WinCE MLC Solution (PocketMory) Porting Guide

WinCE 6.0 MLC NAND Solution (PocketMory) Porting Guide

Contact Email : mobilesol.cs@samsung.com

Revision History

Date	Version	Author	Amendment
2006.11.17	0.0	Hyungmin Seo	Master Copy
2006.11.22	0.1	Hyungmin Seo	Preliminary
2006.11.30	0.2	Hyungmin Seo	Preliminary
2006.12.10	0.3	Hyungmin Seo	Preliminary
2006.12.19	0.4	Hyungmin Seo	Preliminary
2006.12.16	0.5	Hyungmin Seo	Preliminary
2007.04.18	0.6	OS Team	Added functions
2007.07.24	0.7	OS Team	Added for WinCE6.0
2007.11.27	0.8	TS Team	Revised for s3c6400 WinCE6.0
2008.02.22	0.81	TS Team	Preliminary
2008.09.16	0.9	OS Team	Revised for s3c6410 WinCE6.0
2009.06.12	1.0	DD Team	Updated
2009.07.28	1.1	DD Team	Updated

Contents

Tables

Figures

1. Overview

1.1 What is MLC NAND Flash?

The MLC NAND Flash has some weakness point compared with SLC NAND.

A. The NOP number is only 1.
   i. The NOP means number of partial program times in one sector.
   ii. It means driver have to write down 1 page size at one times. And it can not be updated any more without erase.
B. The P/E cycle is only 5K or 10K. It depends on MLC NAND device specification.
   i. Therefore the NAND flash device driver has to support strong wear-leveling algorithm.
C. The write performance is slower than SLC NAND flash.
   i. Therefore, it needs write performance enhanced algorithm.
D. The 4bit ECC error can be occurred by 512bytes.
   i. Therefore, nand controller have to support hardware 4bit-ECC logic.
E. Initial bad block detection :
   i. SLC : Samsung makes sure that either the 1st or 2nd page of every initial invalid block has non-FFh data at the column address of 2,048.
   ii. MLC : Samsung makes sure that the last page of every initial invalid block has non-FFh data at the column address of 2,048.

1.2 What is PocketMory™ ?

The PocketMory™ code is based on Whimory solution. We develop it to work on WindowsCE 5.0. The PocketMory has Whimory core part.

1.3 What is Whimory™ ?

Whimory is Samsung Electronics' flash management software. Whimory has same functionalities like other well-know FTL (Flash Translation Layer).

1.4 PocketMory System Architecture

PocketMory exists between the file system and NAND flash memory.

Figure 1-1 shows the system architecture of PocketMory.

