Stat Reg Clear */
    ONLD_REG_INT(nBAddr)    = (UINT16)INT_CLEAR;
    ONLD_REG_CMD(nBAddr)    = (UINT16)ONLD_CMD_ERASE_BLK;

    nPbn = nSecondchipPbn;

    while (GET_ONLD_INT_STAT(nBAddr, PEND_ERASE)
           != (UINT16)PEND_ERASE)
    {
        /* Wait until device ready */
    }
}

pstPrevOpInfo[nDev]->ePreOp = MERASE;
pstPrevOpInfo[nDev]->nFlag = nFlag;
pstPrevOpInfo[nDev]->nPsn = nPbn * GET_SCTS_PER_BLK(nDev);
MEMCPY(pstPrevOpInfo[nDev]->pstPreMEArg,
       pstMEArg, sizeof(LLDMEArg));

/* in case of async mode, interrupt should be enabled */
if (nFlag & ONLD_FLAG_ASYNC_OP)
{
    PAM_ClearInt((UINT32)XSR_INT_ID_NAND_0);
    PAM_EnableInt((UINT32)XSR_INT_ID_NAND_0);
}

return nRet;
}

```

SEE ALSO

XXX_Read, XXX_Write, XXX_Erase, XXX_Copyback, XXX_MRead,
XXX_MWrite, XXX_EraseVerify

XXX_EraseVerify

DESCRIPTION

This function verifies an erase operation whether it checks blocks are properly erased. Mainly this function is used with XXX_MErase function.

SYNTAX

```
INT32
MyLLD_EraseVerify(UINT32 nDev, LLDMEArg *pstMEArg, UINT32 nFlag)
```

PARAMETERS

Parameter	Type	In/Out	Description
nDev	UINT32	In	Physical Device Number (0 ~ 7)
pstMEArg	LLDMEArg	In	Pointer to LLDMEArg data structure
nFlag	UINT32	In	Operation options (Sync/Async)

nFlag has the the operation options as follows .

Flag	Value	Description
LLD_FLAG_ASYNC_OP	(1 << 0)	Asynchronous Operation
LLD_FLAG_SYNC_OP	(0 << 0)	Synchronous Operation

RETURN VALUE

Return Value	Description
LLD_SUCCESS	Erase Success
LLD_WR_PROTECT_ERROR LLD_PREV_OP_RESULT	In DCOP (Deferred Check Operation), previous write Protection error happens
LLD_PREV_WRITE_ERROR LLD_PREV_OP_RESULT	In DCOP (Deferred Check Operation), previous write error happens
LLD_PREV_ERASE_ERROR LLD_PREV_OP_RESULT	In DCOP (Deferred Check Operation), previous erase error happens
LLD_ERASE_ERROR	Erase Failure

REMARKS

This function is a mandatory flash operation function.

DCOP(Deferred Check Operation) : a method optimizing the operation performance. The DCOP method is a procedure to terminate the function (write or erase) right after the command issue. Then it checks the result of the operation at next function call.

After erase operation, XXX_EraseVerify checks all blocks in LLDMEList of LLDMEArg. If blocks that is not erased properly are detected, XXX_EraseVerify returns erase error. XXX_MErase requires XXX_EraseVerify because XXX_MErase does not support the functionality to verify erase errors. XXX_EraseVerify only can be used when a device supports erase verify functionality. For more information about LLDMEList and LLDMEArg data structure, refer to the API page of XXX_MErase.

EXAMPLE

(1) Example to call the function

```
#include <XSRTypes.h>
#include <PAM.h>
#include <LLD.h>
#include <XXX.h> /* Header File of Low Level Device Driver */

BOOL32
Example(VOID)
{
    INT32      nErr;
    UINT32     nDev = 0;
    UINT32     nVol = 0;
    UINT16     nPbn = 13;
    UINT16     nSbn = 13;
    UINT16     nNum = 0;
    UINT16     nNumOfPbn = 1;
    UINT32     nFlag = LLD_FLAG_ASYNC_OP;

    LowFuncTbl stLFT[MAX_VOL];
    LLDMEArg   *pstLLDMEArg[XSR_MAX_DEV];
    LLDMEList  *pstLLDMEList;

    pstLLDMEArg[nDev]->nBitMapErr = (UINT16)0x0;
    pstLLDMEArg[nDev]->nNumOfMList = nNumOfPbn;
    pstLLDMEArg[nDev]->bFlag      = TRUE32;

    pstLLDMEList = pstLLDMEArg[nDev]->pstMEList;

    pstLLDMEList[nNum].nMEListSbn = nSbn;
    pstLLDMEList[nNum].nMEListPbn = nPbn;

    PAM_RegLFT((VOID *)stLFT);

    nErr = stLFT[nDev].EraseVerify(nDev, pstLLDMEArg[nDev],
                                   nFlag)

    if (nErr != LLD_SUCCESS)
    {
        printf("XXX_EraseVerify()fail. ErrCode = %x\n", nErr);
        return (FALSE32);
    }
    return (TRUE32);
}
```

(2) Example to implement the function in OneNAND

```
INT32
ONLD_EraseVerify(UINT32 nDev, LLDMEArg *pstMEArg, UINT32 nFlag)
{
    UINT32     nBAddr;
    UINT32     nCnt;
    INT32      nRet = ONLD_SUCCESS;
    LLDMEList  *pstMEPList;

    nBAddr = GET_DEV_BADDR(nDev);
```

```

/* for verify */
pstMEPList = pstMEArg->pstMEList;

if((nFlag & ONLD_FLAG_SYNC_MASK) == ONLD_FLAG_SYNC_OP)
{
    while (GET_ONLD_INT_STAT(nBAddr, PEND_ERASE)
           != (UINT16)PEND_ERASE)
    {
        /* Wait until device ready */
    }
    pstPrevOpInfo[nDev]->ePreOp      = NONE;
}

for(nCnt = 0; nCnt < pstMEArg->nNumOfMList; nCnt++)
{
    /* Block Number Set */
    ONLD_REG_START_ADDR1(nBAddr)
        = (UINT16)MAKE_FBA(pstMEPList[nCnt].nMEListPbn);
    ONLD_REG_START_ADDR2(nBAddr)
        = (UINT16)MAKE_DBS(pstMEPList[nCnt].nMEListPbn);

    /* Erase Verify Command issue */
    ONLD_DBG_PRINT((TEXT("[ONLD:MSG] Erase Verify\r\n")));

    /* INT Stat Reg Clear */
    ONLD_REG_INT(nBAddr) = (UINT16)INT_CLEAR;

    ONLD_REG_CMD(nBAddr) = (UINT16)ONLD_CMD_ERASE_VERIFY;

    while (GET_ONLD_INT_STAT(nBAddr, PEND_INT)
           != (UINT16)PEND_INT)
    {
        /* Wait until device ready */
    }

    /* Erase Operation Verifying Error Check */
    if (GET_ONLD_CTRL_STAT(nBAddr, ERROR_STATE)
        == ERROR_STATE)
    {
        pstMEArg->nBitMapErr |= (1 << nCnt);
        nRet = LLD_MERASE_ERROR;
    }
}

return (nRet);
}

```

SEE ALSO

XXX_Read, XXX_Write, XXX_Erase, XXX_Copyback, XXX_MRead,
XXX_MWrite, XXX_MErase

XXX_CopyBack

DESCRIPTION

This function copies data by using internal buffer in the device.

SYNTAX

```
INT32
XXX_CopyBack(UINT32 nDev, CpBkArg *pstCpArg, UINT32 nFlag)
```

PARAMETERS

Parameter	Type	In/Out	Description
nDev	UINT32	In	Physical Device Number (0 ~ 7)
pstCpArg	CpBkArg	In	Pointer to CpBkArg data structure
nFlag	UINT32	In	Operation options (ECC on/off, Sync/Async)

nFlag has the operation options as follows.

Flag	Value	Description
LLD_FLAG_ASYNC_OP	(1 << 0)	Asynchronous Operation
LLD_FLAG_SYNC_OP	(0 << 0)	Synchronous Operation
LLD_FLAG_ECC_ON	(1 << 1)	ECC on (Read Operation with ECC)
LLD_FLAG_ECC_OFF	(0 << 1)	ECC off (Read Operation without ECC)

Flag value is divided into Sync/Async Operation flag and ECC on/off flag. These two flags can be merged as follows.

```
nFlag = LLD_FLAG_SYNC_OP | LLD_FLAG_ECC_ON;
```

RETURN VALUE

Return Value	Description
LLD_SUCCESS	Copyback Success
LLD_READ_ERROR ECC result code	Data Integrity Fault (1 or 2bit ECC error)
LLD_WRITE_ERROR	Write Operation Error
LLD_PREV_WRITE_ERROR LLD_PREV_OP_RESULT	In DCOP (Deferred Check Operation), previous write error happens
LLD_PREV_ERASE_ERROR LLD_PREV_OP_RESULT	In DCOP (Deferred Check Operation), previous erase error happens
LLD_ILLEGAL_ACCESS	Illegal Copyback Operation

REMARKS

This function is mandatory.

DCOP(Deferred Check Operation) : a method optimizing the operation performance. The DCOP method is a procedure to terminate the function (write or erase) right after the command issue. Then it checks the result of the operation at next function call.

Examples in this document do not implement the interrupt, because this Copyback example is implemented by calling Read and Write functions directly. If implementation of Copyback does not call Read and Write functions directly, codes which execute next steps must be added to support asynchronous mode. First, check the value of a flag. And then, clear an interrupt. Finally, enable the interrupt.

Copyback means the operation method to copy pages using the internal buffer in a NAND device. This copyback method improves the performance by cutting the transfer time and operation procedure, because this method does not use the external memory. When copying a page using the copyback method, a part of data can be brought the outside device; this is called **Random-in**.

RndInArg and **CpBkArg** data structures are declared in `XsrTypes.h`.

□ **RndInArg** data structure

```
typedef struct
{
    UINT16    nOffset;        /* nOffset : sector offset(0 - 3) * 1024
                               + Main(0 - 511) + Spare(512 - 527) */
    UINT16    nNumOfBytes;    /* Random In Bytes */
    UINT8     *pBuf;          /* Data Buffer Pointer */
} RndInArg;
```

Table 5-5 describes **RndInArg** data structure.

Table 5-5. RndInArg data structure in XsrTypes.h

Member Variable	Description
nNumOfBytes	The size of data to be changed
nOffset	The location of data to Random-in Offset : sector offset(0 - 3) * 1024+ Main(0 - 511) + Spare(512 - 527)
pBuf	The pointer of data to be changed

□ **CpBkArg** data structure

```
typedef struct
{
    UINT32    nSrcSn;         /* Copy Back Source Vsn, should be page
                               aligned */
    UINT32    nDstSn;         /* Copy Back Dest. Vsn, should be page
                               aligned */
    UINT32    nRndInCnt;      /* Random In Count */
    RndInArg  *pstRndInArg;   /* RndInArg Array pointer */
} CpBkArg;
```

Table 5-6 describes **CpBkArg** data structure.

Table 5-6. CpBkArg data structure in XsrTypes.h

Member Variable	Description
nSrcSn	The location of a source page in NAND flash memory Physical sector number aligned at Physical page address
nDstSn	The location of data to be copied Physical sector number aligned at Physical page address
nRndInCnt	The count of Random-in
pstRndInArg	Pointer to RndInArg data structure

EXAMPLE

(1) Example to call the function

```
#include <XSRTypes.h>
#include <PAM.h>
#include <LLD.h>
#include <XXX.h> /* Header File of Low Level Device Driver */

BOOL32
Example(VOID)
{
    INT32    nErr;
    UINT32   nDev = 0;
    UINT32   nVol = 0;
    UINT8    aMBuf[XSR_MAIN_SIZE];
    UINT8    aSBuf[XSR_SPARE_SIZE];
    CpBkArg  stCpArg;
    RndInArg stRIArg[2];
    UINT32   nFlag = LLD_FLAG_ASYNC_OP;
    LowFuncTbl stLFT[MAX_VOL];

    PAM_RegLFT((VOID *)stLFT);

    stCpArg.nSrcSn      = 32;
    stCpArg.nDstSn      = 64;
    stCpArg.nRndInCnt    = 2;
    stCpArg.pstRndInArg = &stRIArg[0];
    stRIArg[0].nOffset   = 0;
    stRIArg[0].nNumOfBytes = LLD_MAIN_SIZE;
    stRIArg[0].pBuf      = &aMBuf[0];
    stRIArg[1].nOffset   = 512;
    stRIArg[1].nNumOfBytes = LLD_SPARE_SIZE;
    stRIArg[1].pBuf      = &aSBuf[0];

    nErr = stLFT[nVol].CopyBack(nDev, &stCpArg, nFlag);

    if (nErr != LLD_SUCCESS)
    {
        printf("XXX_CopyBack()fail. ErrCode = %x\n", nErr);
        return (FALSE32);
    }
    return (TRUE32);
}
```

(2) Example to implement the function in OneNAND

```
INT32
```

```

ONLD_CopyBack(UINT32 nDev, CpBkArg *pstCpArg, UINT32 nFlag)
{
    UINT8          aBABuf[2];
    UINT8          *pRIBuf;
    INT32          nSctNum, nOffset;
    UINT16         nBSA;
    UINT16         nEccRes;
    UINT16         nPbn;
    volatile UINT16 *pDevBuf;
    UINT32         nCnt;
    UINT32         *pONLDMBuf, *pONLDSBuf;
    UINT32         nBAddr;
    UINT32         nRISize;
    INT32          nRes;
    RndInArg      *pstRIArg;

    nBAddr = GET_DEV_BADDR(nDev);

    if ((nRes = _ChkPrevOpResult(nDev)) != ONLD_SUCCESS)
    {
        return nRes;
    }
    /*-----*/
    /* Step 1. READ */
    /*-----*/

    nPbn = GET_PBN(nDev, pstCpArg->nSrcSn);

    /* Block Number Set */
    ONLD_REG_START_ADDR1(nBAddr) = (UINT16)MAKE_FBA(nPbn);
    /* Device BufferRam Select */
    ONLD_REG_START_ADDR2(nBAddr) = (UINT16)MAKE_DBS(nPbn);

    /* Sector Number Set */
    ONLD_REG_START_ADDR8(nBAddr) = (UINT16)(
        ((pstCpArg->nSrcSn << astONLDInfo[nDev].nFPASelSft)
        & MASK_FPA)
        | (pstCpArg->nSrcSn & astONLDInfo[nDev].nFSAMask));

    nBSA = (UINT16)MAKE_BSA(0, DATA_BUF0);

    ONLD_REG_START_BUF(nBAddr) = (UINT16)(
        (nBSA & MASK_BSA) | (GET_SCTS_PER_PG(nDev) & MASK_BSC));

    pONLDMBuf = (UINT32*)GET_ONLD_MBUF_ADDR(
        nBAddr, 0, DATA_BUF0);
    pONLDSBuf = (UINT32*)GET_ONLD_SBUF_ADDR(
        nBAddr, 0, DATA_BUF0);

    if (nFlag & ONLD_FLAG_ECC_ON)
    {
        /* System Configuration Reg set (ECC On)*/
        ONLD_REG_SYS_CONF1(nBAddr) &= CONF1_ECC_ON;
    }
    else
    {

```

```

    /* System Configuration Reg set (ECC Off)*/
    ONLD_REG_SYS_CONF1(nBAddr) |= CONF1_ECC_OFF;
}

/* INT Stat Reg Clear */
ONLD_REG_INT(nBAddr)          = (UINT16)INT_CLEAR;

/*-----*/
/* ONLD Read CMD is issued */
/*-----*/
/* Page Read Command issue */
ONLD_REG_CMD(nBAddr) = (UINT16)ONLD_CMD_READ_PAGE;

while (GET_ONLD_INT_STAT(nBAddr, PEND_READ)
      != (UINT16)PEND_READ)
{
    /* Wait until device ready */
}

if (nFlag & ONLD_FLAG_ECC_ON)
{
    nEccRes = ONLD_REG_ECC_STAT(nBAddr);
    if (nEccRes & ONLD_READ_UERROR_A)
    {
        return (ONLD_READ_ERROR | nEccRes);
    }
}

/*-----*/
/* Step 2. DATA RANDOM IN */
/*-----*/

pstRIArg = pstCpArg->pstRndInArg;

for (nCt = 0 ; nCt < pstCpArg->nRndInCnt ; nCt++)
{
    nSctNum = (pstRIArg[nCt].nOffset / 1024);
    nOffset = (pstRIArg[nCt].nOffset % 1024);
    pRIBuf = pstRIArg[nCt].pBuf;
    nRISize = pstRIArg[nCt].nNumOfBytes;

    if (nOffset < ONLD_MAIN_SIZE)
    {
        /* byte align for 16 */
        if ((nOffset % sizeof(UINT16)) != 0)
        {
            pDevBuf =(volatile UINT16*)((UINT32)
                pONLDMBuf + (nSctNum * ONLD_MAIN_SIZE)
                + nOffset - 1);

            *(volatile UINT16*)aBABuf = *pDevBuf;
            aBABuf[1] = *pRIBuf++;
            *pDevBuf = *(volatile UINT16*)aBABuf;

            if (--nRISize <= 0)
            {

```

```

        continue;
    }
    nOffset++;
}

if ((nRISize % sizeof(UINT16)) != 0)
{
    pDevBuf = (volatile UINT16*)((UINT32)
        pONLDMBuf + (nSctNum * ONLD_MAIN_SIZE)
        + nOffset + nRISize - 1);

    *(volatile UINT16*)aBABuf = *pDevBuf;
    aBABuf[0] = *(pRIBuf + nRISize - 1);
    *pDevBuf = *(volatile UINT16*)aBABuf;

    if (--nRISize <= 0)
    {
        continue;
    }
}

MEMCPY((UINT8*)((UINT32)
    pONLDMBuf+(nSctNum * ONLD_MAIN_SIZE)+nOffset),
    pRIBuf,
    nRISize);
}
else
{
    if ((nOffset % sizeof(UINT16)) != 0)
    {
        pDevBuf = (volatile UINT16*)((UINT32)
            pONLDSBuf + (nSctNum * ONLD_SPARE_SIZE)
            + (nOffset - ONLD_MAIN_SIZE) - 1);

        *(volatile UINT16*)aBABuf = *pDevBuf;
        aBABuf[1] = *pRIBuf++;
        *pDevBuf = *(volatile UINT16*)aBABuf;
        if (--nRISize <= 0)
        {
            continue;
        }
        nOffset++;
    }

    if ((nRISize % sizeof(UINT16)) != 0)
    {
        pDevBuf = (volatile UINT16*)((UINT32)
            pONLDSBuf + (nSctNum * ONLD_SPARE_SIZE)
            + (nOffset - ONLD_MAIN_SIZE)
            + nRISize - 1);

        *(volatile UINT16*)aBABuf = *pDevBuf;
        aBABuf[0] = *(pRIBuf + nRISize - 1);
        *pDevBuf = *(volatile UINT16*)aBABuf;

        if (--nRISize <= 0)

```

```

        {
            continue;
        }
    }

    MEMCPY((UINT8*)((UINT32)
        pONLDSBuf + (nSctNum * ONLD_SPARE_SIZE)
        + (nOffset - ONLD_MAIN_SIZE)),
        pRIBuf,
        nRISize);
    }
}

/*-----*/
/* Step 3. WRITE */
/*-----*/
nPbn = GET_PBN(nDev, pstCpArg->nDstSn);

/* Block Number Set */
ONLD_REG_START_ADDR1(nBAddr) = (UINT16)MAKE_FBA(nPbn);
ONLD_REG_START_ADDR2(nBAddr) = (UINT16)MAKE_DBS(nPbn);
/* Sector Number Set */
ONLD_REG_START_ADDR8(nBAddr) = (UINT16)(
    ((pstCpArg->nDstSn << astONLDInfo[nDev].nFPASelSft)
    & MASK_FPA)
    | (pstCpArg->nDstSn & astONLDInfo[nDev].nFSAMask));

nBSA = (UINT16)MAKE_BSA(pstCpArg->nDstSn, DATA_BUF0);

ONLD_REG_START_BUF(nBAddr) = (UINT16)(
    (nBSA & MASK_BSA) |
    (GET_SCTS_PER_PG(nDev) & MASK_BSC));

if (nFlag & ONLD_FLAG_ECC_ON)
{
    /* System Configuration Reg set (ECC On)*/
    ONLD_REG_SYS_CONF1(nBAddr) &= CONF1_ECC_ON;
}
else
{
    /* System Configuration Reg set (ECC Off)*/
    ONLD_REG_SYS_CONF1(nBAddr) |= CONF1_ECC_OFF;
}

ONLD_REG_INT(nBAddr) = (UINT16)INT_CLEAR;

/*-----*/
/* ONLD Write CMD is issued */
/*-----*/

/* Main Write Command issue */
ONLD_REG_CMD(nBAddr) = (UINT16)ONLD_CMD_WRITE_PAGE;

pstPrevOpInfo[nDev]->ePreOp = CPBACK;
pstPrevOpInfo[nDev]->nPsn = pstCpArg->nDstSn;
pstPrevOpInfo[nDev]->nScts = GET_SCTS_PER_PG(nDev);

```

```
pstPrevOpInfo[nDev]->nFlag      = nFlag;
pstPrevOpInfo[nDev]->nBufSel    = DATA_BUF0;

/* in case async mode, interrupt should be enabled */
if (nFlag & ONLD_FLAG_ASYNC_OP)
{
    PAM_ClearInt((UINT32)XSR_INT_ID_NAND_0);
    PAM_EnableInt((UINT32)XSR_INT_ID_NAND_0);
}

return (ONLD_SUCCESS);
}
```

SEE ALSO

XXX_Read, XXX_Write, XXX_Erase, XXX_MRead, XXX_MWrite,
XXX_MErase, XXX_EraseVerify

XXX_ChkInitBadBlk

DESCRIPTION

This function checks whether a block is an initial bad block or not.

SYNTAX