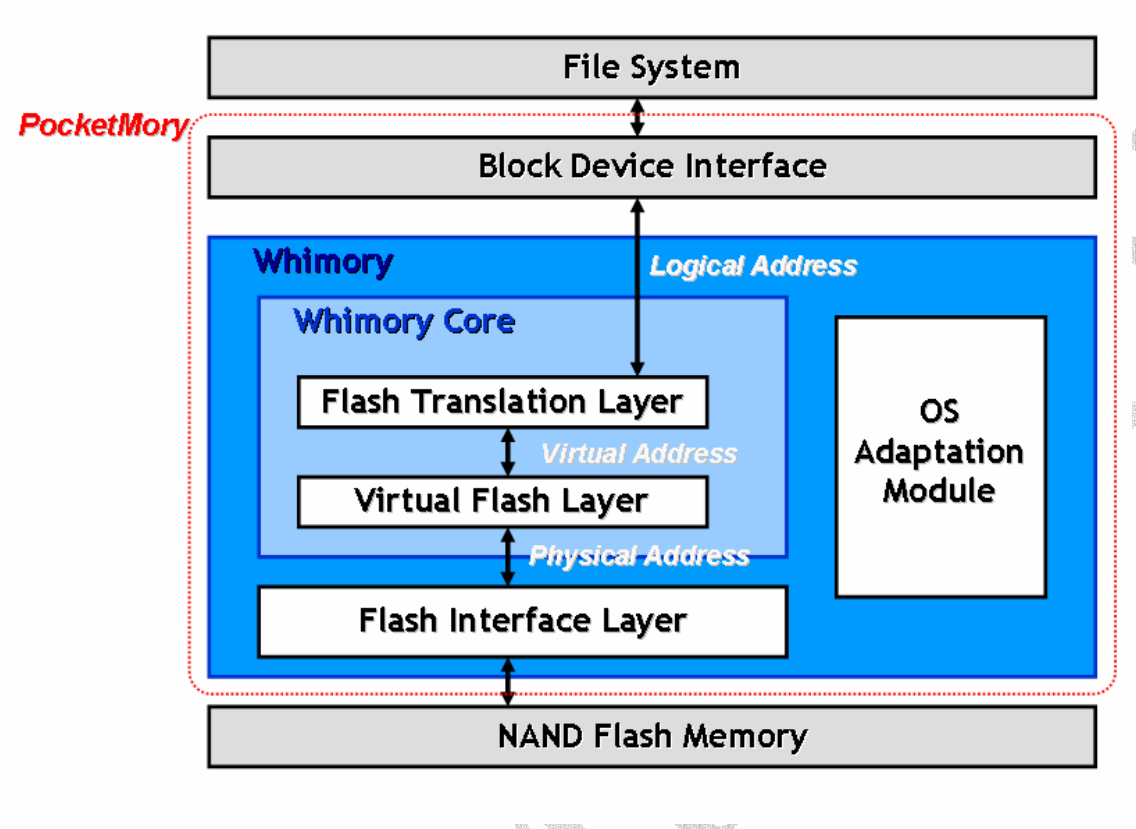


Figure 1-1. PocketMory System Architecture

 Whimory core: Whimory core is composed of two layers: FTL (Flash Translation Layer) at the top and VFL (Virtual Flash Layer) at the bottom. The layers have different features, but they perform the basic functionalities of Whimory as block device emulation and flash memory management. The main features of each layer are as follows.

• FTL: translates a logical address from the file system into the virtual flash address.

• VFL: translates the virtual address from the upper layer into the physical address. At this time, VFL does the address translation considering bad blocks and the number of NAND device in use. VFL accesses FIL, which actually performs read, write, or erase operation, with the physical address.

 OAM: OAM connects Whimory with the OS. OAM needs to be configured according to your OS environment to use NAND flash memory.

 FIL: There is a low level device driver between VFL and NAND flash memory. It reads, writes, or erases data on the physical sector address received from Whimory and is controlled by VFL.

1.5 Whimory Features

Following are the main benefits and features of Whimory implementation.

 It emulates a full block device and manages the data on NAND flash memory efficiently.

 It can be embedded in any kind of product using NAND flash memory, independent of OSs or platform types.

 It guarantees data integrity by managing a bad block and performing error detection or correction.

 It reliably works and reduces the data loss in case of sudden power failure with the advanced algorithms.

 Whimory uses smaller memory size than other FTL software. It can be used in embedded application that has small memory size.

1.6 Definitions and Acronyms

Block	NAND flash memory is partitioned into fixed-sized blocks. A block is 16K bytes or 128K bytes.
VFL	Virtual Flash Layer
FTL (Flash Translation Layer)	A software module which maps between logical addresses and physical addresses when accessing flash memory. FTL is One of the main layer in Whimory
Initial bad block	Invalid blocks upon arrival from the manufacturers
FIL	Flash Interface Layer
NAND flash controller	NAND flash controller is a controller for NAND flash memory
NAND flash device	NAND flash device is a device that contains NAND flash memory or NAND flash controller.
Page	NAND flash memory is partitioned into fixed-sized pages. A page is (512+16) bytes, (2048+64) bytes or (4096+128) bytes.
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.
SLC	Single Level Cell
MLC	Multi Level Cell (Currently there is 4 level in one cell)
Wear-Leveling algorithm	Wear-leveling algorithm distributes the memory allocation in an evenly manner so as to increase the lifetime of NAND flash memory.
LSB	Least Significant Bit (First programmed data in one cell)
MSB	Most Significant Bit (Second programmed data in one cell)
BBM	Bad Block Mapping Manager

Table 1-1 Definitions and Acronyms

2. PocketMory block mapping.

2.1. Sample Block Mapping.

The Figure 2-1 shows the block mapping table when use K9G8G08 1GBytes NAND flash.

Block #	Section	Blocks	Area
0 6	1st/2nd Bootloader(1) + TOC(2) + Eboot(4)	7	WMR_AREA
7 106	OS	100	Special
107 110	VFL Info Section	4	VFL
111 212	Reserved Section	102	
213 222	FTL Info Section	10	FTL
223 239	Free Section (Log Section + Free List)	17	
240 2047	Data Section	1808	

Figure 2-1. NAND flash memory map

- WMR_AREA is combined whith Block0 image and TOC area and Eboot area.
  Block 0 is used for 1st bootloader and 2nd bootloader.
  And Block 0's 126 page is used for initial bad block information and 127 page is used for signature. The signature of current version is "W220".
  TOC block is reserved with 3 blocks and Eboot is reserved with 5 blocks include Bad block reserving.

- The special area is used for OS area.

- The VFL Info Section is used for VFL information.
- The Reserved Section is for replace the blocks when runtime bad block is occurred.

- The FTL Info Section is for FTL information.
- The Free Section is used for write buffer. Log Section needs 14 blocks for the function of LSB recovery. If you do not need it, 7 blocks is needed.
- The Data Section is for storage region for OS.