```
INT32
XXX_ChkInitBadBlk(UINT32 nDev, UINT32 nPbn)
```

PARAMETERS

Parameter	Type	In/Out	Description
nDev	UINT32	In	Physical Device Number (0 ~ 7)
nPbn	UINT32	In	Physical Block Number

RETURN VALUE

Return Value	Description
LLD_SUCCESS	Bad Block Check Success
LLD_INIT_GOODBLOCK	In a Good Block (Not a Bad Block)
LLD_INIT_BADBLOCK	In an Initial Bad Block
LLD_PREV_WRITE_ERROR LLD_PREV_OP_RESULT	In DCOP (Deferred Check Operation), previous write error happens
LLD_PREV_ERASE_ERROR LLD_PREV_OP_RESULT	In DCOP (Deferred Check Operation), previous erase error happens
LLD_ILLEGAL_ACCESS	Illegal Operation

REMARKS

This function is mandatory.

DCOP(Deferred Check Operation) : a method optimizing the operation performance. The DCOP method is a procedure to terminate the function (write or erase) right after the command issue. Then it checks the result of the operation at next function call.

If the value of the bad mark position in the first or second page of a block is not 0xff(a normal statement), the block is the initial bad block.

EXAMPLE

(1) Example to call the function

```
#include <XSRTypes.h>
#include <PAM.h>
#include <LLD.h>
#include <XXX.h> /* Header File of Low Level Device Driver */

BOOL32
Example(VOID)
{
    INT32    nErr;
    UINT32   nDev = 0;
```

```

UINT32    nVol = 0;
UINT32    nPbn = 0;
LowFuncTbl stLFT[MAX_VOL];

PAM_RegLFT((VOID *)stLFT);

nErr = stLFT[nVol].ChkInitBadBlk(nDev, nPbn);

if (nErr != LLD_SUCCESS)
{
    printf("XXX_ ChkInitBadBlk()fail. ErrCode = %x\n", nErr);
    return (FALSE32);
}
return (TRUE32);
}

```

(2) Example to implement the fuinction in OneNAND

```

INT32
ONLD_ChkInitBadBlk(UINT32 nDev, UINT32 nPbn)
{
    INT32    nRet;
    UINT16   aSpare[ONLD_SPARE_SIZE / 2];
    UINT32   nPsn;
    UINT32   nBAddr;

    nBAddr = GET_DEV_BADDR(nDev);

    nPsn = nPbn * GET_SCTS_PER_BLK(nDev);

    nRet = ONLD_Read(nDev,          /* Device Number          */
                    nPsn,          /* Physical Sector Number */
                    (UINT32)1,     /* Number of Sectors to be read */
                    (UINT8*)NULL,  /* Buffer pointer for Main area */
                    (UINT8*)aSpare, /* Buffer pointer for Spare area */
                    (UINT32)ONLD_FLAG_ECC_OFF); /*flag */

    if ((nRet != ONLD_SUCCESS) && ((nRet & (0xFFFF0000))
        != (UINT32)ONLD_READ_ERROR))
    {
        return (nRet);
    }

    if (aSpare[0] != (UINT16)VALID_BLK_MARK)
    {
        return (ONLD_INIT_BADBLOCK);
    }

    return (ONLD_INIT_GOODBLOCK);
}

```

SEE ALSO

XXX_SetRWArea

DESCRIPTION

This function is called when NAND device provides write/erase protection functionality in hardware.

SYNTAX

```
INT32
XXX_SetRWArea(UINT32 nDev, UINT32 n1stUB, UINT32 nNumOfUBs)
```

PARAMETERS

Parameter	Type	In/Out	Description
nDev	UINT32	In	Physical Device Number (0 ~ 7)
n1stUB	UINT32	In	Start block index of unlocked area
nNumOfUBs	UINT32	In	Total number of blocks of unlocked area nNumOfUBs = 0, the device is locked.

RETURN VALUE

Return Value	Description
LLD_SUCCESS	Unlock Aea Setting Success
LLD_ILLEGAL_ACCESS	Illegal Setting

REMARKS

This function is optional.

If the hardware does not provide write/erase protection functionality in hardware, a user does not need to implement this function.

EXAMPLE

(1) Example to call the function

```
#include <XSRTypes.h>
#include <PAM.h>
#include <LLD.h>
#include <XXX.h> /* Header File of Low Level Device Driver */

BOOL32
Example(VOID)
{
    INT32    nErr;
    UINT32   nDev      = 0;
    UINT32   nVol      = 0;
    UINT32   n1stUB    = 0;
    UINT32   nNumOfUBs = 1024;
    LowFuncTbl stLFT[MAX_VOL];

    PAM_RegLFT((VOID *)stLFT);
```

```

nErr = stLFT[nVol].SetRWArea(nDev, n1stUB, nNumOfUBs);

if (nErr != LLD_SUCCESS)
{
    printf("XXX_ SetRWArea()fail. ErrCode = %x\n", nErr);
    return (FALSE32);
}
return (TRUE32);
}

```

(2) Example to implement the function in OneNAND

```

INT32
ONLD_SetRWArea(UINT32 nDev, UINT32 nSubn, UINT32 nUBlks)
{
    INT32    nRes;

    if ((nRes = _ChkPrevOpResult(nDev)) != ONLD_SUCCESS)
    {
        pstPrevOpInfo[nDev]->ePreOp = NONE;
        return nRes;
    }

    if (astONLDInfo[nDev].SetRWArea != NULL)
    {
        astONLDInfo[nDev].SetRWArea(nDev, nSubn, nUBlks);
    }
    else
    {
        return (ONLD_ILLEGAL_ACCESS);
    }

    return (ONLD_SUCCESS);
}

```

SEE ALSO

XXX_IOCtl

DESCRIPTION

This function is called to extend LLD functionality.

SYNTAX

```
INT32
XXX_IOCtl(UINT32 nDev, UINT32 nCode, UINT8 *pBufI,
          UINT32 nLenI, UINT8 *pBufO, UINT32 nLenO,
          UINT32 *pByteRet)
```

PARAMETERS

Parameter	Type	In/Out	Description
nDev	UINT32	In	Physical Device Number (0 ~ 7)
nCode	UINT32	In	IO Control Command
pBufI	UINT8 *	In	Input Buffer pointer
nLenI	UINT32	In	Length of Input Buffer
pBufO	UINT8 *	Out	Output Buffer pointer
nLenO	UINT32	In	Length of Output Buffer
pByteRet	UINT32 *	Out	The number of bytes (length) of Output Buffer as the result of function call

RETURN VALUE

Return Value	Description
LLD_SUCCESS	Success
LLD_IOC_NOT_SUPPORT	In not-supported command

REMARKS

This function is optional.

DCOP(Deferred Check Operation) : a method optimizing the operation performance. The DCOP method is a procedure to terminate the function (write or erase) right after the command issue. Then it checks the result of the operation at next function call.

The following list is IO control code command.

- LLD_IOC_SET_SECURE_LT
- LLD_IOC_SET_BLOCK_LOCK
- LLD_IOC_GET_SECURE_STAT
- LLD_IOC_RESET_NAND_DEV

The following explains the description, parameters, and return value of each IO control code.

LLD_IOC_SET_SECURE_LT

1. Description

If NAND flash memory supports the lock scheme, this command code lock-tighten the current lock statement.

If NAND flash memory does not support the lock scheme, this command returns LLD_ILLEGAL_ACCESS

2. Parameter

Parameter	Description
nDev	Physical Device Number (0 ~ 7)
nCode	LLD_IOC_SET_SECURE_LT
pBufI	Pointer to Start Block Number and Number of Blks if Lock Block device is used.
nLenI	Not used if Lock Range device is used Length of Input Buffer if Lock Block device is used
pBufO	Not used if Lock Range device is used
nLenO	Not used
pByteRet	This value is set as 0 in XXX_IOCTL.

3. Return Value

Return Value	Description
LLD_SUCCESS	Lock-tighten the current lock statement

LLD_IOC_SET_BLOCK_LOCK

1. Description

If NAND flash memory supports the lock scheme, this command code locks the blocks given in pBufI.

2. Parameter

Parameter	Description
nDev	Physical Device Number (0 ~ 7)
nCode	LLD_IOC_SET_SECURE_LT
pBufI	Pointer to Start Block Number and Number of Blks
nLenI	Length of Input Buffer
pBufO	Not used
nLenO	Not used
pByteRet	This value is set as 0 in XXX_IOCTL.

3. Return Value

Return Value	Description
LLD_SUCCESS	Lock the blocks given in pBufI

LLD_IOC_GET_SECURE_STAT

1. Description

If NAND flash memory supports the lock scheme, this command code gets the current lock statement. The current lock statement is saved at pBufO.

If NAND flash memory does not support the lock scheme, the current lock statement is LLD_IOC_SECURE_US.

2. Parameter

Parameter	Description
nDev	Physical Device Number (0 ~ 7)
nCode	LLD_IOC_GET_SECURE_STAT
pBufI	Not used
nLenI	Not used
pBufO	Buffer to store return value
nLenO	Length of pBufO
pByteRet	This value is set as output buffer length in XXX_IOCTL.

3. Return Value

Return Value	Description
LLD_SUCCESS	Get the current lock statement
LLD_ILLEGAL_ACCESS	The value of pBufO or nLenO parameter is abnormal

4. Remark

There are three values of the lock statement. The lock statement is declared in LLD.h. The following describes each lock statement.

LLD_IOC_SECURE_LT is that the locked block of NAND flash memory is lock-tightened.

The block cannot be changed to unlocked or locked by software.

LLD_IOC_SECURE_LS is that all block of NAND flash memory is locked. All block can be read and cannot be written or erased.

LLD_IOC_SECURE_US is that the sequential block of NAND flash memory is unlocked. The unlocked sequential block can be read/written/erased. The locked block keeps the locked statement, so it cannot be read/written/erased.

LLD_IOC_RESET_NAND_DEV

1. Description

This control command resets NAND flash memory.

2. Parameter

Parameter	Description
nDev	Physical Device Number (0 ~ 7)
nCode	LLD_IOC_RESET_NAND_DEV
pBufI	Not used
nLenI	Not used
pBufO	Not used
nLenO	Not used
pByteRet	This value is set as 0 in XXX_IOCTL.

3. Return Value

Return Value	Description
LLD_SUCCESS	Reset NAND flash memory

EXAMPLE

(1) Example to implement the function in OneNAND

```

INT32
ONLD_IOCTL(UINT32 nDev, UINT32 nCode,
            UINT8 *pBufI, UINT32 nLenI,
            UINT8 *pBufO, UINT32 nLenO,
            UINT32 *pByteRet)
{
    INT32 nRet;
    UINT32 nBAddr;
    UINT16 nRegVal;
    INT32 nRes;

    if (pByteRet == NULL)
    {
        return (ONLD_ILLEGAL_ACCESS);
    }

    nBAddr = GET_DEV_BADDR(nDev);

    if ((nRes = _ChkPrevOpResult(nDev)) != ONLD_SUCCESS)
    {
        return nRes;
    }

    switch (nCode)
    {
        case LLD_IOC_SET_SECURE_LT:
    
```

```

        if (astONLDInfo[nDev].LockTight != NULL)
            astONLDInfo[nDev].LockTight(nDev, pBufI, nLenI);
        else
            return (ONLD_ILLEGAL_ACCESS);
        *pByteRet = (UINT32)0;
        nRet = ONLD_SUCCESS;
        break;

    case LLD_IOC_SET_BLOCK_LOCK:

        if (_LockBlock(nDev, pBufI, nLenI) != ONLD_SUCCESS)
        {
            return (ONLD_ILLEGAL_ACCESS);
        }

        *pByteRet = (UINT32)0;
        nRet = ONLD_SUCCESS;
        break;

    case LLD_IOC_GET_SECURE_STAT:
        if ((pBufO == NULL) || (nLenO < 2))
        {
            return (ONLD_ILLEGAL_ACCESS);
        }
        nRegVal = ONLD_REG_WR_PROTECT_STAT(nBAddr);
        MEMCPY(pBufO, &nRegVal, sizeof(UINT16));
        *pByteRet = (UINT32)2;
        nRet = ONLD_SUCCESS;
        break;

    case LLD_IOC_RESET_NAND_DEV:

        if ((GET_DEV_DID(nDev) & 0x0008) == DDP_CHIP)
        {
            ONLD_REG_START_ADDR1(nBAddr) = (UINT16)(0x0000);
            ONLD_REG_START_ADDR2(nBAddr) = (UINT16)(0x0000);

            ONLD_REG_INT(nBAddr) = (UINT16)INT_CLEAR;
            /*-----*/
            /* ONLD Unlock CMD is issued */
            /*-----*/
            ONLD_REG_CMD(nBAddr) = (UINT16)ONLD_CMD_RESET;

            while (GET_ONLD_INT_STAT(nBAddr, PEND_RESET)
                    != (UINT16)PEND_RESET)
            {
                /* Wait until device ready */
            }

            ONLD_REG_START_ADDR1(nBAddr) = (UINT16)(0x8000);
            ONLD_REG_START_ADDR2(nBAddr) = (UINT16)(0x8000);
        } /*if DDP_CHIP */

        ONLD_REG_INT(nBAddr) = (UINT16)INT_CLEAR;

```

```

/*-----*/
/* ONLD Unlock CMD is issued */
/*-----*/
ONLD_REG_CMD(nBAddr) = (UINT16)ONLD_CMD_RESET;

while (GET_ONLD_INT_STAT(nBAddr, PEND_RESET)
      != (UINT16)PEND_RESET)
{
    /* Wait until device ready */
}

*pByteRet = (UINT32)0;
nRet = ONLD_SUCCESS;
break;

default:
    nRet = ONLD_IOC_NOT_SUPPORT;
    break;
}
return nRet;
}

```

SEE ALSO

XXX_GetPrevOpData

DESCRIPTION

This function copies data of previous write operation to the given buffer. This function is called to rewrite the block in the error case of the previous write operation.

SYNTAX

```
INT32
XXX_GetPrevOpData(UINT32 nDev, UINT8 *pMBuf, UINT8 *pSBuf)
```

PARAMETERS

Parameter	Type	In/Out	Description
nDev	UINT32	In	Physical Device Number (0 ~ 7)
pMBuf	UINT8 *	Out	Memory buffer for main array of NAND flash memory
pSBuf	UINT8 *	Out	Memory buffer for spare array of NAND flash memory

RETURN VALUE

Return Value	Description
LLD_SUCCESS	Copy Success
LLD_ILLEGAL_ACCESS	Illegal access

REMARKS

This function is an optional Deferred Check Operation function.

DCOP(Deferred Check Operation) : a method optimizing the operation performance. The DCOP method is a procedure to terminate the function (write or erase) right after the command issue. Then it checks the result of the operation at next function call.

EXAMPLE

(1) Example to call the function

```
#include <XSRTypes.h>
#include <PAM.h>
#include <LLD.h>
#include <XXX.h> /* Header File of Low Level Device Driver */

BOOL32
Example(VOID)
{
    INT32    nErr;
    UINT32   nDev = 0;
    UINT32   nVol = 0;
    UINT8    aMBuf[LLD_MAIN_SIZE];
    UINT8    aSBuf[LLD_SPARE_SIZE];
    LowFuncTbl stLFT[MAX_VOL];
```

```

PAM_RegLFT((VOID *)stLFT);

nErr = stLFT[nVol].GetPrevOpData(nDev, aMBuf, aSBuf);

if (nErr != LLD_SUCCESS)
{
    printf("XXX_GetPrevOpData()fail. ErrCode = %x\n", nErr);
    return (FLASE32);
}
return (TRUE32);
}

```

(2) Example to implement the function in OneNAND

```

INT32
ONLD_GetPrevOpData(UINT32 nDev, UINT8 *pMBuf, UINT8 *pSBuf)
{
    UINT32  nBAddr;
    UINT32  nPbn;
    UINT32  *pONLDMBuf, *pONLDSBuf;

    nBAddr = GET_DEV_BADDR(nDev);

    if ((pMBuf == NULL) && (pSBuf == NULL))
    {
        return (ONLD_ILLEGAL_ACCESS);
    }

    nPbn = GET_PBN(nDev, pstPrevOpInfo[nDev]->nPsn);
    ONLD_REG_START_ADDR2(nBAddr) = (UINT16)MAKE_DBS(nPbn);

    pONLDMBuf = (UINT32*)GET_ONLD_MBUF_ADDR(
                                                nBAddr,
                                                pstPrevOpInfo[nDev]->nPsn,
                                                GET_CUR_BUF_SEL(nDev));
    pONLDSBuf = (UINT32*)GET_ONLD_SBUF_ADDR(
                                                nBAddr,
                                                pstPrevOpInfo[nDev]->nPsn,
                                                GET_CUR_BUF_SEL(nDev));

    /*-----*/
    /* Data is loaded into Memory */
    /*-----*/
    if (pMBuf != NULL)
    {
        /* Memcopy for main data */
        MEMCPY(pMBuf, pONLDMBuf,
                (ONLD_MAIN_SIZE * pstPrevOpInfo[nDev]->nScts));
    }
    if (pSBuf != NULL)
    {
        /* Memcopy for spare data */
        MEMCPY(pSBuf, pONLDSBuf,
                (ONLD_SPARE_SIZE * pstPrevOpInfo[nDev]->nScts));
    }

    return (ONLD_SUCCESS);
}

```



```
}  
}
```

SEE ALSO

XXX_FlushOp

XXX_FlushOp

DESCRIPTION

This function completes the current working operation.

SYNTAX

```
INT32
XXX_FlushOp(UINT32 nDev)
```

PARAMETERS

Parameter	Type	In/Out	Description
nDev	UINT32	In	Physical Device Number (0 ~ 7)

RETURN VALUE

Return Value	Description
LLD_SUCCESS	Success
LLD_ILLEGAL_ACCESS	Illegal Access
LLD_PREV_WRITE_ERROR LLD_PREV_OP_RESULT	In DCOP (Deferred Check Operation), previous write error happens
LLD_PREV_ERASE_ERROR LLD_PREV_OP_RESULT	In DCOP (Deferred Check Operation), previous erase error happens

REMARKS

This function is an optional Deferred Check Operation function.

EXAMPLE

(1) Example to call the function

```
#include <XSRTypes.h>
#include <PAM.h>
#include <LLD.h>
#include <XXX.h> /* Header File of Low Level Device Driver */

BOOL32
Example(VOID)
{
    INT32    nErr;
    UINT32   nDev = 0;
    UINT32   nVol = 0;
    LowFuncTbl stLFT[MAX_VOL];

    PAM_RegLFT((VOID *)stLFT);

    nErr = stLFT[nVol].FlushOp(nDev);

    if (nErr != LLD_SUCCESS)
    {
        printf("XXX_FlushOp()fail. ErrCode = %x\n", nErr);
    }
}
```

```
        return (FALSE32);  
    }  
    return (TRUE32);  
}
```

(2) Example to implement the function in OneNAND

```
INT32  
ONLD_FlushOp(UINT32 nDev)  
{  
    UINT32 nBAddr;  
    INT32 nRes;  
  
    nBAddr = GET_DEV_BADDR(nDev);  
  
    if ((nRes = _ChkPrevOpResult(nDev)) != ONLD_SUCCESS)  
    {  
        return nRes;  
    }  
  
    return (ONLD_SUCCESS);  
}
```

SEE ALSO

XXX_GetPrevOpData

6. PAM (Platform Adaptation Module)

This chapter describes the definition, system architecture, features, and APIs of PAM.

6.1. Description & Architecture

PAM is an abbreviation of Platform Adaptation Module. PAM links XSR with the platform. PAM is responsible for the platform-dependent part of XSR layer (STL, and BML). If the platform is changed, a user only changes PAM.

Reference

Generally, the platform is underlying the computer system on which application program can run. This document calls the platform is the board that consists of CPU, DRAM, NAND flash memory, etc.

For example, a layer of XSR wants to requests the volume and device information of NAND flash memory. The requested information is dependent on the platform. Each layer calls an adaptation module PAM to use the platform functionalities. Therefore, a user must implement PAM suitable for the platform when a user ports XSR.

Figure 6-1 shows PAM in XSR system architecture.

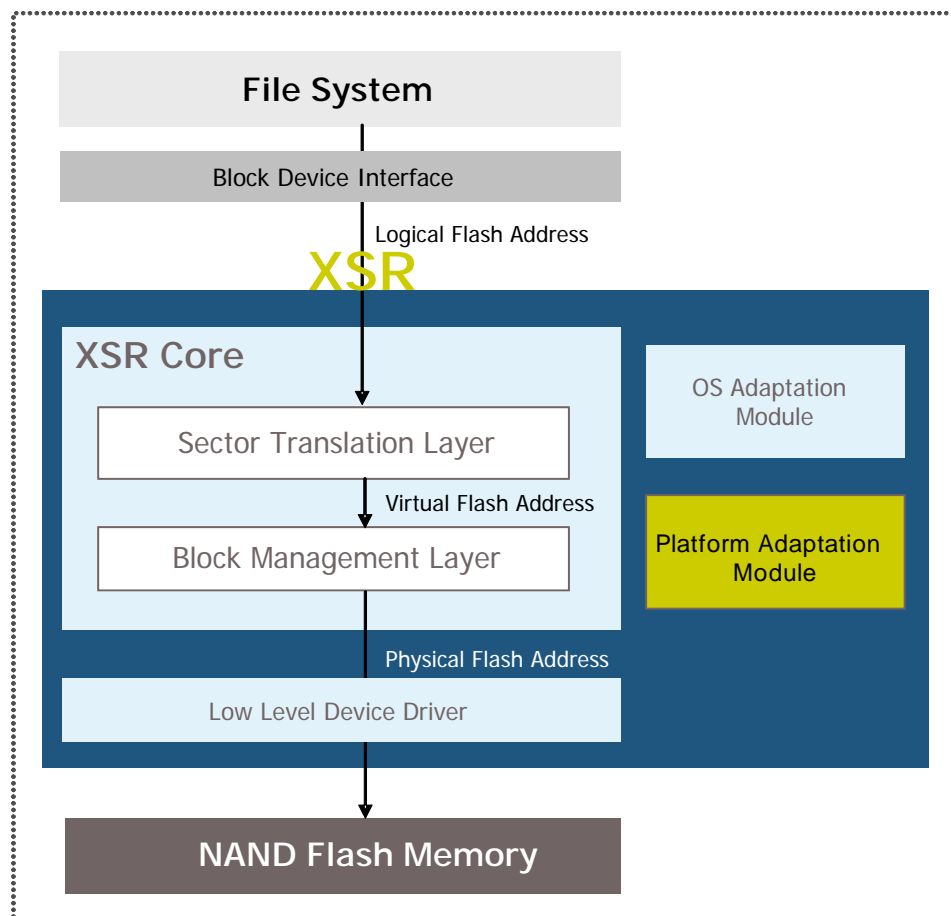


Figure 6-1. PAM in XSR System Architecture

PAM has 9 functions that are classified into 4 categories as follows.

- ☐ **Initialization function** : hardware initialization
- ☐ **Device Configuration functions** : LFT registration, and XSR and LLD information return
- ☐ **Interrupt functions** : interrupt initialization, interrupt bind, interrupt enable, interrupt disable, and interrupt clear
- ☐ **Memory Operation function** : memory copy

In the next chapter, PAM APIs are covered in detail.

6.2. API

This chapter describes PAM APIs.

Reference

All the PAM function has a prefix “PAM_” on each function name.

Table 6-1 shows the lists of PAM APIs.

The right row in table shows that the function is **M**andatory or **O**ptional or **R**ecommended. Optional functions should be existed, but contents of the functions does not need to be implemented.

Table 6-1. PAM API

Function	Description	
PAM_Init	This function initializes NAND specific hardware	O
PAM_GetPAParm	This function maps the platform and device.	M
PAM_RegLFT	This function registers LLD to XSR.	M
PAM_InitInt	This function initializes the interrupt for NAND device.	O
PAM_BindInt	This function binds the interrupt for NAND device.	O
PAM_EnableInt	This function enables the interrupt for NAND device.	O
PAM_DisableInt	This function disables the interrupt for NAND device.	O
PAM_ClearInt	This function clears the interrupt for NAND device.	O
PAM_Memcpy	This function copies data from source to destination	M

PAM_Init

DESCRIPTION

This function initializes NAND specific Hardware.

SYNTAX

```
VOID  
PAM_Init(VOID)
```

PARAMETERS

None

RETURN VALUE

None

REMARKS

This function performs platform specific initialization for allowing access to NAND flash device. If necessary, this function should initialize memory bus width, and memory configuration registers for NAND flash device.

EXAMPLE

(1) Example to call the function

```
#include <PAM.h>  
#include <XSRTypes.h>  
  
VOID  
Example(VOID)  
{  
    /* PAM_Init() should be called at open time */  
    PAM_Init();  
  
    /* LLD Function Table initialization */  
    PAM_RegLFT(gstLFT);  
}
```

(2) Example to implement the function in S3C2410

```
VOID  
PAM_Init(VOID)  
{  
}
```

SEE ALSO

PAM_GetPAParm

DESCRIPTION

This function maps the platform and device.

SYNTAX

```
VOID*
PAM_GetPAParm(VOID)
```

PARAMETERS

None

RETURN VALUE

Return Value	Description
VOID *	Returns the address of the array having information about the volume and device

REMARKS

This function is mandatory.

Currently, XSR supports two volumes and eight devices at maximum, because the number of maximum volume (XSR_MAX_VOL) is defined as 2 and the number of maximum device (XSR_MAX_DEV) is defined as 8 in XSRTypes. When calling PAM_GetPAParm, a user can get the volume and device information together. So, a user gets the platform information by calling PAM_GetPAParm.

XsrVolParm data structure is declared in PAM.h.

□ **XsrVolParm** data structure

```
typedef struct {
    UINT32  nBaseAddr[XSR_MAX_DEV/XSR_MAX_VOL];
    /* the base address for accessing NAND device*/

    UINT16  nEccPol;
    /* Ecc Execution Section
       NO_ECC : No ECC or ECC execution by
               another type of ECC algorithm
       SW_ECC : ECC execution by XSR Software
               (based on Hamming code)
       HW_ECC : ECC execution by HW
               (if HW has ECC functionality
                based on Hamming code) */

    UINT16  nLSchemePol;
    /* Lock Scheme Policy Section
       NO_LOCK_SCHEME : No Lock Scheme
```

```

        SW_LOCK_SCHEME : Lock Scheme Execution by SW
        HW_LOCK_SCHEME : Lock Scheme Execution by HW
                        (if HW has Lock/Unlock functionality)*/

    BOOL32  bByteAlign;
    /* Memory Byte Alignment Problem
       TRUE32 : Byte Align Problem Free in XSR and LLD
               (Memory usage increased)
       FALSE32 : No Action for Byte Align Problem */

    UINT32  nDevsInVol;
    /* number of devices in the volume */

    VOID    *pExInfo;
    /* For Device Extension. For Extra Information of Device,
       data structure can be mapped.          */

} XsrVolParm;

```

More detailed explanation about `XsrVolParm` is as follow.

- **nBaseAddr** is a base address of NAND device for LLD.
- **nEccPol** is a policy whether using ECC code or not: `nEccPol` can be `HW_ECC`, `SW_ECC` and `NO_ECC`. A user sets as `HW_ECC` when NAND device provides ECC generation/correction based on Hamming code. In that case, spare assignment for generated ECC code should be compatible with Samsung spare assignment standard. A user sets as `NO_ECC` when NAND device uses no ECC algorithm or hardware ECC algorithm which is not compatible with Samsung standards⁶. When a user wants to use software ECC supported by XSR and sets as `SW_ECC`, BML handles ECC generation/correction. For more information about the ECC policy, refer to Chapter 7.6 ECC Policy.
- **nLSchemePol** is a policy related to write/erase protection: `nLSchemePol` can be `HW_LOCK_SCHEME`, `SW_LOCK_SCHEME`, and `NO_LOCK_SCHEME`. A user sets as `NO_LOCK_SCHEME` when NAND device does not provide the lock scheme, and sets as `HW_LOCK_SCHEME` when NAND device use the lock scheme. When a user wants to use the lock scheme in software and sets as `SW_LOCK_SCHEME`, BML handles the write/erase protection.
- **bByteAlign** is used to decide whether LLD fits the alignment or not, when LLD gets not-aligned buffer from the upper layer in read/write operation. A users set as `TRUE32` that means to fit the alignment, while a user sets as `FALSE32` that means to hand it over LLD, although it is not-aligned.
- **nDevsInVol** is the number of the allocated device in the volume.
- **pExInfo** is entry for extension usage. It is available for developers who want to add their own platform dependent information.

⁶ Memory Division, Samsung Electronics Co., Ltd, "ECC(Error Checking & Correction) Algorithm", http://www.samsung.com/Products/Semiconductor/Flash/TechnicalInfo/eccalgo_040624.pdf

Memory Division, Samsung Electronics Co., Ltd, "NAND Flash Spare Assignment recommendation", http://www.samsung.com/Products/Semiconductor/Flash/TechnicalInfo/Spare_assignment_recommendation.pdf

EXAMPLE

(1) Example to call the function

```
#include <PAM.h>
#include <XSRTypes.h>

VOID
Example(VOID)
{
    XsrVolParm *pstPAM;

    pstPAM = (XsrVolParm *) PAM_GetPAParm();
}
```

(2) Example to implement the function in S3C2410

```
#include <XSRTypes.h>
#include <PAM.h>
#include <XXX.h> /* Header File of Low Level Device Driver */

#define VOL0 0
#define VOL1 1

#define DEV0 0
#define DEV1 1
#define DEV2 2
#define DEV3 3

VOID*
PAM_GetPAParm(VOID)
{
    gstParm[VOL0].nBaseAddr[DEV0] = 0x20000000;
    gstParm[VOL0].nBaseAddr[DEV1] = NOT_MAPPED;
    gstParm[VOL0].nBaseAddr[DEV2] = NOT_MAPPED;
    gstParm[VOL0].nBaseAddr[DEV3] = NOT_MAPPED;

    gstParm[VOL0].nEccPol          = SW_ECC;
    gstParm[VOL0].nLSchemePol     = SW_LOCK_SCHEME;
    gstParm[VOL0].bByteAlign      = TRUE32;
    gstParm[VOL0].nDevsInVol      = 1;
    gstParm[VOL0].pExInfo         = NULL;

    gstParm[VOL1].nBaseAddr[DEV0] = NOT_MAPPED;
    gstParm[VOL1].nBaseAddr[DEV1] = NOT_MAPPED;
    gstParm[VOL1].nBaseAddr[DEV2] = NOT_MAPPED;
    gstParm[VOL1].nBaseAddr[DEV3] = NOT_MAPPED;

    gstParm[VOL1].nEccPol          = HW_ECC;
    gstParm[VOL1].nLSchemePol     = HW_LOCK_SCHEME;
    gstParm[VOL1].bByteAlign      = TRUE32;
    gstParm[VOL1].nDevsInVol      = 0;
    gstParm[VOL1].pExInfo         = NULL;

    return (VOID *) gstParm;
}
```



SEE ALSO

PAM_RegLFT

DESCRIPTION

This function registers functions to XSR.

SYNTAX

```
VOID
PAM_RegLFT(VOID *pstFunc)
```

PARAMETERS

Parameter	Type	In/Out	Description
pstFunc	VOID *	Out	Pointer to LowFuncTbl data structure

RETURN VALUE

None

REMARKS

This function is mandatory.

1. Registering LLD address to BML

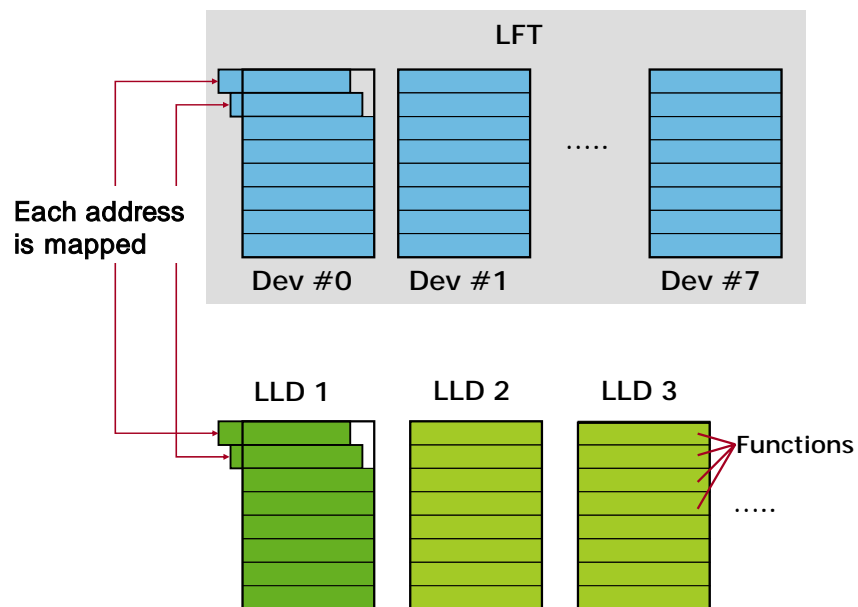


Figure 6-2. Register LLD Address to BML

BML encapsulates the various device driver(LLD)s. LFT(LLD Function Table) is a list of 17 functions that is encapsulated by BML. LFT is defined at LLD.h as LowFuncTbl.

□ LowFuncTbl data structure

```
typedef struct {
    INT32 (*Init)      (VOID *pParm);
    INT32 (*Open)      (UINT32 nDev);
    INT32 (*Close)     (UINT32 nDev);
    INT32 (*Read)      (UINT32 nDev,  UINT32 nPsn,  UINT32 nScts,
                        UINT8 *pMBuf,  UINT8 *pSBuf,  UINT32 nFlag);
    INT32 (*Write)     (UINT32 nDev,  UINT32 nPsn,  UINT32 nScts,
                        UINT8 *pMBuf,  UINT8 *pSBuf,  UINT32 nFlag);
    INT32 (*MRead)     (UINT32 nDev,  UINT32 nPsn,  UINT32 nScts,
                        SGL *pstSGL,  UINT8 *pSBuf,  UINT32 nFlag);
    INT32 (*MWrite)    (UINT32 nDev,  UINT32 nPsn,  UINT32 nScts,
                        SGL *pstSGL,  UINT8 *pSBuf,  UINT32 nFlag,
                        UINT32 *pErrPsn);
    INT32 (*CopyBack)  (UINT32 nDev,  CpBkArg *pstCpArg,
                        UINT32 nFlag);
    INT32 (*Erase)     (UINT32 nDev,  UINT32 Pbn,  UINT32 nFlag);
    INT32 (*GetDevInfo)(UINT32 nDev,  LLDSpec *pstDevInfo);
    INT32 (*ChkInitBadBlk)(UINT32 nDev,  UINT32 Pbn);
    INT32 (*FlushOp)  (UINT32 nDev);
    INT32 (*SetRWArea)(UINT32 nDev,  UINT32 nSUBn,  UINT32 nUBlks);
    INT32 (*GetPrevOpData)(UINT32 nDev,  UINT8 *pMBuf,
                           UINT8 *pSBuf);
    INT32 (*IOctl)    (UINT32 nDev,  UINT32 nCmd,
                        UINT8 *pBufI,  UINT32 nLenI,
                        UINT8 *pBufO,  UINT32 nLenO,
                        UINT32 *pByteRet);
    INT32 (*MErase)   (UINT32 nDev,  LLDMEArg *pstMEArg,
                        UINT32 nFlag);
    INT32 (*EraseVerify)(UINT32 nDev,  LLDMEArg *pstMEArg,
                        UINT32 nFlag);
} LowFuncTbl;
```

LowFuncTbl is allocated corresponding to each device and is able to support up to 8. Thus, it is necessary to register the function of real LLD to the corresponding LowFuncTbl.

XSR can work together by calling PAM_RegLFT and registering the implemented LLD function. BML can call the real LLD function using the function pointer defined at LFT.

pstFunc is a pointer of LowFuncTbl, a LLD address table.

When BML calls PAM_RegLFT, it finds the real function address to access NAND device using LFT.

2. Defining the volume and device of BML

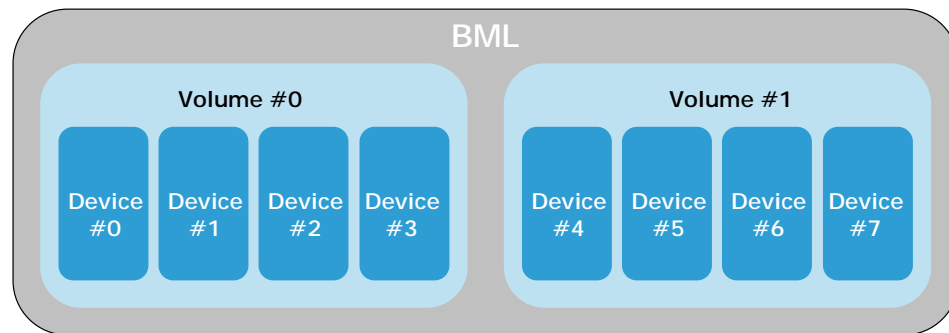


Figure 6-3. Define Volume and Device of BML

A user must recognize the definition of a volume and device to use BML. XSR can support the device up to eight, and BML can manage several devices once bind them with a volume.

BML can support the plural volumes, but it currently supports 2 volumes. BML binds the maximum four devices with a volume, and the devices must be the same type in a volume. For example, the device number #0 ~ #3 and #4 ~ #7 must be the same type of LLD.

A user should know this volume and device features at BML, so a user can understand the mapping from BML to LLD using LFT.

EXAMPLE

(1) Example to call the function

```
#include <PAM.h>
#include <XSRTypes.h>

VOID
Example(VOID)
{
    PAM_Init();

    /* LLD Function Table initialization */
    PAM_RegLFT(gstLFT);
}
```

(2) Example to implement the function in S3C2410

```
#include <XSRTypes.h>
#include <PAM.h>
#include <XXX.h> /* Header File of Low Level Device Driver */

#define VOL0 0

VOID
PAM_RegLFT(VOID *pstFunc)
{
    LowFuncTbl *pstLFT;
```



```

pstLFT = (LowFuncTbl*)pstFunc;

pstLFT[VOL0].Init           = ONLD_Init;
pstLFT[VOL0].Open          = ONLD_Open;
pstLFT[VOL0].Close         = ONLD_Close;
pstLFT[VOL0].Read          = ONLD_Read;
pstLFT[VOL0].Write         = ONLD_Write;
pstLFT[VOL0].CopyBack      = ONLD_CopyBack;
pstLFT[VOL0].Erase         = ONLD_Erase;
pstLFT[VOL0].GetDevInfo    = ONLD_GetDevInfo;
pstLFT[VOL0].ChkInitBadBlk = ONLD_ChkInitBadBlk;
pstLFT[VOL0].FlushOp       = ONLD_FlushOp;
pstLFT[VOL0].SetRWArea     = ONLD_SetRWArea;
pstLFT[VOL0].GetPrevOpData = ONLD_GetPrevOpData;
pstLFT[VOL0].IOctl         = ONLD_IOCTL;
pstLFT[VOL0].MRead         = ONLD_MRead;
pstLFT[VOL0].MWrite        = ONLD_MWrite;
pstLFT[VOL0].MErase        = ONLD_MErase;
pstLFT[VOL0].EraseVerify   = ONLD_EraseVerify;
}

```

SEE ALSO

PAM_InitInt

DESCRIPTION

This function initializes the specified logical interrupt.

SYNTAX

```
VOID
PAM_InitInt(UINT32 nLogIntId)
```

PARAMETERS

Parameter	Type	In/Out	Description
nLogIntId	UINT32	In	Logical interrupt ID number

RETURN VALUE

None

REMARKS

This function is an optional interrupt function.
Currently, this function is used for asynchronous operation.

This function set registers to use interrupts of platform. This function initializes the specified logical interrupt through OS dependent interrupt initialization function.

EXAMPLE

(1) Example to call the function

```
#include <PAM.h>
#include <XSRTypes.h>

#define INT_ID_NAND_0 (0) /* Interrupt ID : 1st NAND */

VOID
Example(VOID)
{
    /* initializes interrupt for 1st NAND device */
    PAM_InitInt( (UINT32)INT_ID_NAND_0 );

    /* initializes interrupt for 1st NAND device */
    PAM_BindInt( (UINT32)INT_ID_NAND_0 );
}
```

(2) Example to implement the function in S3C2410

```
VOID
PAM_InitInt(UINT32 nLogIntId)
{
    switch (nLogIntId)
    {
        case XSR_INT_ID_NAND_0:
```



```
        TS3C2440::GPIO::SelectSignalMethod(EExtInt4,  
            ERisingEdgeSignal);  
        break;  
  
        default:  
            break;  
    }  
  
    OAM_InitInt(nLogIntId);  
}
```

SEE ALSO

PAM_BindInt, PAM_EnableInt, PAM_DisableInt, PAM_ClearInt

PAM_BindInt

DESCRIPTION

This function binds the specified logical interrupt for NAND device.

SYNTAX

```
VOID
PAM_BindInt(UINT32 nLogIntId)
```

PARAMETERS

Parameter	Type	In/Out	Description
nLogIntId	UINT32	In	Logical interrupt ID number

RETURN VALUE

None

REMARKS

This function is an optional interrupt function.
Currently, this function is used for asynchronous operation.

This function translates the specified logical interrupt ID into the physical interrupt ID and binds the interrupt through OS dependent function which binds interrupt.

EXAMPLE

(1) Example to call the function

```
#include <PAM.h>
#include <XSRTypes.h>

#define    INT_ID_NAND_0    (0)    /* Interrupt ID : 1st NAND */

VOID
Example(VOID)
{
    /* initializes interrupt for 1st NAND device */
    PAM_InitInt( (UINT32)INT_ID_NAND_0 );

    /* initializes interrupt for 1st NAND device */
    PAM_BindInt( (UINT32)INT_ID_NAND_0 );
}
```

(2) Example to implement the function in S3C2410

```
VOID
PAM_BindInt(UINT32 nLogIntId)
{
    UINT32 nPhyIntId = 0;

    switch (nLogIntId)
```



```
{
    case XSR_INT_ID_NAND_0:
        nPhyIntId = EIrqExt4_7;
        break;

    default:
        break;
}

OAM_BindInt(nLogIntId, nPhyIntId);
}
```

SEE ALSO

PAM_InitInt, PAM_EnableInt, PAM_DisableInt, PAM_ClearInt

PAM_EnableInt

DESCRIPTION

This function enables the specified logical interrupt for NAND device.

SYNTAX

```
VOID
PAM_EnableInt(UINT32 nLogIntId)
```

PARAMETERS

Parameter	Type	In/Out	Description
nLogIntId	UINT32	In	Logical interrupt ID number

RETURN VALUE

None

REMARKS

This function is an optional interrupt function.
Currently, this function is used for asynchronous operation.

This function translates the specified logical interrupt ID into the physical interrupt ID and enables the interrupt through OS dependent function which enables interrupt.

EXAMPLE

(1) Example to call the function

```
#include <PAM.h>
#include <XSRTypes.h>

#define INT_ID_NAND_0 (0) /* Interrupt ID : 1st NAND */

VOID
Example(VOID)
{
    PAM_ClearInt((UINT32)INT_ID_NAND_0);
    PAM_EnableInt((UINT32)INT_ID_NAND_0);
}
```

(2) Example to implement the function in S3C2410

```
VOID
PAM_EnableInt(UINT32 nLogIntId)
{
    UINT32 nPhyIntId = 0;

    switch (nLogIntId)
    {
        case XSR_INT_ID_NAND_0:
            nPhyIntId = EIrqExt4_7;
```



```
        break;

        default:
            break;
    }

    OAM_EnableInt(nLogIntId, nPhyIntId);
}
```

SEE ALSO

PAM_InitInt, PAM_BindInt, PAM_DisableInt, PAM_ClearInt

PAM_DisableInt

DESCRIPTION

This function disables the specified logical interrupt for NAND device.

SYNTAX

```
VOID
PAM_DisableInt(UINT32 nLogIntId)
```

PARAMETERS

Parameter	Type	In/Out	Description
nLogIntId	UINT32	In	Logical interrupt ID number

RETURN VALUE

None

REMARKS

This function is an optional interrupt function.
Currently, this function is used for asynchronous operation.

This function translates the specified logical interrupt ID into the physical interrupt ID and disables the interrupt through OS dependent function which disables interrupt.

EXAMPLE

(1) Example to call the function