These values are calculated in CalcGlobal function in WMRGlobal.c file.
And some values are defined in WMRTypes.h and WMRConfig.h file.
We have some tool for calculate the memory map automatically.

3. Porting procedure for WinCE 6.0

3.1. SMDK6410 BSP Directory Structure with PocketMory Solution

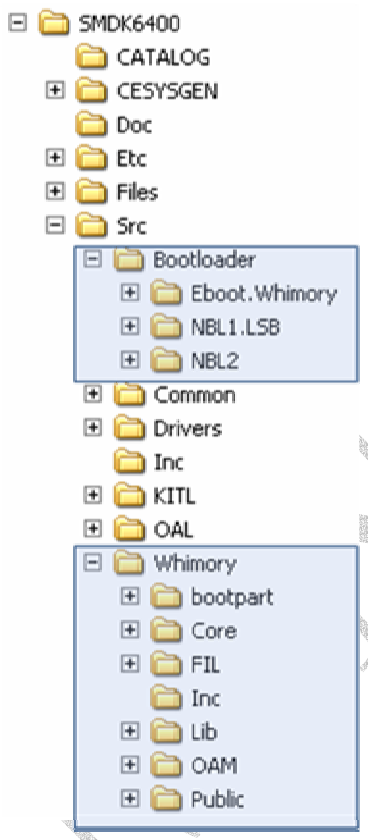


Figure 3-1. PocketMory Directory Structure

The blue boxed code is for PocketMory solution.

- ✓ Eboot.Whimory : The eboot loader directory for support MLC nand flash using PocketMory solution.
- ✓ NBL1 : First Bootloader code.
- ✓ NBL2 : Second Bootloader code.
- ✓ Whimory : PocketMory Directory.

3.2. Copy PocketMory related code to original BSP.

Copy Eboot.Whimory, NBL1, NBL2, Whomory directory to original BSP as Figure 3.1.

3.3. Change DIRS to build PocketMory Solution.

3.3.1. Src\Bootloader\dirs

```
DIRS=   \
```

```
        Eboot.Whimory   \
        NBL1.LSB        \
        NBL2    \
```

3.3.2. Src\

```
DIRS =  whimory     \
        common      \
        oal         \
        kitl        \
        drivers     \
        bootloader
```

3.3.3. Src\Whimory

```
DIRS =  Core    \
        FIL     \
        OAM     \
        Public  \
        BOOTPART
```

3.4. Change BSP for PocketMory

3.4.1. Smdk6410.bat file

Call “src\Whimory\wmrenv.bat” file to set PocketMory related environment.

```
@REM To support iROM NANDFlash boot
set BSP_IROMBOOT=1

@REM Multipl-XIP using demand paging, BINFS must be turned on
set IMGMULTIXIP=1

@REM - To support PocketMory
call %_TARGETPLATROOT%\src\Whimory\wmrenv.bat
```

3.4.2. Sources.cmm file

Add below settings to set PocketMory related environment.

```
!IF "$(WMR_NAND_SUPPORT)" == "MLC"
CDEFINES=$(CDEFINES) -DSUPPORTMLC
!ENDIF
```

```
!if "$(BSP_IROMBOOT)" == "1"
CDEFINES=$(CDEFINES) -D_IROMBOOT_
ADEFINES=$(ADEFINES) -pd "_IROMBOOT_ SETS \"1\""
!endif
```