```
#include <PAM.h>
#include <XSRTypes.h>

#define INT_ID_NAND_0 (0) /* Interrupt ID : 1st NAND */

VOID
Example(VOID)
{
    PAM_ClearInt((UINT32)INT_ID_NAND_0);
    PAM_DisableInt((UINT32)INT_ID_NAND_0);
}
```

(2) Example to implement the function in S3C2410

```
VOID
PAM_DisableInt(UINT32 nLogIntId)
{
    UINT32 nPhyIntId = 0;

    switch (nLogIntId)
    {
        case XSR_INT_ID_NAND_0:
            nPhyIntId = EIrqExt4_7;
```




```
        break;

        default:
            break;
    }

    OAM_DisableInt(nLogIntId, nPhyIntId);
}
```

SEE ALSO

PAM_InitInt, PAM_BindInt, PAM_EnableInt, PAM_ClearInt

PAM_ClearInt

DESCRIPTION

This function clears the specified logical interrupt for NAND device.

SYNTAX

```
VOID
PAM_ClearInt(UINT32 nLogIntId)
```

PARAMETERS

Parameter	Type	In/Out	Description
nLogIntId	UINT32	In	Logical interrupt ID number

RETURN VALUE

None

REMARKS

This function is an optional interrupt function.
Currently, this function is used for asynchronous operation.

This function translates the specified logical interrupt ID into the physical interrupt ID and clears the interrupt through OS dependent function which clears interrupt.

EXAMPLE

(1) Example to call the function

```
#include <PAM.h>
#include <XSRTypes.h>

#define INT_ID_NAND_0 (0) /* Interrupt ID : 1st NAND */

VOID
Example(VOID)
{
    PAM_ClearInt((UINT32)INT_ID_NAND_0);
    PAM_EnableInt((UINT32)INT_ID_NAND_0);
}
```

(2) Example to implement the function in S3C2410

```
VOID
PAM_ClearInt(UINT32 nLogIntId)
{
    UINT32 nPhyIntId = 0;

    switch (nLogIntId)
    {
        case XSR_INT_ID_NAND_0:
            nPhyIntId = EIrqExt4_7;
```



```
        break;

        default:
            break;
    }

    OAM_ClearInt(nLogIntId, nPhyIntId);
}
```

SEE ALSO

PAM_InitInt, PAM_BindInt, PAM_EnableInt, PAM_DisableInt

PAM_Memcpy

DESCRIPTION

This function copies data from the source to the destination.

SYNTAX

```
VOID
PAM_Memcpy(VOID *pDst, VOID *pSrc, UINT32 nLen)
```

PARAMETERS

Parameter	Type	In/Out	Description
pDst	VOID *	Out	Array Pointer of destination data to be copied
pSrc	VOID *	In	Array Pointer of source data to be copied
nLen	UINT32	In	Length to be copied

RETURN VALUE

None

REMARKS

This function is a mandatory memory operation function.

This function is called by the function that wants to copy data in the source buffer to data in the destination buffer. If system can support a functionality for memory copy by hardware, PAM_Memcpy uses it to adjust the memory copy operation to specific environment of platform. If not, PAM_Memcpy just calls OS dependent Memcpy function.

EXAMPLE

(1) Example to call the function

```
#include <PAM.h>
#include <XSRTypes.h>

UINT8 SrcBuf[512];
UINT8 DstBuf[512];

VOID
Example(VOID)
{
    PAM_Memcpy(&DstBuf[0], &SrcBuf[0], (512));
}
```

(2) Example to implement the function in S3C2410

```
VOID
PAM_Memcpy(VOID *pDst, VOID *pSrc, UINT32 nLen)
{
    OAM_Memcpy(pDst, pSrc, nLen);
}
```

(3) Example to implement the function in O5912

```

VOID
PAM_Memcpy(VOID *pDst, VOID *pSrc, UINT32 nLen)
{
    if (nLen > 0xffff) return;

    DMA_CSSA_L = (unsigned short)((unsigned int)
                                   (src)&0x0000FFFF);
    DMA_CSSA_U = (unsigned short)((unsigned int)
                                   (src)&0xFFFF0000)>>16);
    DMA_CDSA_L = (unsigned short)((unsigned int)
                                   (dst)&0x0000FFFF);
    DMA_CDSA_U = (unsigned short)((unsigned int)
                                   (dst)&0xFFFF0000)>>16);

    DMA_CEN = byte_len >> 1;    /* channel element number */

    DMA_CCR = 0x54c0;

    /* poll if dma finished */
    while (!(DMA_CSR&0x8))
    {
        //polling if DMA finished
    }
    return;
}

```

SEE ALSO

7. Advanced Topics

This chapter describes Advanced Topics to port XSR: Semaphore, Interrupt, Timer, Deferred Check Operation, Byte Alignment Restrictions and ECC Policy.

7.1. Semaphore

XSR OAM has 4 semaphore functions. This section describes a role of OAM semaphore functions and implementation method of these functions.

If you use XSR on multi-process or multi-task environment, read follows.
Otherwise, skip chapter 7.1 and just leave OAM semaphore functions as template.

7.1.1. Backgrounds

XSR is designed to support multi-process environment partially. In multi-process environment, concurrent requests may be issued to XSR. Each layer of XSR has different ability to manage this situation.

STL supports multiple volumes and does not support multiple accesses for one volume. STL assumes there is only one request for one volume. That is, STL can not handle simultaneous requests for one volume. However for different volumes, STL functions are reenterable. Normally this is enough to STL because STL is used by File System, and generally File System handles these simultaneous requests for one volume.

The condition of BML is somewhat different with the condition of STL.

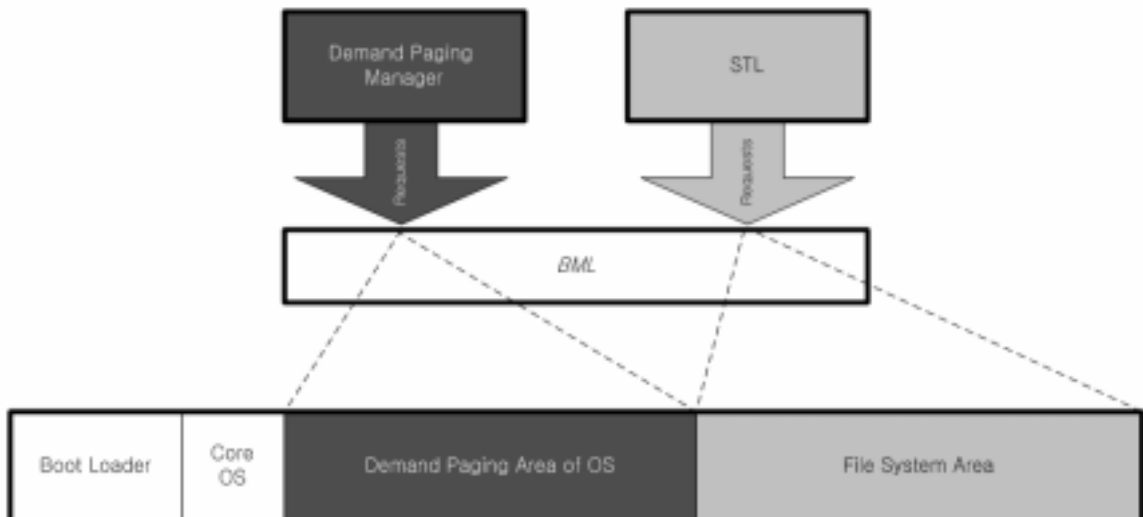


Figure 7-1. Simultaneous Requests to BML

There is an example of NAND device volume in Figure 7-1. The example volume has 4 different areas; boot loader area, Core OS area, OS demand paging area, and File System

area.

There is a demand paging manager which loads proper OS image from NAND device. For example, PocketPC2003 kernel supports this style of demand paging. BML is used by both STL and demand paging manager concurrently; that means BML receives simultaneous requests for same volume. BML is designed to handle these multiple requests for one volume by using semaphore.

If you use multi-task or multi-process environment, there is another situation that BML may receive multiple requests for one volume. In that case, you should implement OAM semaphore functions.

Table 7-1. XSR Multiple volume/device supporting

Supporting	STL	BML
Multiple requests for different volumes	O	O
Multiple requests for same volume	X	O
Use semaphore internally	X	O
Multiple requests for different devices	-	-

In conclusion, semaphore is used by BML to handle multiple requests to NAND devices.

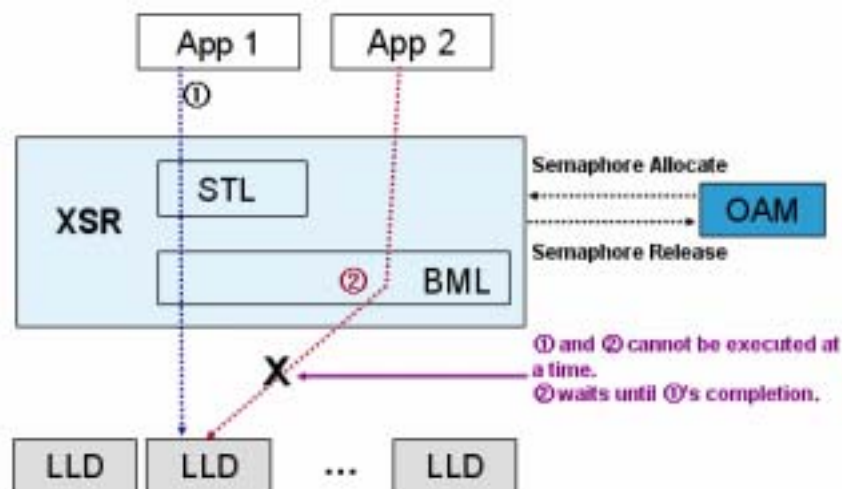


Figure 7-2. Semaphore Operation in BML

7.1.2. Semaphore

Semaphore is traditional IPC(InterProcess Communication) component used in OS. Semaphore is used for two major purposes; shared resources protection and synchronization.

First, semaphore is used to protect shared resources such as memory, device, global variable, program code, etc. This protection can include mutual exclusion and critical section concept. Second, semaphore is used for synchronization. In second case, semaphore can be used like an event or a signal.

In XSR, semaphore is used by BML, to protect device and BML's internal data structure from multiple requests.

When semaphore is used for protection, it is regarded as a key. If one process wants to access shared resource, the process should get a key (admission) before starting shared resource access. Semaphore has internal counter which is called as a token. Token means the number of keys. Semaphore acquiring function is trying to get a key. If semaphore token value is bigger than zero, token is decreased and then calling process is returned from semaphore acquiring function. If semaphore token value is zero, calling process should wait until another process releases semaphore. The process which owns semaphore returns semaphore token by semaphore releasing function.

Most OS have semaphore APIs. These semaphore APIs consist of traditional 4 semaphore functions; create, destroy, acquire, and release.

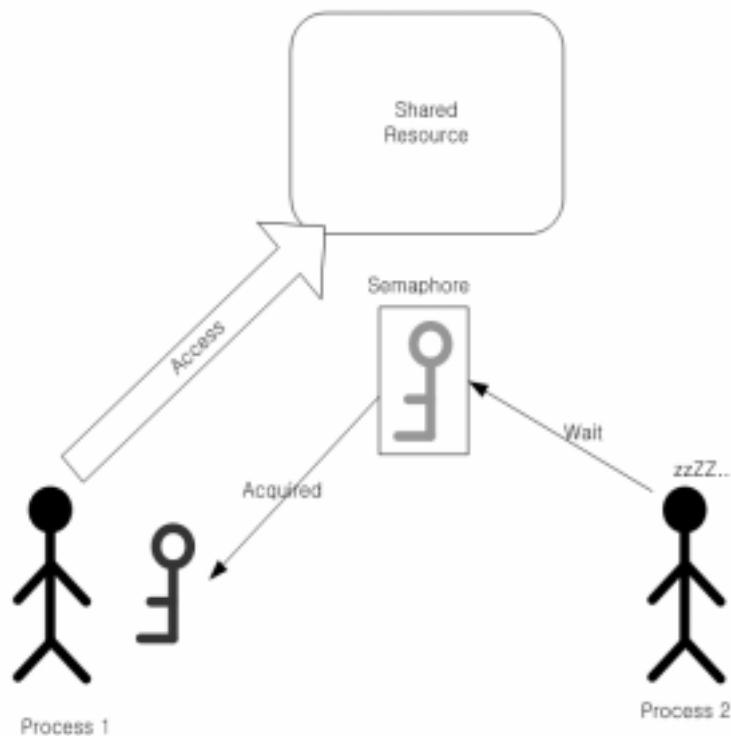


Figure 7-3. Semaphore Usage Example

Figure 7-3 shows semaphore usage example. Both Process1 and Process2 want to access shared resource. To access shared resource, the process should get semaphore first. In figure, Process1 first acquires semaphore, and then accesses shared resource. However Process2 should wait for semaphore released because semaphore is already allocated to Process1. After finishing access of shared resource, Process1 will release semaphore and then Process2 will be able to access shared resource.

7.1.3. Implementation

XSR OAM has 4 semaphore functions as follows.

```
BOOL32 OAM_CreateSM(SM32 *pHandle)
```



```

BOOL32 OAM_DestroySM(SM32 nHandle)
BOOL32 OAM_AcquireSM(SM32 nHandle)
BOOL32 OAM_ReleaseSM(SM32 nHandle)

```

If you use multi-process environment and multiple requests can be given to BML, you should implement OAM semaphore functions. In other words, if you are using NAND device volume only for File System or you do not use multi-process environment, then you do not need to implement OAM semaphore functions. However, OAM semaphore functions should return TRUE32 because BML always calls these functions internally.

XSR defines a type SM32 for semaphore handle. This type is same with UINT32 (32 bit unsigned int type). This handle is set from OAM_CreateSM() and used for other semaphore functions to point the created semaphore.

7.1.3.1. OAM_CreateSM() Implementation

OAM_CreateSM() creates semaphore object.

In current version of XSR, you should remember that BML assumes that created semaphore token is zero. Because of this, BML calls OAM_ReleaseSM() just after semaphore creating. Therefore, OAM_CreateSM() should create semaphore to have initial zero token value. Otherwise, semaphore can not protect NAND device properly.

OAM_CreateSM() returns TRUE32 or FALSE32. When semaphore is successfully created, OAM_CreateSM() should return TRUE32. Otherwise, OAM_CreateSM() returns FALSE32.

(1) OAM_CreateSM() implementation example for pSOS

BOOL32

```

OAM_CreateSM(SM32 *pHandle)
{
    BOOL32 bRet = TRUE32;

    if (sm_create("XSRS", 0, SM_LOCAL | SM_FIFO | SM_UNBOUNDED,
        (ULONG *)pHandle) != 0)
    {
        bRet = FALSE32;
    }

    return bRet;
}

```

(2) OAM_CreateSM() implementation example for WIN32

```

BOOL32
OAM_CreateSM(SM32 *pHandle)
{
    HANDLE hSm;

    hSm = CreateSemaphore(
        (LPSECURITY_ATTRIBUTES) NULL, (LONG) 0, (LONG) 1,
        (LPCTSTR) NULL);

    if ((UINT32) hSm == 0)
        return FALSE32;
}

```

```

        *pHandle = (SM32) hSm;

        return TRUE32;
    }

```

7.1.3.2. OAM_DestroySM() Implementation

OAM_DestroySM() destroys semaphore object.

OAM_DestroySM() returns TRUE32 or FALSE32. When semaphore is successfully destroyed, OAM_DestroySM() should return TRUE32. Otherwise, OAM_DestroySM() returns FALSE32.

(1) OAM_DestroySM() implementation example for pSOS

```

BOOL32
OAM_DestroySM(SM32 nHandle)
{
    BOOL32 bRet = TRUE32;

    if (sm_delete(nHandle) != 0)
    {
        bRet = FALSE32;
    }

    return bRet;
}

```

(2) OAM_DestroySM() implementation example for WIN32

```

BOOL32
OAM_DestroySM(SM32 nHandle)
{
    if (CloseHandle((HANDLE) nHandle) == TRUE)
        return TRUE32;

    return FALSE32;
}

```

7.1.3.3. OAM_AcquireSM() Implementation

OAM_AcquireSM() gets semaphore with created semaphore handle. For this acquiring process, most OS provide several policies for semaphore waiting. When semaphore token is zero, calling process may wait or return error. That wait can be time-outed. However, for OAM_AcquireSM() implementation you do not need to care about these options. Just implement to wait until semaphore is available.

OAM_AcquireSM() returns TRUE32 or FALSE32. When semaphore is successfully acquired, OAM_AcquireSM() should return TRUE32. Otherwise, OAM_DestroySM() returns FALSE32.

(1) OAM_AcquireSM() implementation example for pSOS

```

BOOL32
OAM_AcquireSM(SM32 nHandle)

```

```
{
    BOOL32 bRet = TRUE32;
    if (sm_p(nHandle, SM_WAIT, 0) != 0)
    {
        bRet = FALSE32;
    }

    return bRet;
}
```

(2) **OAM_AcquireSM()** implementation example for WIN32

```
BOOL32
OAM_AcquireSM(SM32 nHandle)
{
    DWORD nRe;

    nRe = WaitForSingleObject((HANDLE) nHandle, INFINITE);
    if (nRe == WAIT_FAILED)
        return FALSE32;

    return TRUE32;
}
```

7.1.3.4. **OAM_ReleaseSM()** Implementation

OAM_ReleaseSM() releases semaphore with created semaphore handle.

OAM_ReleaseSM() returns TRUE32 or FALSE32. When semaphore is successfully released, OAM_ReleaseSM() should return TRUE32. Otherwise, OAM_ReleaseSM() returns FALSE32.

(1) **OAM_ReleaseSM()** implementation example for pSOS

```
BOOL32
OAM_ReleaseSM(SM32 nHandle)
{
    BOOL32 bRet = TRUE32;

    if (sm_v(nHandle) != 0)
    {
        bRet = FALSE32;
    }

    return bRet;
}
```

(2) **OAM_ReleaseSM()** implementation example for WIN32

```
BOOL32
OAM_ReleaseSM(SM32 nHandle)
{
    if (ReleaseSemaphore((HANDLE) nHandle, 1, NULL) == TRUE)
        return TRUE32;

    return FALSE32;
}
```

7.2. Interrupt

XSR OAM has 5 interrupt functions. This section describes background of XSR interrupt and implementation method of these functions.

7.2.1. Interrupt of Device Driver Concept

Device driver operates hardware device such as keyboard, mouse, harddisk, NAND memory device, etc. Host processor and hardware device have host - client relationship. In this relationship, host processor can send data and command to device easily. But, it is not easy to send an event from hardware device to host processor.

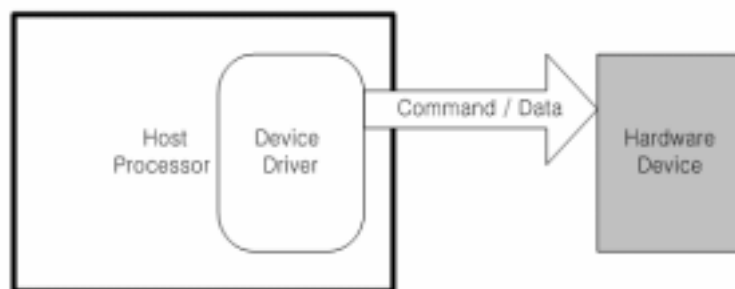


Figure 7-4. Device Driver Concept

Interrupt is a traditional, useful method when hardware sends an event to host. For example, when keyboard key is pushed by a user, keyboard controller invokes interrupt to host process.

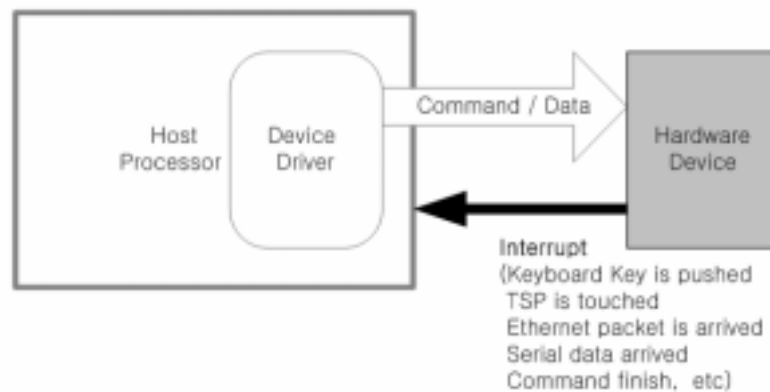


Figure 7-5. Interrupt of Hardware Device

Interrupt is often used to inform that given command is finished by hardware device. NAND device provides this type of interrupt.

7.2.2. Synchronous / Asynchronous Device Driver Model

Depending on the interrupt usage, device driver is classified into synchronous and asynchronous device driver.

Synchronous device driver does not use interrupt. After issuing command to hardware device, host should wait until the command ends. To know end time, host checks status register of hardware device periodically. This periodic checking is called as **polling**.

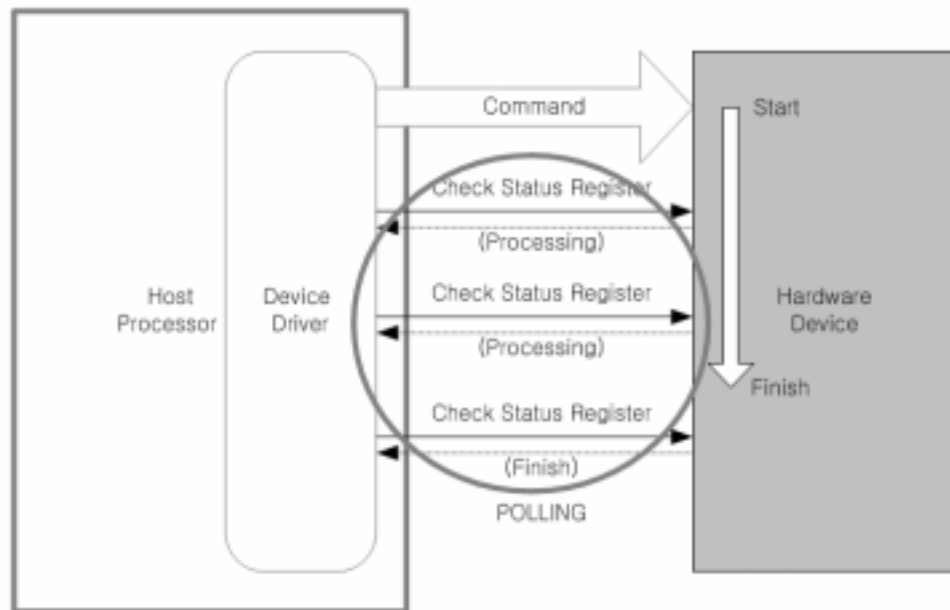


Figure 7-6. Polling

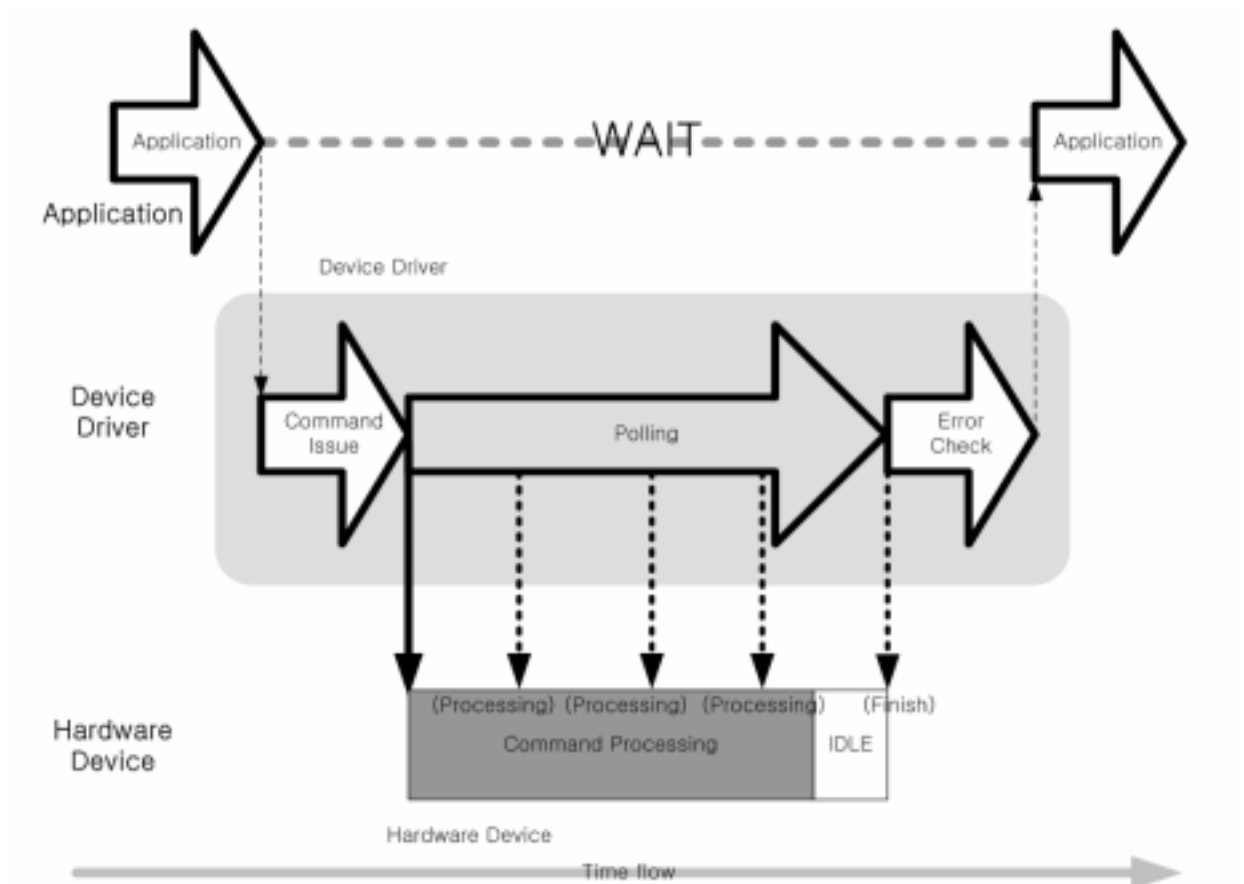


Figure 7-7. Synchronous Device Driver Operation

Polling mode device driver is easy to implement but not efficient because application should wait while device is processing the command. Especially in multi-process OS environment, synchronous device driver model is not good because there may be another process wants to be executed.

Asynchronous device driver uses interrupt to avoid useless waiting. Device driver returns just after issuing command then application gets CPU control. Hardware device processes given command and invokes interrupt to host device driver when given command is over. Interrupt service routine of device driver receives CPU control by interrupt, then checks error and reports proper result to application.

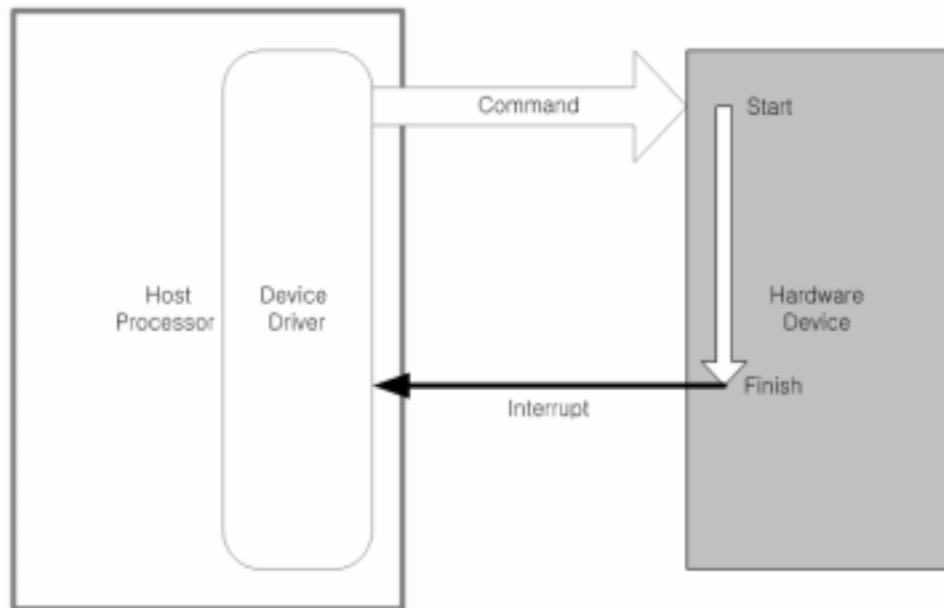


Figure 7-8. Interrupt Usage at Device Driver

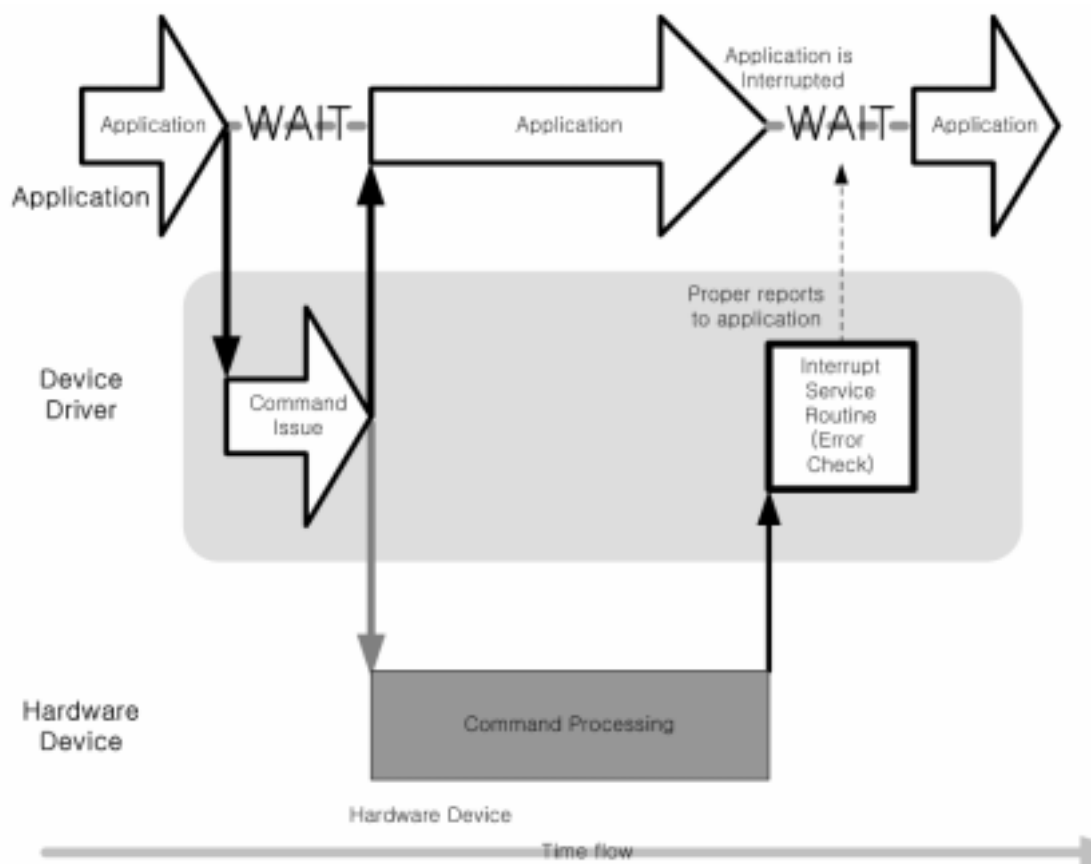


Figure 7-9. Asynchronous Device Driver Operation

Asynchronous model device driver is more efficient than synchronous model driver. However, hardware device must have interrupt function and device driver must include interrupt functions and interrupt service routine.

7.2.3. BML and Interrupt

It takes long time to operate `BML_Write()`, `BML_Erase()` and `BML_Copyback()`. Small block NAND's typical page write time is 200 microsecond(μ s), and typical block erase time is 2 millisecond(ms). Therefore XSR BML provides a method to use asynchronous device driver model. You can use `BML_FLAG_ASYNC_OP` flag on `nFlag` argument of BML functions. If you use `BML_FLAG_ASYNC_OP` when you are issuing BML commands, it is passed to LLD. LLD receives `LLD_FLAG_ASYNC_OP` option for BML's `BML_FLAG_ASYNC_OP`. So, if you want to use asynchronous device driver model at XSR LLD then you should process properly `LLD_FLAG_ASYNC_OP` in your LLD.

However there is a serious problem when you implement asynchronous model LLD. The problem is error processing. If LLD receives `LLD_FLAG_ASYNC_OP`, LLD goes back just after issuing NAND command. Because at this time LLD can not know error status, LLD can not return proper error code. After that, interrupt service routine can recognize error situation but there is no BML API to handle this error for interrupt service routine. To solve this problem, you should use Deferred Check Operation model. In this model, previous operation error is processed at next operation. So interrupt service routine does not need to handle NAND error.

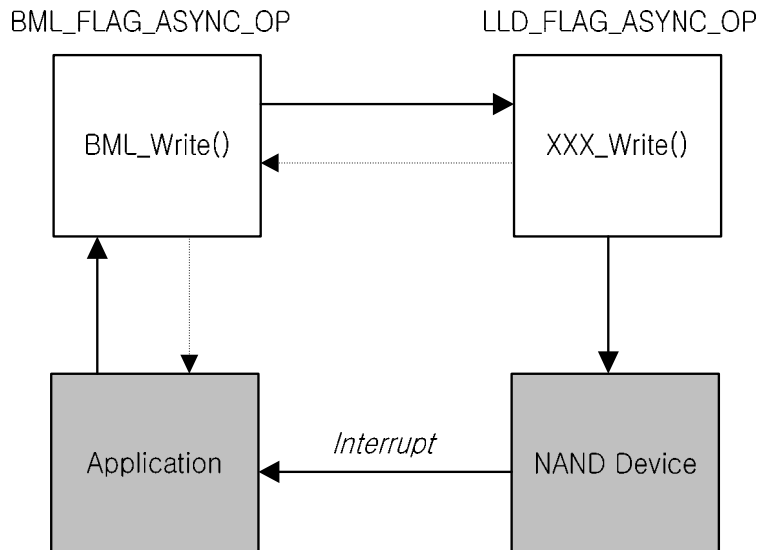


Figure 7-10. BML and Interrupt

7.2.4. STL and Interrupt

To support asynchronous mode, STL enable to select asynchronous/synchronous mode using `bAsyncMode` flag in `STL_Open()`. STL also supports `STL_AWrite()` and `STL_ADelete()` functions for asynchronous mode.

The mechanism of asynchronous mode of STL is different with the mechanism of asynchronous mode of BML. BML functions can be mapped to LLD functions directly in no error case. If `BML_FLAG_ASYNC_OP` flag is selected to use asynchronous driver model at BML, the interrupt will be invoked after finishing the given command. However, STL functions cannot be mapped to LLD functions directly. STL functions are able to be mapped to several BML functions. Because several NAND commands per one STL function are issued, the mechanism of asynchronous mode of BML can not be adapted to STL.

Asynchronous feature of STL is different concept with asynchronous API calling of BML.

7.2.4.1. Asynchronous Feature of STL

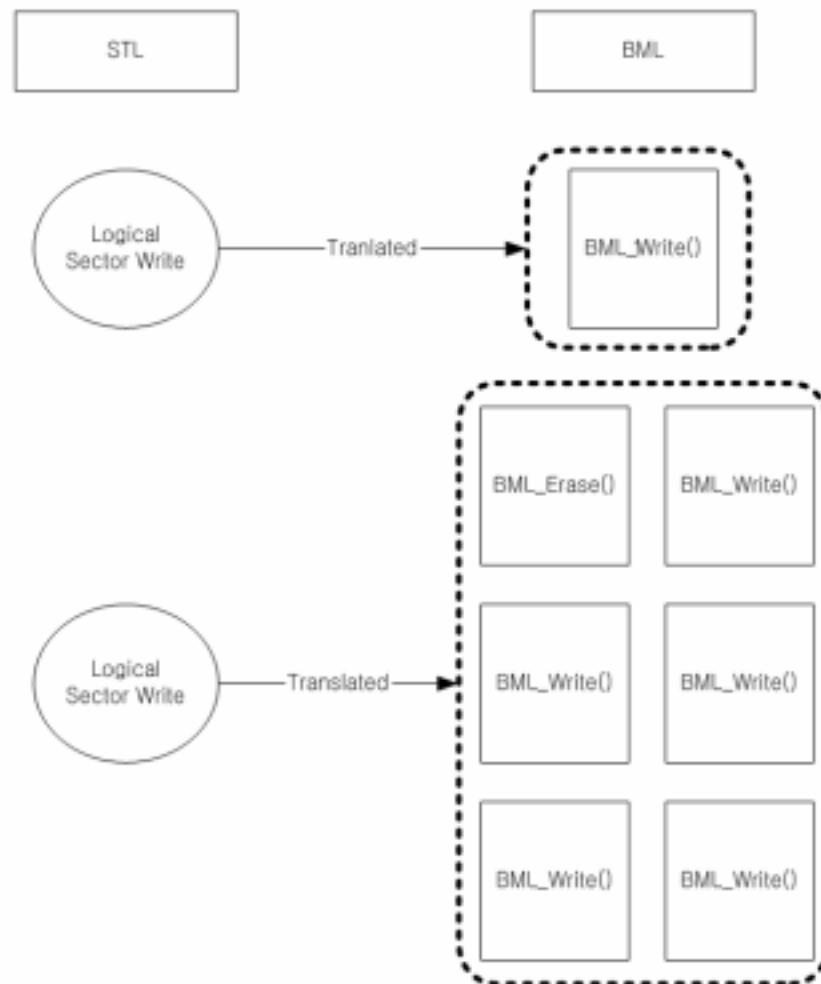


Figure 7-11. STL to BML Mapping Example

At first, `STL_AWrite()` puts BML operations which are needed to execute `STL_AWrite()` operation into internal operation queue. However, It does not mean that BML functions are already called. Actually, a BML function can be called after `STL_Sync()` is called first.

Once STL_Sync() is called, a BML function was dequeued from head of the operation queue, and then can be handled. To minimize the response time of STL, STL_Sync() calls only one BML operation at a time. To call BML function in asynchronous mode, STL uses BML_FLAG_ASYNC_OP flag as a parameter of BML functions.

In case of using BML_FLAG_ASYNC_OP flag, the interrupt is occurred after the operation of the corresponding command is completed. Then, interrupt service routine can call STL_Sync() again.

Once STL_Sync() is called again, STL gets the next BML function and handle it. STL repeats BML functions one by one until the queue becomes empty. In asynchronous mode of STL, STL manages one BML function at once to prevent delays in the response time of the whole system



Figure 7-12. Operation Queue Created by STL_AWrite()

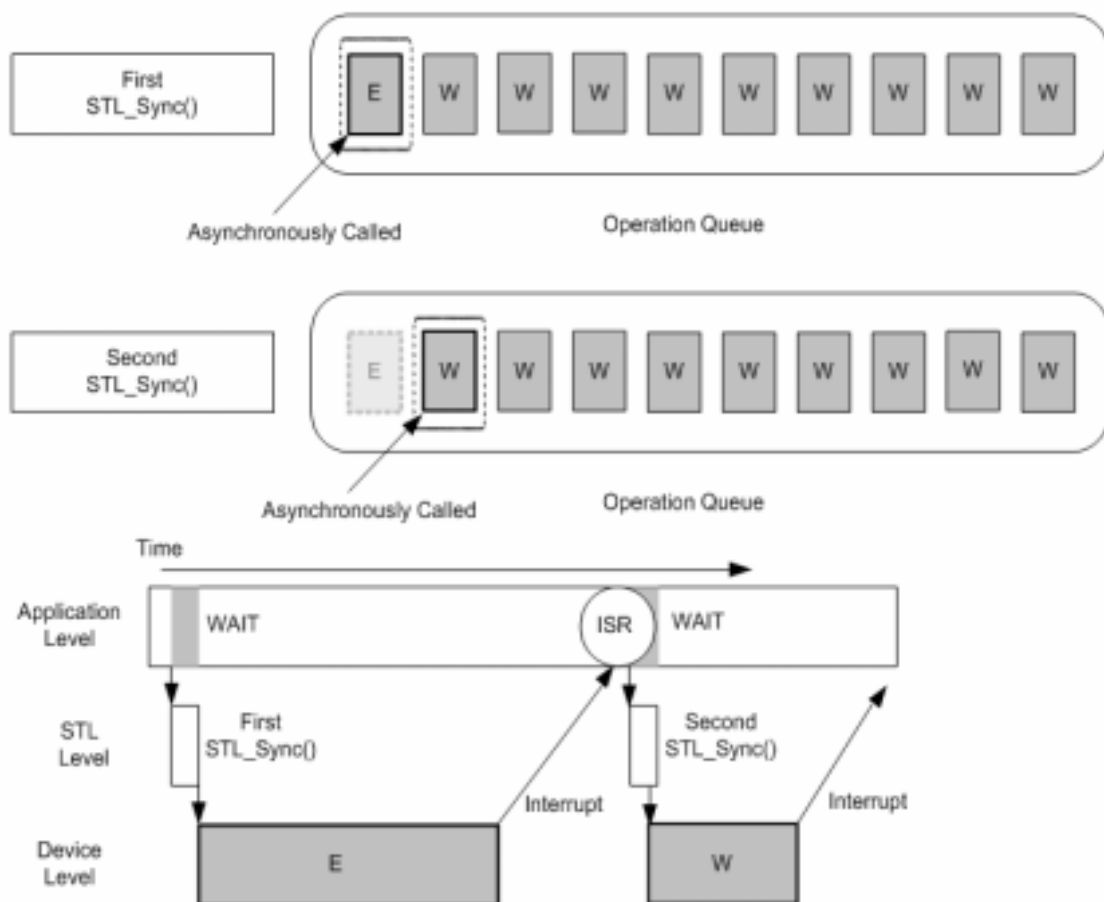


Figure 7-13. `STL_Sync()` Operation (nQuantum: 1)

7.2.5. Timer

7.2.5.1. Asynchronous Feature of XSR and Timer

As mentioned in previous chapter, `STL_AWrite()/STL_ADelete()` generates an operation queue which has BML job schedules. After creating an operation queue, `STL_Sync()` actually calls one BML operation at a time. Because the major purpose of this asynchronous mechanism is to reduce response time of STL, processing time is a key factor to `STL_Sync()`.

`STL_Sync()` receives `nQuantum` which is related to STL response time.

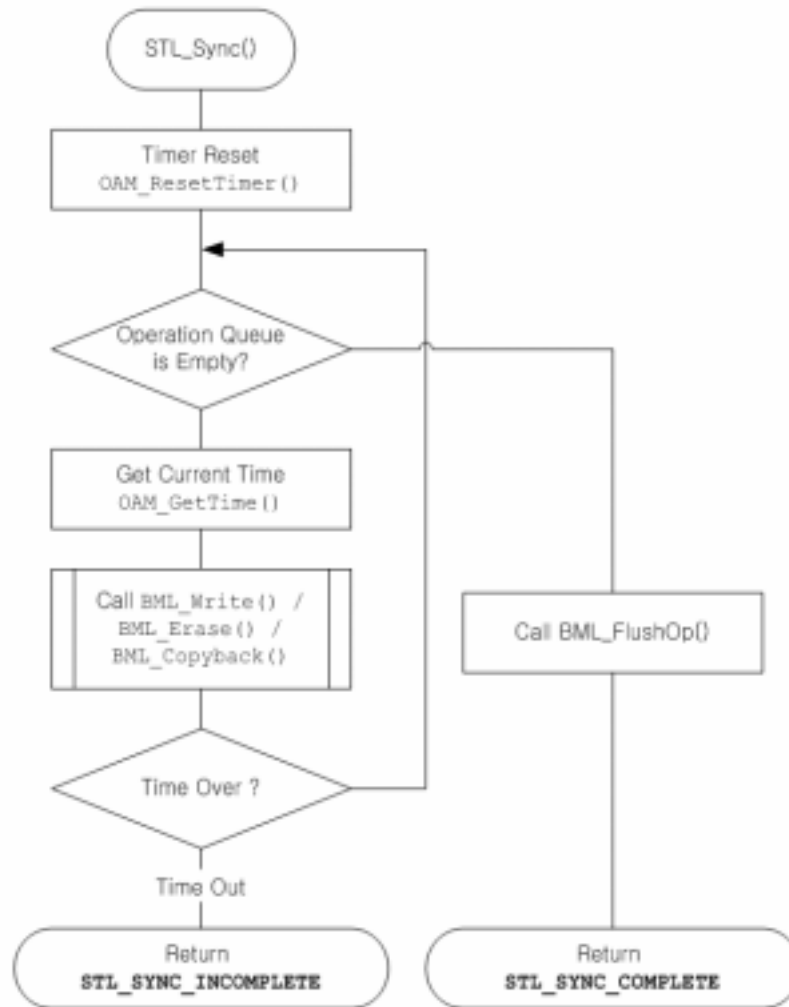


Figure 7-14. Flowchart of STL_Sync()

In current XSR implementation, nQuantum and timer functions do not mean real timer values. The value of nQuantum means a number of BML operations to be executed at once. Timer function returns the value of a global counter variable instead of the timer tick of the operating system. The value of counter is related with the number of BML operations.

Actually, OAM_GetTime() returns a current value of a global counter in unsigned int type. To understand OAM_GetTime() and OAM_ResetTimer(), you can imagine the ordinary counter which increases continuously. OAM_ResetTimer() resets the global counter variable as 0, and OAM_GetTime() returns the current value of the counter and increases the counter. The return value of the timer function is used for comparing with the quantum value of STL_Sync().

However, if user wants to implement the asynchronous feature based on execution time of operation, timer functions should be changed to use the real OS timer instead of the counter.

7.2.5.2. Implementation

XSR OAM has 2 functions for asynchronous feature as follows.