3.4.3. config.bib file

Change the config.bib file to support multipleXIP function.

```
MEMORY

IF IMGMULTIXIP !

#define     NKNAME      NK          ; for Single NK.bin
#define     RAMNAME             RAM

#define     NKSTART     80100000
#define     NKLEN       03F00000    ; 63MB (Max size, to match image_cfg.* files. This will be
auto-sized)
#define     RAMSTART    84000000
#define     RAMLEN      02500000    ; 37MB (Will be auto-sized from the end of NK)

;   Single XIP
;-------------------------------------------------
;   NAME        ADDRESS     SIZE            TYPE
;-------------------------------------------------
    $(NKNAME)   $(NKSTART)  $(NKLEN)        RAMIMAGE
    $(RAMNAME)      $(RAMSTART) $(RAMLEN)       RAM

ENDIF

IF IMGMULTIXIP

#define     NKNAME          XIPKERNEL
#define     SYSTEMNAME      NK
#define         RAMNAME                     RAM
```

```
IF WINCEDEBUG == debug

#define     NKSTART          80100000
#define     NKLEN            00600000

#define     SYSTEMSTART       80700000
#define     SYSTEMLEN         038FC000

#define     RAMSTART         80700000
#define     RAMLEN           05E00000

ELSE

IF IMGPROFILER

#define     NKSTART          80100000
#define     NKLEN            00600000

#define     SYSTEMSTART       80700000
#define     SYSTEMLEN         038FC000

#define     RAMSTART         80400000
#define     RAMLEN           06100000

ELSE

#define     NKSTART          80100000
#define     NKLEN            00300000

#define     SYSTEMSTART       80400000
#define     SYSTEMLEN         03BFC000

#define     RAMSTART         80400000
#define     RAMLEN           06100000
```

```
ENDIF

ENDIF

#define     CHAINSTART      83FFC000
#define     CHAINLEN        00004000


;   Kernel and RAM region setting
;-------------------------------------------------------
;   NAME            ADDRESS     SIZE            TYPE
;-------------------------------------------------------
    $(NKNAME)           $(NKSTART)              $(NKLEN)            RAMIMAGE
    $(SYSTEMNAME)   $(SYSTEMSTART)      $(SYSTEMLEN)    NANDIMAGE
    $(RAMNAME)              $(RAMSTART)             $(RAMLEN)       RAM
    CHAIN               $(CHAINSTART)   $(CHAINLEN)     RESERVED    ; XIP CHAIN
information

ENDIF               ; IMGMULTIXIP

; Common RAM areas

    AUD_DMA     80002000    00002000        RESERVED
    TEMPS       80010000    00010000        RESERVED
    ARGS        80020800    00000800        RESERVED
    DBGSER_DMA  80022000    00002000        RESERVED
    SER_DMA     80024000    00002000        RESERVED
    IR_DMA      80026000    00002000        RESERVED
    SLEEP       80028000    00002000        RESERVED
    EDBG        80030000    00020000        RESERVED
    CMM         86500000    00300000        RESERVED
    DISPLAY     86800000    00C00000        RESERVED
    MFC_JPEG    87400000    00C00000        RESERVED

CONFIG                      ; Other System Configuration for making image
```

```
    COMPRESSION=ON          ; Binary compression for minimizing download transfer data
    KERNELFIXUPS=ON         ; Kernel address fixup
    AUTOSIZE=ON             ; ROM and RAM size will be resizing automatically for padding region

IF IMGMULTIXIP

    ROM_AUTOSIZE=OFF        ; you can measure how much rom is needed to each binary image if you set this flag as ON
    RAM_AUTOSIZE=OFF        ; RAM size will be resizing automatically only when ROM_AUTOSIZE is ON
    DLLADDR_AUTOSIZE=ON

    XIPSCHAIN=$(CHAINSTART)

    AUTOSIZE_ROMGAP=10000
    AUTOSIZE_DLLADDRGAP=0
    AUTOSIZE_DLLDATAADDRGAP=0
    AUTOSIZE_DLLCODEADDRGAP=0

;
; ROMFLAGS is a bitmask of options for the kernel
;   ROMFLAGS    0x0000
;   ROMFLAGS    0x0001      Disallow Paging
;   ROMFLAGS    0x0010      Trust Module only
;
    ROMFLAGS=0

ELSE

IF IMGTRUSTROMONLY
    ROMFLAGS=10
ELSE
    ROMFLAGS=00
ENDIF   ; END of IMGTRUSTROMONLY
    ROMSTART = $(NKSTART)
    ROMWIDTH = 32
```

```
    ROMSIZE    = $(NKLEN)

ENDIF

IF IMGPROFILER
    PROFILE=ON
ELSE
    PROFILE=OFF
ENDIF
```

To modify the image region to support multipleXIP, you should include MultipleXIP.bib file.

But this setting configuration is only confirmed on our SMDK board with our default catalog items. For more detail information for this issue, please refer to chapter "17 Multiple XIP" in "SMDK6410_WinCE60_PocketMory_PortingGuide.doc" file.

```
IF IMGMULTIXIP
#include "$(_TARGETPLATROOT)\FILES\MultipleXIP.bib"
ENDIF
```

3.4.4. platform.bib file

Remove nandflash.dll from WinCE image.

Add BIBDrv.dll to WinCE Image

```
IF BSP_NONANDFS !
; This is needed in the NK region because it is needed by BINFS to load other regions
;           nandflash.dll   $(_FLATRELEASEDIR)\nandflash.dll       $(XIPKERNEL)    SHK
; block driver that binfs interfaces with
           BIBDrv.dll      $(_FLATRELEASEDIR)\BIBDrv.dll         $(XIPKERNEL)    SHK
ENDIF BSP_NONANDFS !
```

Add OnDisk.dll to WinCE Image.

```
IF BSP_POCKETMORY
           ONDisk.dll      $(_FLATRELEASEDIR)\ONDisk.dll         $(XIPKERNEL)    SHK
ENDIF BSP_POCKETMORY
```

3.4.5. platform.reg file.