```
VOID    OAM_ResetTimer(VOID)
UINT32  OAM_GetTime(VOID)
```

OAM_ResetTimer() initializes a timer.

STL_Sync() calls OAM_ResetTimer() before calling OAM_GetTime(). Therefore, OAM_ResetTimer() is good function to initialize a timer. If you do not want to use STL_AWrite(), STL_ADelete(), and STL_Sync(), you do not need to implement this function.

OAM_ResetTimer() implementation example for Symbian OS

```
static UINT32 nTimerCounter = 0;
static UINT16 nTimerPrevCnt = 0;

VOID
OAM_ResetTimer(VOID)
{
    nTimerPrevCnt = 0;
    nTimerCounter = 0;
}
```

OAM_GetTime() returns current timer counter value.

Because STL_Sync() uses difference of OAM_GetTime() calls for every BML function, return timer value's unit is not important. Just remember this value is compared with STL_Sync() argument nQuantum.

If you do not want to use STL_AWrite(), STL_ADelete(), and STL_Sync(), you can implement this function to return constant value.

OAM_GetTime() implementation example for Symbian OS

```
UINT32
OAM_GetTime(VOID)
{
    return nTimerCounter++;
}
```

7.2.6. Implementation

7.2.6.1. PAM Implementation

PAM has 5 interrupt functions as follows.

```
VOID    PAM_InitInt(UINT32 nLogIntId);
VOID    PAM_BindInt(UINT32 nLogIntId);
VOID    PAM_EnableInt(UINT32 nLogIntId);
VOID    PAM_DisableInt(UINT32 nLogIntId);
VOID    PAM_ClearInt(UINT32 nLogIntId);
```

These functions are not used inside XSR layer; STL, and BML. These functions may be used in LLD or STL's user layer.

In current version of XSR, PAM interrupt functions control the interrupts through OAM interrupt functions.

Interrupt handling flow is divided into two: platform dependent part (PAM) and OS dependent part (OAM). PAM interrupt functions translate a logical interrupt ID into a physical interrupt ID. And PAM interrupt functions call OAM interrupt functions to handle the interrupt which is specified by logical interrupt ID and physical interrupt ID.

7.2.6.2. Implementation in upper layer of STL

To use asynchronous feature of XSR, upper layer of STL should have ISR and additional function call handler. The ISR receives interrupts from NAND device and notify it to the function call handler. Activated function call handler calls `STL_Sync` to process asynchronous operations.

If current mode is asynchronous, the upper layer of STL should call `PAM_InitInt` and `PAM_BindInt` before calling `STL_Init` and `STL_Open`. `PAM_InitInt` initialize interrupts for the NAND device. `PAM_BindInt` binds interrupts for the NAND device. `PAM_BindInt` maps logical interrupts for the NAND device to specific physical interrupts.

In asynchronous mode, `STL_AWrite` should be called instead of `STL_Write`. Due to limitation of size of the operation queue, maximum size of a write request is restricted in asynchronous mode. The size of a write request from upper layer should be smaller than or equal to 64 KB.

After calling `STL_AWrite`, `STL_Sync` should be called to handle BML operations in the operation queue. Value of `nQuantum`, a parameter of `STL_Sync`, should be 1 in current version of XSR.

After dequeuing BML operation from the operation queue, LLD enables the interrupt and process the operation. `STL_Sync` returns `STL_SYNC_INCOMPLETE` if operations remain in the queue. When the NAND device finishes the operation, it generates interrupt to CPU. Then ISR lets the function call handler call next `STL_Sync`.

If operation queue is empty, `STL_Sync` returns `STL_SYNC_COMPLETE` to the function call handler and upper layer executes next STL operation.

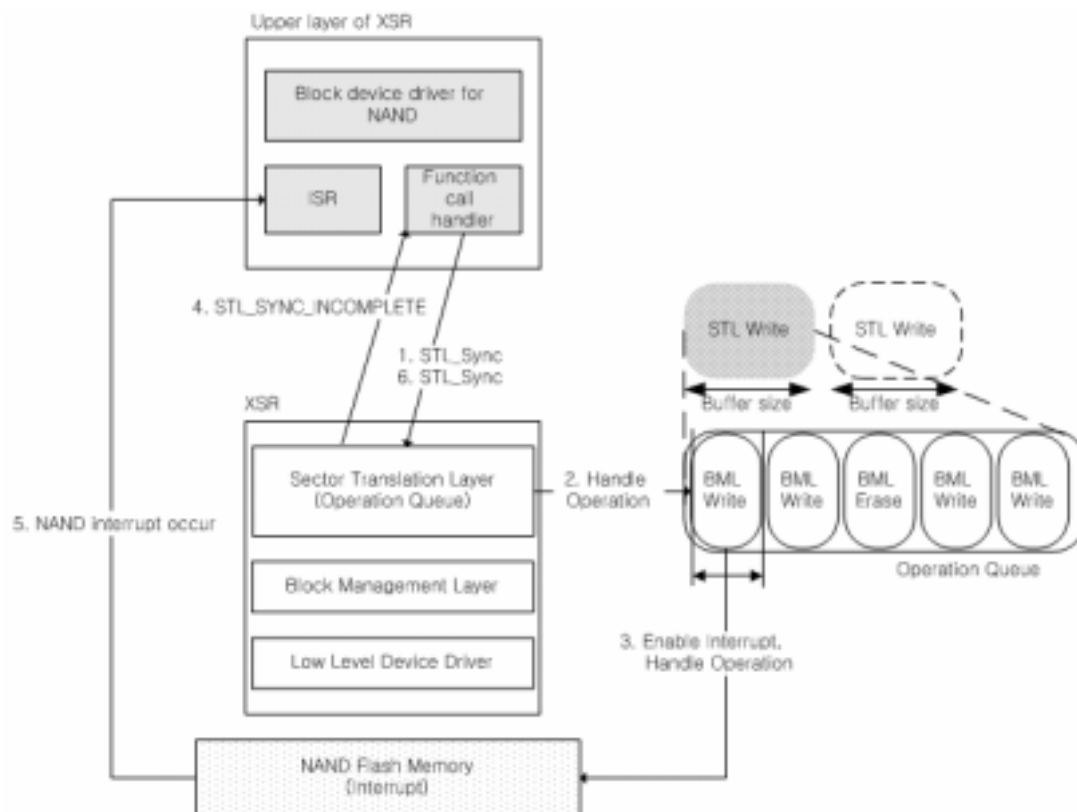


Figure 7-15. Flow between upper layer and XSR in asynchronous mode

Example for upper layer of STL to support interrupts

```

VOID
Example(VOID)
{
    if (asynchorouns mode)
    {
        Calls PAM_InitInt()
        Calls PAM_BindInt()
    }

    Calls STL_Init()

    Calls STL_Open()
}

```

7.2.6.3. Implementation in LLD

To use the asynchronous feature of XSR, LLD should call `PAM_ClearInt()` and `PAM_EnableInt()` at `XXX_Write`, `XXX_Erase`, `XXX_MEraser` and `XXX_CopyBack` functions. These functions can receive `LLD_FLAG_ASYNC_OP` flag. When the flag is set, LLD should call `PAM_ClearInt()` and `PAM_EnableInt()` before issuing a command to NAND device.

`PAM_ClearInt()` clears NAND interrupt. When interrupt service routine is called by

hardware interrupt, after proper processing of interrupt PAM_ClearInt() should be called to avoid endless invoking of the interrupt.

PAM_EnableInt() enables NAND interrupt for next NAND command. Interrupt which is handled in LLD should support both interrupt mode command and non-interrupt mode command. Because of this, LLD should enable and disable the interrupt.

Example for LLD Erase to support interrupts

```
INT32
XXX_Erase( ..., UINT32 nFlag)
{
    Wait processing command
    Checks previous error and return error code

    if (nFlag & LLD_FLAG_ASYNC_OP)
    {
        Calls PAM_ClearInt()
        Calls PAM_EnableInt()
    }

    Issues erase command to NAND device
    (do not wait)
}
```

7.3. Power-Off Recovery

XSR OAM has a timer function to handle power-off error in BML layer. This section describes background of power-off error processing and implementation method of the timer function.

7.3.1. Power-Off Error Processing and Timer

BML handles all NAND device errors such as page read error, page write error, block erase error, , etc. In case of processing block erase error, there is a problem relate with power-off situation.

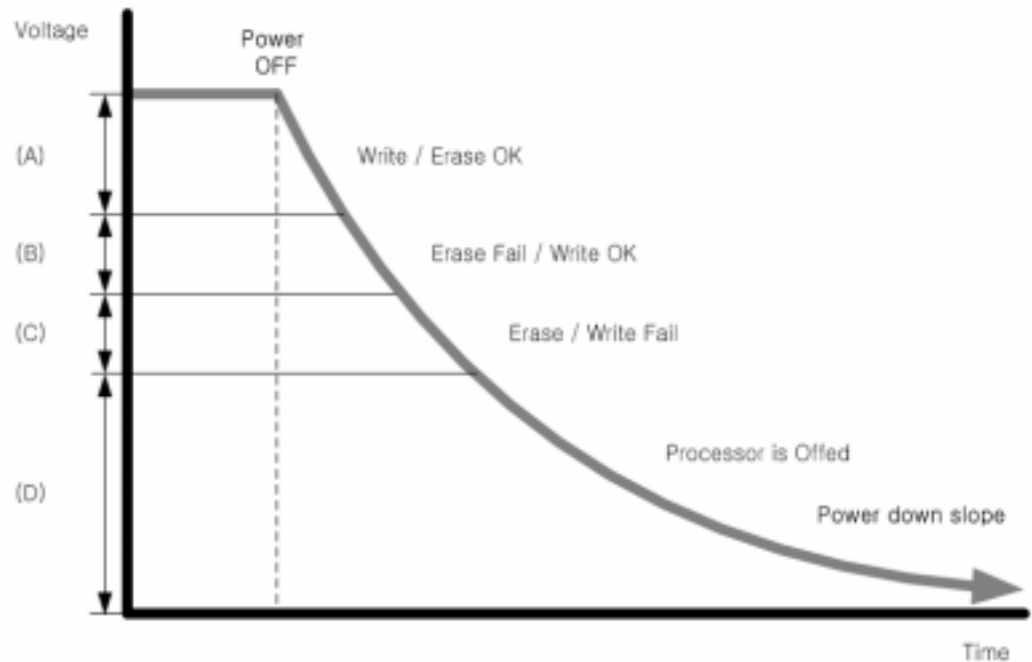


Figure 7-16. Power-off Slope Assumption

Generally NAND block erase operation consumes more electronic current than NAND page write operation. Because of this, power-off slope may cause BML to mistake for block erase error handling. Figure 7-16 shows an example of power-off slope. When power is offed, the voltage goes down to zero with slope. Because of low voltage, NAND block erase and NAND page write operation may fail. If processor's operating voltage threshold is lower than NAND's threshold, and NAND erase operation is issued at (B) area in Figure 7-16, BML will try to replace the block. However the block is not real run-time badblock. In (B) area, block erase operation is failed but page write function is still working. Therefore normal block may be recognized as run-time bad block by BML.

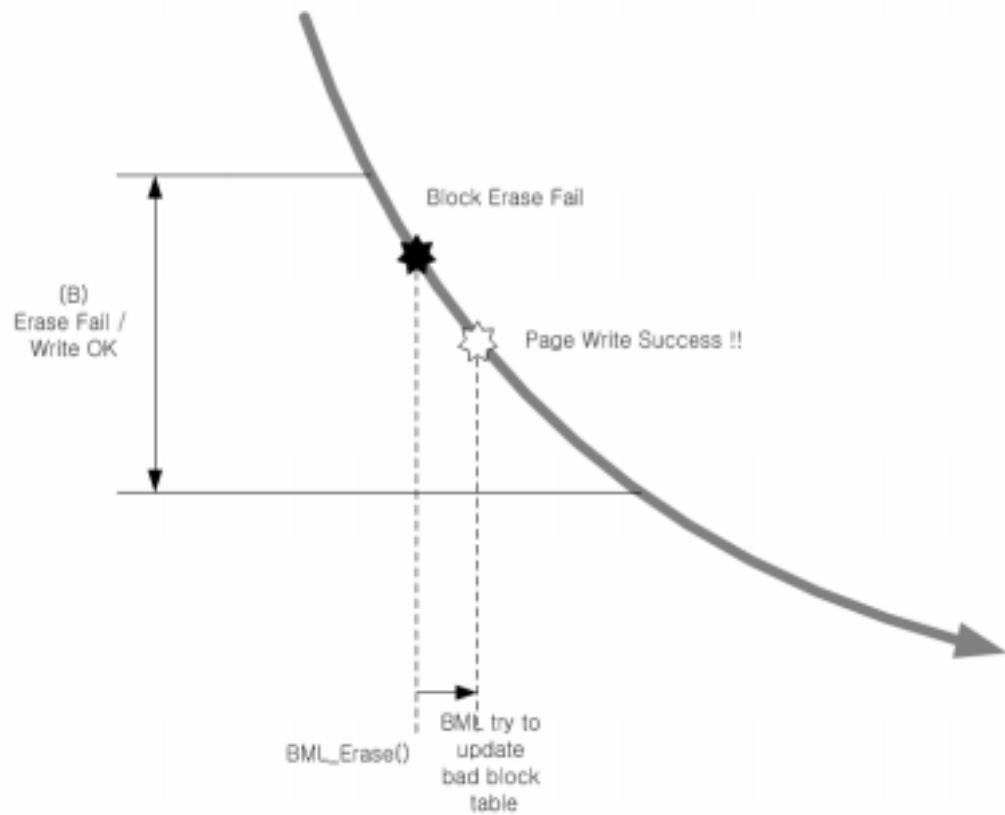


Figure 7-17. NAND Block Erase Error at Power-off

To avoid this wrong run-time bad block processing, BML uses `OAM_WaitNMSec()`. `OAM_WaitNMSec()` is delay function and receives an argument about delay time. The argument is given as millisecond unit.

When BML receives block erase error from LLD, BML calls `OAM_WaitNMSec()` before starting error handling. This delay time can be modified by `BML_IOCTL1()`.

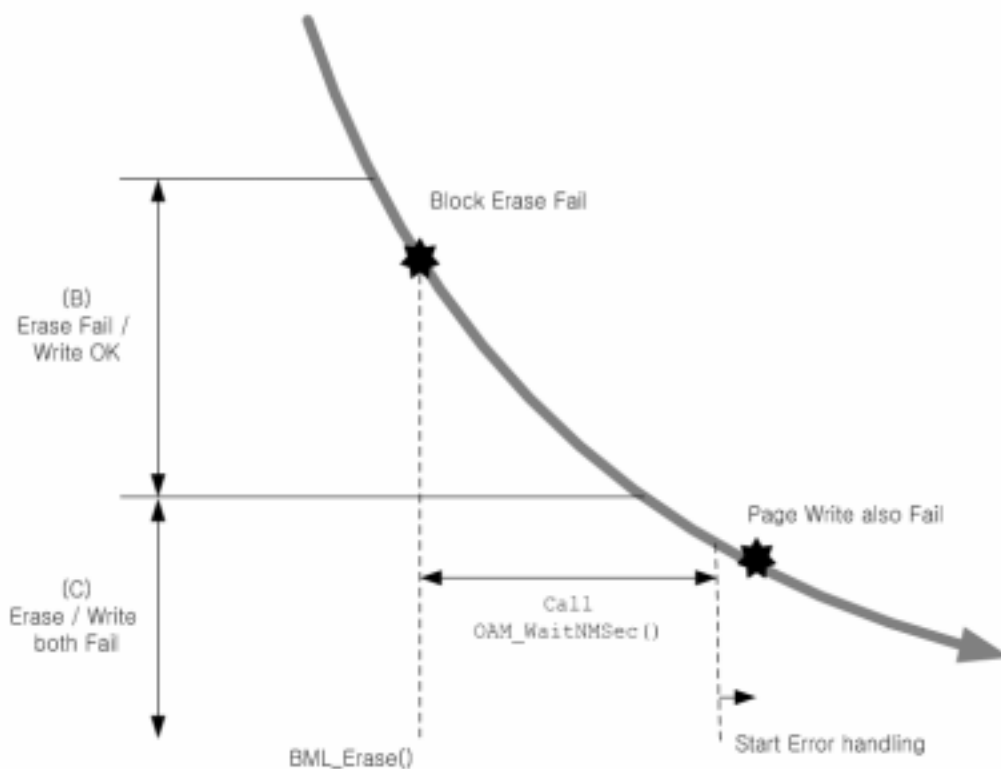


Figure 7-18. OAM_WaitNMSec() Usage

7.3.2. Implementation

XSR OAM has 1 function for power-off recovery as follows.

```
VOID OAM_WaitNMSec(UINT32 nNMSec);
```

OAM_WaitNMSec() delays during given milliseconds.

If you do not care about power-off case, you do not need to implement OAM_WaitNMSec().

OAM_GetTime() implementation example for Win32

```
VOID
OAM_WaitNMSec(UINT32 nNMSec)
{
    Sleep(nNMSec);
}
```

7.4. Deferred Check Operation

Deferred Check Operation is an algorithm to change the operation sequence of the general operation in software. It efficiently uses time for Physical Operation of the device; that improves the system performance. A user can adapt Deferred Check Operation in write operation and erase operation function.

The sequence of the general operation of the device is divided into three;

- Pre-Operation before Command Issue
(Register Setting + Data Transfer)
- Physical Operation the real operation is executed
- Normal operation Check

The following figure compares the general operation with Deferred Check Operation.

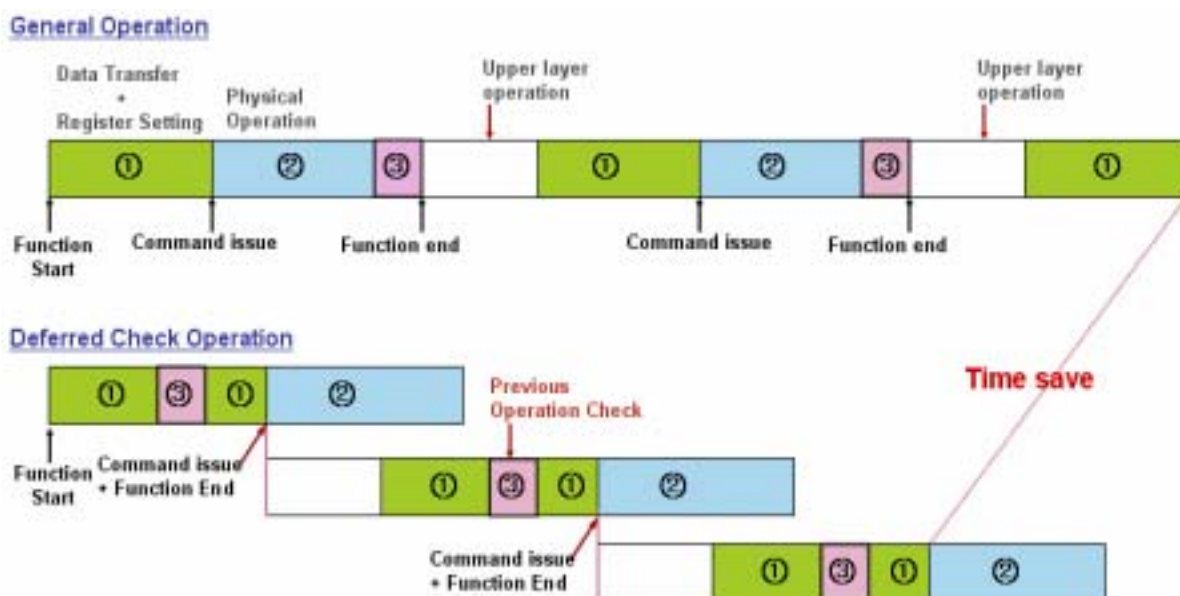


Figure 7-19. Compare General Operation with Deferred Check Operation

The sequence that the general operation is executed is as follows;

- Register related to the operation is set.
- Data of the host memory is transferred to the buffer of the device
(in write operation),
- Physical Operation is executed in the buffer of the device to NAND array cell
(in write/erase operation),
- Finally, check the operation is normally executed.

Here, step is the time for Physical Operation. CPU suspends as idle during Physical operation.

Deferred Check Operation is designed to efficiently use CPU idle time. Using Deferred Check Operation, the corresponding function is closed after issuing the command. LLD can



execute other operation during time for Physical Operation. Thus, the total system performance is improved.

As the function of Deferred Check Operation closes its operation without handling after step , Operation Check (step) is **deferred** to the middle of the sequence of next Operation (step). A user implement that step (checking the former operation is normally executed) in the middle of all Operation functions.

After issuing the command, LLD handles the corresponding function. CPU does not wait as idle during Physical Operation time about the device. Else, CPU executes the next operation. Thus, the overall system performance is improved using Deferred Check Operation.

The sequence that Deferred Check Operation is executed is (The sequence depends on the device) as follows;

1. Data Transfer or Register Setting ()
2. Host checks the former operation is normally executed ()
3. If no error, LLD issues the command (Command Issue) ()
4. LLD finishes the corresponding function of the new command
5. While the device executes Physical Operation, CPU can execute the next operation.
()

If a user wants to use Deferred Check Operation, LLD has the process to handle the previous operation error. When the previous write operation error occurs in LLD, BML has to handle this error. In this case, BML requests data used in the previous write operation. XXX_GetPrevOpData is used for data request of BML.

If a user does not use LLD, a user does not need to implement XXX_GetPrevOpData in LLD functions. (However, XXX_GetPrevOpData itself must be existed because of LFT policy of BML.)

Deferred Check Operation functions are as follow;

- XXX_GetPrevOpData
- XXX_FlushOp

7.5. Byte Alignment Restrictions

Most of CPUs require that objects and variables reside at particular location in the system memory. For example, 32 bit microprocessors typically organize the memory as shown below. The memory is accessed by performing 32 bit bus cycles. 32 bit bus cycles can be performed at the addresses that are divisible by 4. This requirement is called "byte alignment". Thus, a 4byte integer can be located at 0x1000 or 0x1004, but cannot be located at 0x1001.

To handle the misaligned interger for the read/write operation costs more than to handle the aligned interger for the read/write operation.

For example, an aligned 4byte integer X would be written as X0, X1, X2 and X3. Thus the microprocessor can read the complete 4byte integer in a single bus cycle. However, if the same microprocessor attempts to access 4byte integer at address 0x000D, it will have to read bytes Y0, Y1, Y2 and Y3. This read cannot be performed by a single 32 bit bus cycle. The microprocessor has to perform twice different read operations at address 0x100C and 0x1010 to read the complete 4byte integer Thus it takes twice time to read the misaligned 4byte integer than to read the aligned 4byte integer. .

Table 7-2. 4Byte Alignment Example

	Byte 0	Byte 1	Byte 2	Byte 3
0x1000				
0x1004	X0	X1	X2	X3
0x1008				
0x100C		Y0	Y1	Y2
0x1010				

Thus, users must handle the "byte alignment" problem when writing LLD codes.

For example, let suppose to use NAND device that accesses bus Bandwidth. If the buffer address is an odd number in read/write operation, the "byte alignment" problem occurs. To solve this byte alignment problem, LDD fixes 2byte alignment using the aligned buffer.

Also, if a device access data using assembler, a user has to set as 4Byte alignment not to broke the buffer.

The following is an example code of 4Byte alignment in reading data using assembler.