Add below settings to support PocketMory storage folder.

```
IF BSP_POCKETMORY
```

```
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\ FDSK]
    "Prefix"="DSK"
    "Dll"="ONDisk.dll"
    "Order"=dword:1
;    "Index"=dword:1
    "IClass"=multi_sz:"{A4E7EDDA-E575-4252-9D6B-4195D48BB865}"
    "Profile"=" FDSK"
    "BmlVolumeId"=dword:0            ; BML volume ID = 0
    "BmlPartitionId"=dword:8         ; BML parition ID = PARTITION_ID_FILESYSTEM
    "WMRStartSector"=dword:0
    "WMRNumOfSector"=dword:10000          ; 32MByte
;    "Flags"=dword:11000              ; do not load again in boot phase 2      ;
    "ONDSectorShift"=dword:2         ; (512 << 2) : 2K Page

[HKEY_LOCAL_MACHINE\System\StorageManager\Profiles\ FDSK]
      "DefaultFileSystem"="FATFS"
      "PartitionDriver"="mspart.dll"
      "Name"="PocketMory MLC Disk"
;      "Folder"="PocketMory"
      "AutoMount"=dword:1
      "AutoPart"=dword:1
      "AutoFormat"=dword:1
;      "MountFlags"=dword:0
;      "Ioctl"=dword:4

[HKEY_LOCAL_MACHINE\System\StorageManager\AutoLoad\ FDSK]
      "DriverPath"="Drivers\\BuiltIn\\FlashDisk"
      ; LoadFlags 0x01 == load synchronously
      "LoadFlags"=dword:1
      "BootPhase"=dword:0

[HKEY_LOCAL_MACHINE\System\StorageManager\Profiles\ FDSK\FATFS]
      "FriendlyName"="PocketMory FAT FileSystem"
;      "Dll"="fatfsd.dll"
      "Flags"=dword:00000014    ; FATFS_ENABLE_BACKUP_FAT | FATFS_DISABLE_AUTOSCAN
```

```
        "Folder"="PocketMory"
        "FormatExfat"=dword:1
;       "EnableCacheWarm"=dword:0
        "CheckForFormat"=dword:1
        "EnableWriteBack"=dword:1

[HKEY_LOCAL_MACHINE\System\StorageManager\AutoLoad\ FDSK\Filters\CacheFilt]
        "Dll"="cachefilt.dll"
        "LockIOBuffers"=dword:1

[HKEY_LOCAL_MACHINE\System\StorageManager\Profiles\ FDSK\FATFS\Filters\CacheFilt]
        "Dll"="cachefilt.dll"
        "LockIOBuffers"=dword:1


;-------------------------------------------------------------------------
; 2nd FAT Area
;-------------------------------------------------------------------------

[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\ FDSK1]
     "Prefix"="DSK"
     "Dll"="ONDisk.dll"
     "Order"=dword:1
;     "Index"=dword:1
     "IClass"=multi_sz:"{A4E7EDDA-E575-4252-9D6B-4195D48BB865}"
     "Profile"="FlashDisk1"
     "BmlVolumeId"=dword:0              ; BML volume ID = 0
     "BmlPartitionId"=dword:9           ; BML parition ID = PARTITION_ID_FILESYSTEM1
     "WMRStartSector"=dword:10000
     "WMRNumOfSector"=dword:ffffffff            ; last location
;     "Flags"=dword:11000                ; do not load again in boot phase 2        ;
     "ONDSectorShift"=dword:2           ; (512 << 2) : 2K Page

[HKEY_LOCAL_MACHINE\System\StorageManager\Profiles\ FDSK1]
        "DefaultFileSystem"="FATFS"
        "PartitionDriver"="mspart.dll"
        "Name"="PocketMory MLC Disk1"
```

```
;       "Folder"="PocketMory1"
        "AutoMount"=dword:1
        "AutoPart"=dword:1
        "AutoFormat"=dword:1
;       "MountFlags"=dword:0
;       "Ioctl"=dword:4

[HKEY_LOCAL_MACHINE\System\StorageManager\AutoLoad\ FDSK1]
        "DriverPath"="Drivers\\BuiltIn\\FlashDisk1"
        ; LoadFlags 0x01 == load synchronously
        "LoadFlags"=dword:1
        "BootPhase"=dword:0

[HKEY_LOCAL_MACHINE\System\StorageManager\Profiles\ FDSK1\FATFS]
        "FriendlyName"="PocketMory FAT FileSystem1"
;       "Dll"="fatfsd.dll"
        "Flags"=dword:00000014    ; FATFS_ENABLE_BACKUP_FAT | FATFS_DISABLE_AUTOSCAN
        "Folder"="PocketMory1"
        "FormatExfat"=dword:1
;       "EnableCacheWarm"=dword:0
        "CheckForFormat"=dword:1
        "EnableWriteBack"=dword:1

[HKEY_LOCAL_MACHINE\System\StorageManager\AutoLoad\ FDSK1\Filters\CacheFilt]
        "Dll"="cachefilt.dll"
        "LockIOBuffers"=dword:1

[HKEY_LOCAL_MACHINE\System\StorageManager\Profiles\ FDSK1\FATFS\Filters\CacheFilt]
        "Dll"="cachefilt.dll"
        "LockIOBuffers"=dword:1

ENDIF BSP_POCKETMORY
```

Add below setting to support PocketMory

```
; HIVE BOOT SECTION
IF BSP_NONANDFS !
```

```
[HKEY_LOCAL_MACHINE\System\StorageManager\AutoLoad\SMFlash]
        "DriverPath"="Drivers\\BlockDevice\\SMFlash"
        "LoadFlags"=dword:1
        "MountFlags"=dword:11
        "BootPhase"=dword:0

[HKEY_LOCAL_MACHINE\Drivers\BlockDevice\SMFlash]
        "Prefix"="DSK"
        "Dll"="BIBDrv.dll"
        "Order"=dword:0
        "Ioctl"=dword:4
        "Profile"="SMFlash"
        "FriendlyName"="Samsung Flash Driver"
        "MountFlags"=dword:11
        "BootPhase"=dword:0

; Bind BINFS to the block driver
[HKEY_LOCAL_MACHINE\System\StorageManager\Profiles\SMFlash]
        "DefaultFileSystem"="BINFS"
        "PartitionDriver"="mspart.dll"
        "AutoMount"=dword:1
        "AutoPart"=dword:1
        "MountFlags"=dword:11
        "Folder"="ResidentFlash"
        "Name"="Samsung Flash Disk"
        "BootPhase"=dword:0
ENDIF ; BSP_NONANDFS

; END HIVE BOOT SECTION
```

3.4.6. ioctl.c file

Path: Src\Oal\Oallib\

Insert IO command for handling PocketMory

```
//------------------------------------------------------------------------------
//
```

```
//    define PSII control
//
#define __PSII_DEFINED__

CRITICAL_SECTION csPocketStoreVFL;

#if defined(__PSII_DEFINED__)
UINT32 PSII_HALWrapper(VOID *pPacket, VOID *pInOutBuf, UINT32 *pResult);
#endif //#if defined(__PSII_DEFINED__)

#define IOCTL_POCKETSTORE_CMD           CTL_CODE(FILE_DEVICE_HAL,           4070,
METHOD_BUFFERED, FILE_ANY_ACCESS)
#define IOCTL_POCKETSTOREII_CMD         CTL_CODE(FILE_DEVICE_HAL,           4080,
METHOD_BUFFERED, FILE_ANY_ACCESS)

BOOL OALIoCtlPostInit(
        UINT32 code, VOID *pInpBuffer, UINT32 inpSize, VOID *pOutBuffer,
        UINT32 outSize, UINT32 *pOutSize)
{
        RETAILMSG(1,(TEXT("[OEMIO:INF]   + IOCTL_HAL_POSTINIT\r\n")));
        InitializeCriticalSection(&csPocketStoreVFL);
        RETAILMSG(1,(TEXT("[OEMIO:INF]   - IOCTL_HAL_POSTINIT\r\n")));

        return TRUE;
}

BOOL OALIoCtlPocketStoreCMD(
        UINT32 code, VOID *pInpBuffer, UINT32 inpSize, VOID *pOutBuffer,
        UINT32 outSize, UINT32 *pOutSize)
{
        BOOL bResult;

        EnterCriticalSection(&csPocketStoreVFL);
        bResult = PSII_HALWrapper(pInpBuffer, pOutBuffer, pOutSize);
        LeaveCriticalSection(&csPocketStoreVFL);
```

```
        if (bResult == FALSE)
        {
                RETAILMSG(1,(TEXT("[OEMIO:INF]      *    IOCTL_POCKETSTOREII_CMD
Failed\r\n")));
                return FALSE;
        }
        return TRUE;
}


//------------------------------------------------------------------------------
//
//    Global: g_oalIoCtlTable[]
//
//    IOCTL handler table. This table includes the IOCTL code/handler pairs
//    defined in the IOCTL configuration file. This global array is exported
//    via oal_ioctl.h and is used by the OAL IOCTL component.
//
const OAL_IOCTL_HANDLER g_oalIoCtlTable[] =
{
#if defined(__PSII_DEFINED__)
{ IOCTL_POCKETSTOREII_CMD,                  0,    OALIoCtlPocketStoreCMD        },
{ IOCTL_HAL_POSTINIT,                       0,    OALIoCtlPostInit              },
#endif //#if defined(__PSII_DEFINED__)

#include "ioctl_tab.h"
};
```