```
INT32  PNS_Read(UINT32 nDev,  UINT32 nPsn,  UINT32 nNumOfScts,
                UINT8  *pMBuf,  UINT8  *pSBuf,  UINT32 nFlag)
{
    UINT32      nBAddr;
    PNSSpec     *pstDevSpec;
    UINT8       n4Cycles;
    UINT8       *pTmpMBuf;
    UINT8       *pTmpSBuf;

    PNS_DBG_PRINT((TEXT(" [PNS : IN] ++PNS_Read() nPsn
                        = %d\r\n"), nPsn));

    pstDevSpec = astPNSDev[nDev].pstDevSpec;
    nBAddr     = DEV_BASE(nDev);
    n4Cycles   = pstDevSpec->n4CycleDev;
```

```

pTmpMBuf      = pMBuf;
pTmpSBuf      = pSBuf;

/* if pMBuf is not aligned by 4 bytes, */
if ((UINT32) pMBuf & 0x03)
{
    pTmpMBuf = (UINT8 *) astBufBA[nDev].pBAMBuf;
}

/* if pSBuf is not aligned by 4 bytes, */
if ((UINT32) pSBuf & 0x03)
{
    pTmpSBuf = (UINT8 *) astBufBA[nDev].pBASBuf;
}

if (pTmpMBuf != NULL)
{
    NF_CMD (nBAddr, CMD_READ);
    SET_ADDR(nBAddr, nPsn, n4Cycles);
    NF_WAITRB(nBAddr);
    NF_CMD (nBAddr, CMD_READ);
    /* It is Implemented to Assembler Code */
    pstDevSpec->Read512(pTmpMBuf, rNFDATAPtr(nBAddr));

    if (pTmpSBuf != NULL)
    {
        pstDevSpec->Read16(pTmpSBuf, rNFDATAPtr(nBAddr));
    }
}
else
{
    NF_CMD (nBAddr, CMD_READ_SPARE);
    SET_ADDR(nBAddr, nPsn, n4Cycles);
    NF_WAITRB(nBAddr);
    NF_CMD (nBAddr, CMD_READ_SPARE);

    pstDevSpec->Read16(pTmpSBuf, rNFDATAPtr(nBAddr));
}

/* if pMBuf is not aligned by 4 bytes */
if ((UINT32) pMBuf & 0x03)
{
    PAM_Memcpy(pMBuf, pTmpMBuf, PNS_MAIN_SIZE);
}

/* if pSBuf is not aligned by 4 bytes */
if ((UINT32) pSBuf & 0x03)
{
    PAM_Memcpy(pSBuf, pTmpSBuf, PNS_SPARE_SIZE);
}

PNS_DBG_PRINT((TEXT("[PNS : OUT] --PNS_Read()\r\n")));

return (PNS_SUCCESS);
}

static VOID
_Read512x16(UINT8 *pBuf, UINT32 nAddr)
{
    asm("mov    r0, %0" : : "g" ((UINT32) pBuf));
    asm("mov    r1, %0" : : "g" ((UINT32) nAddr));
}

```

```

asm("stmfd sp!,{r1 - r11}");

asm("mov    r2, #0x200");
asm("asmr512x16:");

asm("ldrh   r4, [r1]");
asm("ldrh   r3, [r1]");
asm("orr    r4, r4, r3, lsl #16");

asm("ldrh   r5, [r1]");
asm("ldrh   r3, [r1]");
asm("orr    r5, r5, r3, lsl #16");

asm("ldrh   r6, [r1]");
asm("ldrh   r3, [r1]");
asm("orr    r6, r6, r3, lsl #16");

asm("ldrh   r7, [r1]");
asm("ldrh   r3, [r1]");
asm("orr    r7, r7, r3, lsl #16");

asm("ldrh   r8, [r1]");
asm("ldrh   r3, [r1]");
asm("orr    r8, r8, r3, lsl #16");

asm("ldrh   r9, [r1]");
asm("ldrh   r3, [r1]");
asm("orr    r9, r9, r3, lsl #16");

asm("ldrh   r10, [r1]");
asm("ldrh   r3, [r1]");
asm("orr    r10, r10, r3, lsl #16");

asm("ldrh   r11, [r1]");
asm("ldrh   r3, [r1]");
asm("orr    r11, r11, r3, lsl #16");

asm("stmia  r0!, {r4 - r11}");
asm("subs   r2, r2, #32");
asm("bne    asmr512x16");

asm("ldmfd  sp!, {r1 - r11}");
}

static VOID
_Read16x16(UINT8 *pBuf, UINT32 nAddr)
{
    asm("mov    r0, %0" : : "g" ((UINT32) pBuf));
    asm("mov    r1, %0" : : "g" ((UINT32) nAddr));

    asm("stmfd  sp!,{r1 - r7}");

    asm("ldrh   r4, [r1]");
    asm("ldrh   r3, [r1]");
    asm("orr    r4, r4, r3, lsl #16");

    asm("ldrh   r5, [r1]");
    asm("ldrh   r3, [r1]");
    asm("orr    r5, r5, r3, lsl #16");

    asm("ldrh   r6, [r1]");

```



```
asm("ldrh  r3, [r1]");
asm("orr   r6, r6, r3, lsl #16");

asm("ldrh  r7, [r1]");
asm("ldrh  r3, [r1]");
asm("orr   r7, r7, r3, lsl #16");

asm("stmia r0!, {r4 - r7}");

asm("ldmfd sp!, {r1 - r7}");
}
```

7.6. ECC Policy

XSR is designed to support both software ECC and hardware ECC. Software ECC codes are included in BML Layer and Hardware ECC should be implemented in LLD Layer.

Therefore, user should set nEccPol corresponding to ECC policy that users use as follows;

1. When the value of nEccPol should set as SW_ECC

If user wants to use software ECC which is supported by XSR, sets the value of nEccPol flag as SW_ECC. In that case, BML handles ECC generation and correction by Hamming code.

2. When the value of nEccPol should set as HW_ECC

If user wants to use hardware ECC which is supported by hardware, sets the value of nEccPol as HW_ECC. In that case, hardware ECC should satisfy following conditions.

First, ECC algorithm supported by hardware should be identical with ECC algorithm used by Samsung⁷. Second, spare assignment for generated ECC code by hardware should be identical with NAND Flash Spare Area Assignment Standard of Samsung⁸. Third, stored ECC pattern should be compatible with SAMSUNG standard. If hardware ECC to be used can not satisfy these conditions, user sets the value of nEccPol as NO_ECC instead of HW_ECC.

3. When the value of nEccPol should set as NO_ECC

When user does not use any ECC algorithm, the value of nEccPol sets as NO_ECC. If hardware ECC algorithm is different from ECC algorithm used by Samsung, stored ECC pattern is not compatible with SAMSUNG standard or spare assignment for ECC code is identical with Samsung standard, the value of nEccPol should also be set as NO_ECC. If spare assignment for ECC code is different from Samsung standard although hardware ECC algorithm and ECC pattern are identical with ECC algorithm used by Samsung, XSR can not be used in this case. To use XSR, users must obey spare assignment standard of Samsung.

⁷ Memory Division, Samsung Electronics Co., Ltd, "ECC(Error Checking & Correction) Algorithm", http://www.samsung.com/Products/Semiconductor/Flash/TechnicalInfo/eccalgo_040624.pdf

⁸ Memory Division, Samsung Electronics Co., Ltd, "NAND Flash Spare Area Assignment Standard", http://www.samsung.com/Products/Semiconductor/Flash/TechnicalInfo/Spare_assignment_recommendation.pdf

General usages of Hardware ECC are as follows;

1. Hardware ECC by NAND Device (for example, OneNAND)
2. Hardware ECC by CPU (for example, NAND controller)

The implementation and attention of the above cases are as follows.

□ Implementation

Method of read/write operation for main area and spare area in LLD are explained in following table.

Table 7-3. Usage of Main Buffer and Spare Buffer in Read/Write Operation

	Main Buffer	Spare Buffer
Page Read/Write	Should be allocated	Should be allocated
Main Read/Write	Should be allocated	NULL
Spare Read/Write	NULL	Should be allocated

When performing page read/write operation with ECC ON in the system where Hardware ECC is integrated or CPU handles ECC, work flow is like below

In Read operation:

1. Read main and spare area
2. Generate ECC Code
3. Compare generated ECC with Original ECC which read from spare area.
4. If 1bit ECC error occurs then return a LLD_READ_CERROR_XX.
or 2bit ECC error occurs then return a LLD_READ_UERROR_XX.

In Write operation:

1. Generate ECC Code
2. Write main and spare area

In case of OneNAND device, it can get generated ECC values by using register of device. If CPU supports Hardware ECC, codes should be implemented to get ECC value generated by CPU, corresponding to the above sequence.

□ Attention that spare buffer is NULL in read/wrie main area

When reading and writing only main area as ECC ON, reading and writing ECC value causes problems because spare buffer is null. In order to solve this problem, follow the below sequences

In Read operation,

1. Allocate spare buffer in LLD layer
2. Read main/spare area
3. Generate ECC value for main area
4. Compare genetated ECC with ECC for main area in spare area
5. If 1bit ECC error occurs then return a LLD_READ_CERROR_XX.
or 2bit ECC error occurs then return a LLD_READ_UERROR_XX.

In Write operation,

1. Allocate spare buffer and fill as 0xFF in LLD layer
2. Fill main buffer with data
3. Generate ECC value for main area
4. Fill generated ECC in ECC position of allocated spare buffer. For more information about ECC position, refer to "Spare assignment Standard".⁹
5. Write main and spare area

The following tables are return values when ECC error occur.

Table 7-4. Return values of Main area ECC error

	Correctable error	Uncorrectable error
1 st Sector of a page	LLD_READ_CERROR_M0	LLD_READ_UERROR_M0
2 nd Sector of a page	LLD_READ_CERROR_M1	LLD_READ_UERROR_M1
3 rd Sector of a page	LLD_READ_CERROR_M2	LLD_READ_UERROR_M2
4 th Sector of a page	LLD_READ_CERROR_M3	LLD_READ_UERROR_M3

Table 7-5. Return values of Spare area ECC error

	Correctable error	Uncorrectable error
1 st Sector of a page	LLD_READ_CERROR_S0	LLD_READ_UERROR_S0
2 nd Sector of a page	LLD_READ_CERROR_S1	LLD_READ_UERROR_S1
3 rd Sector of a page	LLD_READ_CERROR_S2	LLD_READ_UERROR_S2
4 th Sector of a page	LLD_READ_CERROR_S3	LLD_READ_UERROR_S3

Return value is divided into the major value and minor value.

Major value classifies errors of an operation.

Minor value represents details of errors.

For example, If user read a page of large block NAND. When uncorrectable ECC error occurs at first and third sectors, LLD_READ_UERROR_XX has minor return values. It can be described as the following formula after combining with LLD_READ_ERROR.

$nRet = LLD_READ_ERROR \mid LLD_READ_UERROR_S0 \mid LLD_READ_UERROR_S2;$

The following is an example of LLD Read/Write Function using hardware ECC of CPU(NAND Controller). This example adapts small block NAND device.

```

INT32
PNS_Read(UINT32 nDev, UINT32 nPsn, UINT32 nNumOfScts,
          UINT8 *pMBuf, UINT8 *pSBuf, UINT32 nFlag)
{

```

⁹ Memory Division, Samsung Electronics Co., Ltd, "NAND Flash Spare Assignment recommendation", http://www.samsung.com/Products/Semiconductor/Flash/TechnicalInfo/Spare_assignment_recommendation.pdf

```

UINT32  nBAddr;
INT32   nRet = PNS_SUCCESS;
UINT8   aEccVal[5];
UINT8   aEccValNew[5];
UINT8   *pEccM;
UINT8   *pEccS;
UINT8   *pEccMNew;
UINT8   *pEccSNew;
UINT8   *pS;
UINT8   aSBuf[LLD_SPARE_SIZE];
INT32   nERet;
PNSSpec *pstDevSpec;
UINT8   n4Cycles;

PNS_LOG_PRINT((TEXT("[PNS: IN] ++PNS_Read() nPsn = %d\r\n"),
                  nPsn));

pstDevSpec = astPNSDev[nDev].pstDevSpec;
nBAddr      = astPNSDev[nDev].BaseAddr;
n4Cycles    = pstDevSpec->n4CycleDev;

if (nFlag & LLD_FLAG_ECC_ON)
{
    pS = pSBuf;
    if( pSBuf == NULL)
    {
        pS = aSBuf;
    }

    pEccM    = &aEccVal[0];
    pEccS    = &aEccVal[3];
    pEccMNew = &aEccValNew[0];
    pEccSNew = &aEccValNew[3];
}

if (pMBuf != NULL)
{
    NF_CMD  (nBAddr, CMD_READ);

    SET_ADDR(nBAddr, nPsn, n4Cycles);

    NF_WAITRB(nBAddr);

    NF_CMD  (nBAddr, CMD_READ);

    if (nFlag & LLD_FLAG_ECC_ON)
    {
        PAM_ECCRgReset();
    }

    Read512Bx8(pMBuf, rNFDATAPtr(nBAddr));

    if (nFlag & LLD_FLAG_ECC_ON)
    {
        PAM_GenEccM(pEccMNew);
        Read16Bx8(pS, rNFDATAPtr(nBAddr));
    }
    else
    {
        if (pSBuf != NULL)
        {
            Read16Bx8(pSBuf, rNFDATAPtr(nBAddr));
        }
    }
}

```

```

    }
}
else
{
    NF_CMD (nBAddr, CMD_READ_SPARE);

    SET_ADDR(nBAddr, nPsn, n4Cycles);

    NF_WAITRB(nBAddr);
    NF_CMD (nBAddr, CMD_READ_SPARE);

    if (nFlag & LLD_FLAG_ECC_ON)
    {
        Read16Bx8(pS, rNFDATAPtr(nBAddr));
    }
    else
        Read16Bx8(pSBuf, rNFDATAPtr(nBAddr));
}

if (nFlag & LLD_FLAG_ECC_ON)
{
    PAM_Memcpy(aEccVal, (UINT8*)(pS + _GetECCPos(nDev)),
        sizeof(aEccVal));

    if (pMBuf != NULL)
    {
        if ((pEccM[0] != pEccMNew[0]) ||
            (pEccM[1] != pEccMNew[1]) ||
            (pEccM[2] != pEccMNew[2]))
        {
            nERet = PAM_CompEccM(pEccM, pEccMNew,
                (UINT8*)(pMBuf));
            if (nERet == SWECC_C_ERROR)
            {
                PNS_LOG_PRINT((TEXT("[PNS:MSG] 1bit ECC Err
                    at Main of Psn %d\r\n"), nPsn));
                nRet |= (PNS_READ_ERROR |
                    (PNS_READ_CERROR_M0));
            }
            else if(nERet == SWECC_U_ERROR)
            {
                PNS_ERR_PRINT((TEXT("[PNS:ERR] 2bit ECC Err
                    at Main of Psn %d in PNS_Read
                    nDev %d\r\n"), nPsn, nDev));
                nRet |= (PNS_READ_ERROR |
                    (PNS_READ_UERROR_M0));
            }
        }
    }
}

if (pSBuf != NULL)
{
    PAM_GenEccS(pEccSNew, (UINT8*)(pS +
        _GetLsnPos(nDev)));

    if ((pEccS[0] != pEccSNew[0]) ||
        (pEccS[1] != pEccSNew[1]))
    {
        nERet = PAM_CompEccS(pEccS, pEccSNew, (UINT8*)(pS
            + _GetLsnPos(nDev)));
    }
}

```

```

        PNS_LOG_PRINT((TEXT("[PNS:MSG] EccSNEW : 0x%x
                           x\r\n"), *pEccSNew, *pEccS));
        PNS_LOG_PRINT((TEXT("[PNS:MSG] EccSNEW : 0x%x
                           EccS : 0x%x\r\n"), *(pEccSNew+1),
                           *(pEccS+1)));
        if (nERet == SWECC_C_ERROR)
        {
            PNS_LOG_PRINT((TEXT("[PNS:MSG] 1bit ECC Err
                                at Spare of Psn %d\r\n"), nPsn));
            nRet = (PNS_READ_ERROR |
                    (PNS_READ_CERROR_S0));
        }
        else if(nERet == SWECC_U_ERROR)
        {
            PNS_ERR_PRINT((TEXT("[PNS:ERR] 2bit ECC Err
                                at Spare of Psn %d in FIL_Read
                                nDev %d\r\n"), nPsn, nDev));
            nRet = (PNS_READ_ERROR |
                    (PNS_READ_UERROR_S0));
        }
    }
}

PNS_LOG_PRINT((TEXT("[PNS:OUT] --PNS_Read()\r\n")));

return nRet;
}

INT32
PNS_Write(UINT32 nDev, UINT32 nPsn, UINT32 nNumOfScts,
          UINT8 *pMBuf, UINT8 *pSBuf, UINT32 nFlag)
{
    UINT8  aEccVal[5];
    UINT8  *pEccM, *pEccS;
    UINT8  nEccPos;
    UINT8  aSBuf[LLD_SPARE_SIZE];
    UINT8  *pS;
    UINT8  nData;
    UINT32 nBAddr;
    INT32  nRet = PNS_SUCCESS;

    PNS_LOG_PRINT((TEXT("[PNS: IN] ++PNS_Write() nPsn = %d\r\n"),
                      nPsn));

    nBAddr = _DevBase(nDev);

    if (nFlag & LLD_FLAG_ECC_ON)
    {
        PAM_ECCRgReset();
    }
    if ((pMBuf == NULL) && (pSBuf == NULL))
    {
        return(PNS_ILLEGAL_ACCESS);
    }

    if (nFlag & LLD_FLAG_ECC_ON)
    {
        PNS_LOG_PRINT((TEXT("[PNS:MSG] ECC_ON\r\n")));
        pS = pSBuf;
    }

```

```

if (pSBuf == NULL)
{
    OAM_Memset(aSBuf, 0xFF, LLD_SPARE_SIZE);
    pS = aSBuf;
}

OAM_Memset(aEccVal, 0xFF, sizeof(aEccVal));
pEccM    = &aEccVal[0];
pEccS    = &aEccVal[3];
nEccPos  = (UINT8)(_GetECCPos(nDev));

if (pSBuf != NULL)
{
    PAM_GenEccS(pEccS, pS + _GetLsnPos(nDev));
}
}
if (pMBuf != NULL)
{
    NF_CMD(nBAddr, CMD_READ);
    NF_CMD(nBAddr, CMD_WRITE);

    _SetAddr(nDev, nPsn, 0);

    Write512Bx8(pMBuf, rNFDATAPtr(nBAddr));

    if (nFlag & LLD_FLAG_ECC_ON)
    {
        PAM_GenEccM(pEccM);
        PAM_Memcpy((UINT8*)(pS + nEccPos), aEccVal,
                    sizeof(aEccVal));

        Writel6Bx8(pS, rNFDATAPtr(nBAddr));
    }
    else
    {
        if (pSBuf != NULL)
        {
            Writel6Bx8(pSBuf, rNFDATAPtr(nBAddr));
        }
    }
}
else
{
    if (nFlag & LLD_FLAG_ECC_ON)
    {
        PAM_Memcpy((UINT8*)(pS + nEccPos), aEccVal,
                    sizeof(aEccVal));
    }
    NF_CMD(nBAddr, CMD_READ_SPARE);
    NF_CMD(nBAddr, CMD_WRITE);

    _SetAddr(nDev, nPsn, 0);

    if (nFlag & LLD_FLAG_ECC_ON)
    {
        Writel6Bx8(pS, rNFDATAPtr(nBAddr));
    }
    else
    {
        Writel6Bx8(pSBuf, rNFDATAPtr(nBAddr));
    }
}

```