3.4.7. sources file

Path: Src\Oal\Oallib\

Link libraries of PocketMory.

```
INCLUDES=$(INCLUDES);..\..\inc;.\;$(_TARGETPLATROOT)\Src\Whimory\inc;$(_TARGETPLAT
ROOT)\Src\Whimory\Public\inc;$(_TARGETPLATROOT)\Src\Whimory\OAM\OSLess;$(_TARGET
PLATROOT)\Src\Whimory\Core\VFL;
```

```
CDEFINES=$(CDEFINES) -DCEDDK_USEDDKMACRO

!IF "$(BSP_DEBUGPORT)" == "SERIAL_UART0"
CDEFINES=$(CDEFINES) -DDEBUG_PORT=0
!ENDIF

!IF "$(BSP_DEBUGPORT)" == "SERIAL_UART1"
CDEFINES=$(CDEFINES) -DDEBUG_PORT=1
!ENDIF

!IF "$(ENABLE_OAL_ILTIMING)"=="1"
CDEFINES=$(CDEFINES)    -DOAL_ILTIMING
!ENDIF

LDEFINES=-subsystem:native /DEBUG /DEBUGTYPE:CV /FIXED:NO

SOURCELIBS= \

WHIMORYLIB=$(_TARGETPLATROOT)\src\Whimory\Lib

SOURCELIBS=$(SOURCELIBS) \
        $(WHIMORYLIB)\$(_TGTCPU)\$(WINCEDEBUG)\WMRGlobal.lib\
        $(WHIMORYLIB)\$(_TGTCPU)\$(WINCEDEBUG)\FTL_$(WMR_NAND_SUPPORT).lib\
        $(WHIMORYLIB)\$(_TGTCPU)\$(WINCEDEBUG)\VFL_$(WMR_NAND_SUPPORT).lib\
        $(WHIMORYLIB)\$(_TGTCPU)\$(WINCEDEBUG)\PMHALWrapper.lib\
#       $(WHIMORYLIB)\$(_TGTCPU)\$(WINCEDEBUG)\WinCEWMROAM.lib\
        $(WHIMORYLIB)\$(_TGTCPU)\$(WINCEDEBUG)\OSLessWMROAM.lib\
        $(WHIMORYLIB)\$(_TGTCPU)\$(WINCEDEBUG)\$(WMR_FIL)_$(WMR_NAND_SUPP
ORT)_FIL.lib\
        $(WHIMORYLIB)\$(_TGTCPU)\$(WINCEDEBUG)\WMR_Utils.lib\
        $(WHIMORYLIB)\$(_TGTCPU)\$(WINCEDEBUG)\PSIIMDDOAL.lib\
```

3.4.8. image_cfg.h file

Path: Src\Inc\

Modify the Eboot BINFS Buffer area to support PocketMory.

```
#define EBOOT_BINFS_BUFFER_OFFSET               (0x06C00000)
#define                                         EBOOT_BINFS_BUFFER_PA_START
        (DRAM_BASE_PA_START+EBOOT_BINFS_BUFFER_OFFSET)
#define                                         EBOOT_BINFS_BUFFER_CA_START
        (DRAM_BASE_CA_START+EBOOT_BINFS_BUFFER_OFFSET)
#define                                         EBOOT_BINFS_BUFFER_UA_START
        (DRAM_BASE_UA_START+EBOOT_BINFS_BUFFER_OFFSET)
#define EBOOT_BINFS_BUFFER_SIZE                 (0x00480000)
```

3.4.9. image_cfg.inc file

Path: Src\Inc\

Modify the Eboot BINFS Buffer area to support PocketMory.

```
EBOOT_BINFS_BUFFER_OFFSET EQU       (0x06C00000)
EBOOT_BINFS_BUFFER_PA_START         EQU
        (DRAM_BASE_PA_START+EBOOT_BINFS_BUFFER_OFFSET)
EBOOT_BINFS_BUFFER_CA_START         EQU
        (DRAM_BASE_CA_START+EBOOT_BINFS_BUFFER_OFFSET)
EBOOT_BINFS_BUFFER_UA_START         EQU
        (DRAM_BASE_UA_START+EBOOT_BINFS_BUFFER_OFFSET)
EBOOT_BINFS_BUFFER_SIZE             EQU     (0x00480000)
```