```

    }

    NF_CMD(nBAddr, CMD_WRITE_CONFIRM);

    NF_WAITRB(nBAddr);
    NF_CMD (nBAddr, CMD_READ_STATUS);

    nData = NF_RDDATA(nBAddr);

    if ((nData & 0x01) != (UINT8)0x00)
    {
        PNS_ERR_PRINT((TEXT("[PNS:ERR] Write
                               Err(nPsn:%d)\r\n"), nPsn));
        PNS_LOG_PRINT((TEXT("[PNS:OUT] --PNS_Write()\r\n")));
        nRet = PNS_WRITE_ERROR;
    }

    PNS_LOG_PRINT((TEXT("[PNS:OUT] --PNS_Write()\r\n")));

    return (nRet);
}

```


Appendix

Appendix contains the header file of each module (OAM, LLD and PAM), so a user can refer them in porting.

I. OAM.h

```
#ifndef _XSR_OAM_H_
#define _XSR_OAM_H_

/*****
/* OS Selection
/* DO NOT DEFINE WINCE_OAM or SYMOS_OAM or in This Header File
/* WINCE_OAM : the definition for WindowsCE
/* SYMOS_OAM : the definition for SymbianOS
/*
*****/

#if defined(WINCE_OAM)

#include <windows.h>

#define XSR_RTL_PRINT(x) RETAILMSG(1, x)

#if defined(OAM_DBGMSG_ENABLE)
#define XSR_DBG_PRINT(x) RETAILMSG(1, x)
#endif /* OAM_DBGMSG_ENABLE */

#elif defined(SYMOS_OAM)

#if defined(SYMOS_OAM_EKA2)
#include <locmedia.h>
#include <platform.h>
#endif
#if defined(SYMOS_OAM_EKA1)
#include <k32std.h>
#ifdef REAL_TARGET
#include <m32std.h>
#endif
#endif

#define XSR_RTL_PRINT(x) Kern::Printf x

#if defined(OAM_DBGMSG_ENABLE)

#if defined(SYMOS_OAM_EKA2)
#define XSR_DBG_PRINT(x) Kern::Printf x
#endif

#if defined(SYMOS_OAM_EKA1)
#define XSR_DBG_PRINT(x) __KTRACE_OPT(KBOOT, Kern::Printf x)
#endif

#endif /* OAM_DBGMSG_ENABLE */

#endif /* SYMOS_OAM_EKA2
```

```

#define TEXT(x)                (x)
#endif
#if defined(SYMOS_OAM_EKA1)
#define TEXT(x)                _L(x)
#endif

#else /* other case */

    #if defined(OAM_RTLMMSG_DISABLE)
    #define XSR_RTL_PRINT(x)
    #else
    #define XSR_RTL_PRINT(x)      OAM_Debug x
    #endif /* OAM_RTLMMSG_DISABLE */

    #if defined(OAM_DBGMSG_ENABLE)
    #define XSR_DBG_PRINT(x)      OAM_Debug x
    #else
    #define XSR_DBG_PRINT(x)
    #endif /* OAM_DBGMSG_ENABLE */

    #undef TEXT
    #define TEXT(x)              (VOID *) (x)

#endif

/*****
/* NULL #defines
/* If Each OAM header file defines NULL, following define is not used.*/
*****/
#ifndef NULL
#ifdef __cplusplus
#define NULL 0
#else
#define NULL ((void *)0)
#endif
#endif

#ifdef __cplusplus
extern "C" {
#endif /* __cplusplus */

/*****
/* exported function prototype of OAM
*****/

#ifdef ASYNC_MODE
VOID    OAM_InitInt    (UINT32 nLogIntId);
VOID    OAM_BindInt    (UINT32 nLogIntId, UINT32 nPhyIntId);
VOID    OAM_EnableInt  (UINT32 nLogIntId, UINT32 nPhyIntId);
VOID    OAM_DisableInt (UINT32 nLogIntId, UINT32 nPhyIntId);
VOID    OAM_ClearInt   (UINT32 nLogIntId, UINT32 nPhyIntId);
#endif

VOID    *OAM_Malloc    (UINT32 nSize);
VOID    OAM_Free       (VOID *pMem);
VOID    OAM_Memcpy     (VOID *pDst, VOID *pSrc,  UINT32 nLen);
VOID    OAM_Memset     (VOID *pDst, UINT8  nData,  UINT32 nLen);
INT32   OAM_Memcmp     (VOID *pCmp1, VOID *pCmp2,  UINT32 nLen);

BOOL32  OAM_CreateSM   (SM32 *pHandle);
BOOL32  OAM_DestroySM  (SM32 nHandle);
BOOL32  OAM_AcquireSM  (SM32 nHandle);

```

```

BOOL32  OAM_ReleaseSM (SM32  nHandle);
UINT32  OAM_Pa2Va      (UINT32 nPAddr);

VOID     OAM_Debug      (VOID  *pStr, ...);

VOID     OAM_ResetTimer(VOID);
UINT32   OAM_GetTime    (VOID);
VOID     OAM_WaitNMSec  (UINT32 nNMSec);

BOOL32   OAM_GetROLockFlag (VOID);
VOID     OAM_Idle        (VOID);

#ifdef __cplusplus
}
#endif /* __cplusplus */

#endif /* _XSR_OAM_H_ */

```

II. LLD.h

```

#ifndef _XSR_LLD_H_
#define _XSR_LLD_H_

/*****
/* Debug Configurations */
/*****/
#define LLD_RTL_PRINT(x)          XSR_RTL_PRINT(x)

#if defined(LLD_DEBUG)
#define LLD_DBG_PRINT(x)          XSR_DBG_PRINT(x)
#else
#define LLD_DBG_PRINT(x)
#endif /* LLD_DEBUG */

/*****
/* External Constant Definitions */
/*****/
#define LLD_MAIN_SIZE             512
#define LLD_SPARE_SIZE            16
#define LLD_SECTOR_SIZE           (LLD_MAIN_SIZE)

#define LLD_BW_X08                 0
#define LLD_BW_X16                 1
#define LLD_BW_X32                 2

#define LLD_ME_NO                   0
#define LLD_ME_OK                   1

#define LLD_MEBL_LEN               16

/*****
/* LLD Return Values */
/*****/
/*-----*/
/* Major Return Value */
/*-----*/

```

```

#define LLD_READ_DISTURBANCE      XSR_RETURN_VALUE(0, 0x0007, 0x0000)
/* 1-Bit error by read disturbance */
#define LLD_IOC_NOT_SUPPORT      XSR_RETURN_VALUE(0, 0x0006, 0x0000)
/* Not supported IO Ctrl */
#define LLD_DEV_NOT_OPENED      XSR_RETURN_VALUE(0, 0x0005, 0x0000)
/* Device not opened */
#define LLD_DEV_POWER_OFF      XSR_RETURN_VALUE(0, 0x0004, 0x0000)
/* Power is not supported to LLD */
#define LLD_DEV_NO_LLD_FUNC      XSR_RETURN_VALUE(0, 0x0003, 0x0000)
#define LLD_INIT_BADBLOCK      XSR_RETURN_VALUE(0, 0x0002, 0x0000)
#define LLD_INIT_GOODBLOCK      XSR_RETURN_VALUE(0, 0x0001, 0x0000)
#define LLD_SUCCESS      XSR_RETURN_VALUE(0, 0x0000, 0x0000)
/* success */
#define LLD_ERASE_ERROR      XSR_RETURN_VALUE(1, 0x0001, 0x0000)
/* erase error */
#define LLD_MERASE_ERROR      XSR_RETURN_VALUE(1, 0x0002, 0x0000)
/* merase error */
#define LLD_WRITE_ERROR      XSR_RETURN_VALUE(1, 0x0003, 0x0000)
/* write error */
#define LLD_READ_ERROR      XSR_RETURN_VALUE(1, 0x0005, 0x0000)
/* read error */
#define LLD_CRITICAL_ERROR      XSR_RETURN_VALUE(1, 0x0008, 0x0000)
#define LLD_WR_PROTECT_ERROR      XSR_RETURN_VALUE(1, 0x0009, 0x0000)
#define LLD_ILLEGAL_ACCESS      XSR_RETURN_VALUE(1, 0x000A, 0x0000)
/* Invalid op attempts or values */
#define LLD_ALREADY_INITIALIZED      XSR_RETURN_VALUE(1, 0x000B, 0x0000)

#define LLD_INIT_FAILURE      XSR_RETURN_VALUE(1, 0x0011, 0x0000)
#define LLD_OPEN_FAILURE      XSR_RETURN_VALUE(1, 0x0012, 0x0000)
#define LLD_CLOSE_FAILURE      XSR_RETURN_VALUE(1, 0x0013, 0x0000)

/*-----*/
/* Minor Retun Value */
/*-----*/
/* Detailed Read Error Specifiations */
/* READ_U/CERROR_XX is combined with LLD_READ_ERROR
as Minor Return Value */
#define LLD_READ_CERROR_S0      XSR_RETURN_VALUE(0, 0x0000, 0x0001)
#define LLD_READ_UERROR_S0      XSR_RETURN_VALUE(0, 0x0000, 0x0002)
#define LLD_READ_CERROR_M0      XSR_RETURN_VALUE(0, 0x0000, 0x0004)
#define LLD_READ_UERROR_M0      XSR_RETURN_VALUE(0, 0x0000, 0x0008)

#define LLD_READ_CERROR_S1      XSR_RETURN_VALUE(0, 0x0000, 0x0010)
#define LLD_READ_UERROR_S1      XSR_RETURN_VALUE(0, 0x0000, 0x0020)
#define LLD_READ_CERROR_M1      XSR_RETURN_VALUE(0, 0x0000, 0x0040)
#define LLD_READ_UERROR_M1      XSR_RETURN_VALUE(0, 0x0000, 0x0080)

#define LLD_READ_CERROR_S2      XSR_RETURN_VALUE(0, 0x0000, 0x0100)
#define LLD_READ_UERROR_S2      XSR_RETURN_VALUE(0, 0x0000, 0x0200)
#define LLD_READ_CERROR_M2      XSR_RETURN_VALUE(0, 0x0000, 0x0400)
#define LLD_READ_UERROR_M2      XSR_RETURN_VALUE(0, 0x0000, 0x0800)

#define LLD_READ_CERROR_S3      XSR_RETURN_VALUE(0, 0x0000, 0x1000)
#define LLD_READ_UERROR_S3      XSR_RETURN_VALUE(0, 0x0000, 0x2000)
#define LLD_READ_CERROR_M3      XSR_RETURN_VALUE(0, 0x0000, 0x4000)
#define LLD_READ_UERROR_M3      XSR_RETURN_VALUE(0, 0x0000, 0x8000)

/* Previous Operation Flag */
/* LLD_PREV_OP_RESULT is combined with Major Return Value, and means
Previous Operation Error except LLD_READ_ERROR */
#define LLD_PREV_OP_RESULT      XSR_RETURN_VALUE(0, 0x0000, 0xF0F0)

```

```

/*****
/* Flags
*/

/*****
#define LLD_FLAG_ASYNC_OP      (1 << 0)    /* Write/Erase/Copy */
#define LLD_FLAG_SYNC_OP      (0 << 0)    /* Write/Erase/Copy */
#define LLD_FLAG_SYNC_OP_INV  (~LLD_FLAG_ASYNC_OP)
                                   /* Write/Erase/Copy */

#define LLD_FLAG_ECC_ON        (1 << 1)
#define LLD_FLAG_ECC_OFF      (0 << 1)
#define LLD_FLAG_ECC_OFF_INV  (~LLD_FLAG_ECC_ON)

/*****
/* LLD IO Ctrl Code
*/
/*****
/* IOC Code */
#define IOC_SET_SECURE_LT      0x01    /* Set Lock-Tighten    */
#define IOC_GET_SECURE_STAT    0x02    /* Get Secure Status  */
#define IOC_RESET_NAND_DEV     0x03    /* Reset NAND Device   */
#define IOC_SET_BLOCK_LOCK     0x04    /* Block Lock          */

/* IOC Return Value */
#define LLD_IOC_SECURE_LT      0x01    /* Lock-Tight          */
#define LLD_IOC_SECURE_LS      0x02    /* Locked Status        */
#define LLD_IOC_SECURE_US      0x04    /* Unlocked Status      */

/* Set Device lock-tight */
#define LLD_IOC_SET_SECURE_LT  XSR_IOCTL_CODE(XSR_MODULE_LLD, \
                                             IOC_SET_SECURE_LT, \
                                             XSR_METHOD_INOUT_DIRECT, \
                                             XSR_ANY_ACCESS)
#define LLD_IOC_SET_BLOCK_LOCK XSR_IOCTL_CODE(XSR_MODULE_LLD, \
                                             IOC_SET_BLOCK_LOCK, \
                                             XSR_METHOD_INOUT_DIRECT, \
                                             XSR_ANY_ACCESS)

/* Get Device security status
   return value
   DEV_LT : Device is lock-tighten
   DEV_AL : All blocks of device are locked
   DEV_UL : There is an unlocked block */
#define LLD_IOC_GET_SECURE_STAT XSR_IOCTL_CODE(XSR_MODULE_LLD, \
                                             IOC_GET_SECURE_STAT, \
                                             XSR_METHOD_INOUT_DIRECT, \
                                             XSR_READ_ACCESS)

/* Reset Nand Device */
#define LLD_IOC_RESET_NAND_DEV XSR_IOCTL_CODE(XSR_MODULE_LLD, \
                                             IOC_RESET_NAND_DEV, \
                                             XSR_METHOD_INOUT_DIRECT, \
                                             XSR_ANY_ACCESS)

/*****
/* Data Structures
*/
/*****
typedef struct {          /* offset size */
    UINT8  nMCode;        /* Manufacturer Code 0x00 1 */
    UINT8  nDCode;        /* Device Code 0x01 1 */
    UINT16 nNumOfBlks;     /* The Number of Blocks 0x02 2 */

    UINT16 nPgsPerBlk;     /* The Number of Pages per Block 0x04 2 */
    UINT16 nBlksInRsv;     /* The Number of Blocks 0x06 2
                           in Reserved Block Pool */

```

```

UINT8  nSctsPerPg;    /* The Number of Sectors per Page 0x08 1 */
UINT8  nNumOfPlane;   /* The Number of Plane 0x09 1 */
UINT8  nMEFlag;       /* Multiple Erase Flag 0x0A 1 */

UINT8  nBWidth;       /* Device Bus Width 0x0C 1 */
UINT8  nBadPos;       /* BadBlock Information Position 0x0E 1 */
UINT8  nLsnPos;       /* LSN Position 0x0E 1 */
UINT8  nEccPos;       /* ECC Value Position 0x0F 1 */

UINT8  aUID[XSR_UID_SIZE]; /* UID 0x10 16,
                                0xFF(absence case of UID) */

UINT16 nTrTime;       /* Physical Read Operation Time 0x20 2 */
UINT16 nTwTime;       /* Physical Write Operation Time 0x22 2 */
UINT16 nTeTime;       /* Physical Erase Operation Time 0x24 2 */
UINT16 nTfTime;       /* Data Transfer Time 0x26 2 */
} LLDSpec;

typedef struct {
    UINT16 nMEListSbn; /* MEList Semi-physical Block Number */
    UINT16 nMEListPbn; /* MEList Physical Block Number */
} LLDMEList;

typedef struct {
    LLDMEList *pstMEList; /* Pointer to LLDMEList */
    UINT16 nNumOfMList; /* Number of Entries of LLDMEList */
    UINT16 nBitMapErr; /* Error Bitmap Position of MEList */
    BOOL32 bFlag; /* Valid Flag */
} LLDMEArg;

/*****
/* LLD Function Table Data Structures
*****/
typedef struct {
    INT32 (*Init) (VOID *pParm);
    INT32 (*Open) (UINT32 nDev);
    INT32 (*Close) (UINT32 nDev);
    INT32 (*Read) (UINT32 nDev, UINT32 nPsn, UINT32 nScts,
                  UINT8 *pMBuf, UINT8 *pSBuf, UINT32 nFlag);
    INT32 (*Write) (UINT32 nDev, UINT32 nPsn, UINT32 nScts,
                  UINT8 *pMBuf, UINT8 *pSBuf, UINT32 nFlag);
    INT32 (*MRead) (UINT32 nDev, UINT32 nPsn, UINT32 nScts,
                  SGL *pstSGL, UINT8 *pSBuf, UINT32 nFlag);
    INT32 (*MWrite) (UINT32 nDev, UINT32 nPsn, UINT32 nScts,
                  SGL *pstSGL, UINT8 *pSBuf, UINT32 nFlag,
                  UINT32 *pErrPsn);
    INT32 (*CopyBack) (UINT32 nDev, CpBkArg *pstCpArg, UINT32 nFlag);
    INT32 (*Erase) (UINT32 nDev, UINT32 Pbn, UINT32 nFlag);
    INT32 (*GetDevInfo) (UINT32 nDev, LLDSpec *pstDevInfo);
    INT32 (*ChkInitBadBlk) (UINT32 nDev, UINT32 Pbn);
    INT32 (*FlushOp) (UINT32 nDev);
    INT32 (*SetRWArea) (UINT32 nDev, UINT32 nSubn, UINT32 nUBlks);
    INT32 (*GetPrevOpData) (UINT32 nDev, UINT8 *pMBuf,
                          UINT8 *pSBuf);
    INT32 (*IOctl) (UINT32 nDev, UINT32 nCmd,
                  UINT8 *pBufI, UINT32 nLenI,
                  UINT8 *pBufO, UINT32 nLenO,
                  UINT32 *pByteRet);
    INT32 (*MErase) (UINT32 nDev, LLDMEArg *pstMEArg,
                  UINT32 nFlag);
    INT32 (*EraseVerify) (UINT32 nDev, LLDMEArg *pstMEArg,

```

```

                                UINT32 nFlag);
} LowFuncTbl;

#endif /* _XSR_LLD_H_ */

```

III. PAM.h

```

#ifndef _XSR_PAM_H_
#define _XSR_PAM_H_

/*****
/* Definitions of PAM
*****/
#define NOT_MAPPED 0xFFFFFFFF /* Device not mapped
                                in platform memory map */

/* ECC Policy Selection */
#define NO_ECC 0 /* No ECC execution */
#define SW_ECC 1 /* ECC execution by SW */
#define HW_ECC 2 /* ECC execution by HW */

/* Lock Scheme Policy Selection */
#define NO_LOCK_SCHEME 0 /* No Lock Scheme execution */
#define SW_LOCK_SCHEME 1 /* Lock Scheme execution by SW */
#define HW_LOCK_SCHEME 2 /* Lock Scheme execution by HW */

/*****
/* return typedef of PAM_GetPAParm()
*****/
typedef struct {
    UINT32 nBaseAddr[XSR_MAX_DEV/XSR_MAX_VOL];
                                /* the base address for accessing NAND device */
    UINT16 nEccPol; /* Ecc Execution Section
                    NO_ECC : No Ecc or another type of ECC
                    SW_ECC : Ecc Execution by XSR SW
                    HW_ECC : Ecc Execution by HW
                    (based on Hamming code) */
    UINT16 nLSchemePol; /* Lock Scheme Policy Section
                        NO_LOCK_SCHEME : No Lock Scheme
                        SW_LOCK_SCHEME : Lock Scheme Execution
                                by SW
                        HW_LOCK_SCHEME : Lock Scheme Execution
                                by HW
                        (if HW has Lock/Unlock functionality) */
    BOOL32 bByteAlign; /* Memory Byte Alignment Problem
                        TRUE32 : Byte Align Problem Free in
                                XSR and LLD
                        (Memory usage increased)
                        FALSE32 : No Action for Byte Align
                                Problem */

    UINT32 nDevsInVol; /* number of devices in the volume */

    VOID *pExInfo; /* For Device Extension.
                    For Extra Information of Device,
                    data structure can be mapped. */
} XsrVolParm;

/*****

```

```

/* Exported Function Prototype of PAM */
/***** */
#ifdef __cplusplus
extern "C" {
#endif /* __cplusplus */

VOID    PAM_Init          (VOID);
VOID    *PAM_GetPAParm    (VOID);
VOID    PAM_RegLFT        (VOID    *pContext);
VOID    PAM_Memcpy        (VOID    *pDst, VOID    *pSrc, UINT32 nLen);

#ifdef ASYNC_MODE
VOID    PAM_InitInt       (UINT32 nLogIntId);
VOID    PAM_BindInt       (UINT32 nLogIntId);
VOID    PAM_EnableInt     (UINT32 nLogIntId);
VOID    PAM_DisableInt    (UINT32 nLogIntId);
VOID    PAM_ClearInt      (UINT32 nLogIntId);
#endif /* ASYNC_MODE */

#ifdef __cplusplus
}
#endif /* __cplusplus */

#endif /* _XSR_PAM_H_ */

```


Index

O

OAM_AcquireSM	59
OAM_BindInt	65
OAM_ClearInt	71
OAM_CreateSM	55
OAM_Debug	54
OAM_DestroySM	57
OAM_DisableInt	69
OAM_EnableInt	67
OAM_Free	48
OAM_GetROLockFlag	81
OAM_GetTime	75
OAM_Idle	80
OAM_InitInt	63
OAM_Malloc	47
OAM_Memcmp	52
OAM_Memcpy	49
OAM_Memset	51
OAM_Pa2Va	78
OAM_ReleaseSM	61
OAM_ResetTimer	73
OAM_WaitNMSec	77

P

PAM_BindInt	172
-------------------	-----

PAM_ClearInt	178
PAM_DisableInt	176
PAM_EnableInt	174
PAM_GetPAParm	162
PAM_Init	161
PAM_InitInt	170
PAM_Memcpy	180
PAM_RegLFT	166

X

XXX_ChkInitBadBlk	143
XXX_Close	90
XXX_CopyBack	135
XXX_Erase	107
XXX_EraseVerify	132
XXX_FlushOp	156
XXX_GetDevInfo	92
XXX_GetPrevOpData	153
XXX_Init	85
XXX_IOCTL	147
XXX_MErase	126
XXX_MRead	110
XXX_MWrite	118
XXX_Open	87
XXX_Read	97
XXX_SetRWArea	145
XXX_Write	102