3.4.10. Power.c file

Path: Src\Oal\Oallib\

Add the NAND_Init() call function to support PocketMory.

```
#include <windows.h>
#include <bsp.h>

INT32 NAND_Init(VOID);

static void BSPConfigGPIOforPowerOff(void);
static void S3C6410_WakeUpSource_Configure(void);
static void S3C6410_WakeUpSource_Detect(void);

VOID BSPPowerOff()
{
    volatile S3C6410_GPIO_REG *pGPIOReg;
```

```
    volatile S3C6410_ADC_REG *pADCReg;
    volatile S3C6410_RTC_REG *pRTCReg;
    volatile S3C6410_SYSCON_REG *pSysConReg;

    OALMSG(OAL_FUNC, (TEXT("++BSPPowerOff()\n")));

    pGPIOReg = (S3C6410_GPIO_REG*)OALPAtoVA(S3C6410_BASE_REG_PA_GPIO, FALSE);
    pADCReg = (S3C6410_ADC_REG*)OALPAtoVA(S3C6410_BASE_REG_PA_ADC, FALSE);
    pRTCReg = (S3C6410_RTC_REG*)OALPAtoVA(S3C6410_BASE_REG_PA_RTC, FALSE);
    pSysConReg = (S3C6410_SYSCON_REG
*)OALPAtoVA(S3C6410_BASE_REG_PA_SYSCON, FALSE);

    //-----------------------------
    // Wait till NAND Erase/Write operation is finished
    //-----------------------------
    VFL_Sync();

    //-----------------------------
    // Disable DVS and Set to Full Speed
    //-----------------------------
    ChangeDVSLevel(SYS_L0);

    // RTC Control Disable
    pRTCReg->RTCCON = 0x0;          // Subclk 32768 Hz, No Reset, Merged BCD counter,
XTAL 2^-15, Control Disable

    //-------------------------------
    // GPIO Configuration for Sleep State
    //-------------------------------
    BSPConfigGPIOforPowerOff();

    //
    //CLRPORT32(&pIOPort->GPGDAT, 1 << 4);
    //----------------------------
    // Wake Up Source Configuration
    //----------------------------
```

```
    S3C6410_WakeUpSource_Configure();

    OALMSG(OAL_FUNC, (TEXT("--BSPPowerOff()\n")));
}


VOID BSPPowerOn()
{
    OALMSG(OAL_FUNC, (TEXT("++BSPPowerOn()\n")));

// The OEM can add BSP specific procedure here when system power up
    //----------------------------
    // Wake Up Source Determine
    //----------------------------
    S3C6410_WakeUpSource_Detect();

#ifdef _IROM_SDMMC_
    if (!BootDeviceInit())
    {
        OALMSG(OAL_ERROR, (TEXT("[OAL:ERR] BootDeviceInit() returned FALSE\r\n")));
    }
#endif

        // NAND Controller Initialize
        NAND_Init();

    OALMSG(OAL_FUNC, (TEXT("--BSPPowerOn()\n")));
}
```

3.5. For about 0 block guarantee.

The boot code of 1st and 2nd page 4Kbytes is transferred into Steppingstone during reset. Therefore this two pages has to be guaranteed. But the MLC nand flash ECC error incidence is higher than SLC nand flash.

The MLC nand flash's one cell is used by LSB(least significant bit) and MSB(most significant bit).

If we write only LSB, we can use the MLC nand like SLC nand.

This algorithm is for guarantee 0th block.

The below table shows what is LSB and MSB paired page.

Paired Page Address		Paired Page Address	
00h	04h	01h	05h
02h	08h	03h	09h
06h	0Ch	07h	0Dh
0Ah	10h	0Bh	11h
0Eh	14h	0Fh	15h
12h	18h	13h	19h
16h	1Ch	17h	1Dh
1Ah	20h	1Bh	21h
1Eh	24h	1Fh	25h
22h	28h	23h	29h
26h	2Ch	27h	2Dh
2Ah	30h	2Bh	31h
2Eh	34h	2Fh	35h
32h	38h	33h	39h
36h	3Ch	37h	3Dh
3Ah	40h	3Bh	41h
3Eh	44h	3Fh	45h
42h	48h	43h	49h
46h	4Ch	47h	4Dh
4Ah	50h	4Bh	51h
4Eh	54h	4Fh	55h
52h	58h	53h	59h
56h	5Ch	57h	5Dh
5Ah	60h	5Bh	61h
5Eh	64h	5Fh	65h
62h	68h	63h	69h
66h	6Ch	67h	6Dh
6Ah	70h	6Bh	71h
6Eh	74h	6Fh	75h
72h	78h	73h	79h
76h	7Ch	77h	7Dh
7Ah	7Eh	7Bh	7Fh

Figure 3-2. Paired Page Address Information

For example, blue box in above table, 00h and 04h is pared page. It means the 00h page is LSB and the 04h page is MSB. If 00h page is programmed and 04h page is not programmed, this paired page's characteristic is same with SLC NAND. The only red circle is LSB in above picture.

The below code is for program block0image.nb0 to block number 0 in nand.cpp file. The red text is for write LSB page only.

```
dwStartPage = 0;

dwNumPage                                                    =
(dwImageLength)/(BYTES_PER_MAIN_PAGE)+((dwImageLength%BYTES_PER_MAIN_PAGE)? 1 :
```

```
0);
if (SECTORS_PER_PAGE == 8) dwNumPage++;    // page No. 0 and 1 use only 2KByte/Page, so add 1 page.

OALMSG(TRUE, (TEXT("Write Steploader (NBL1+NBL2) image to BootMedia dwNumPage : %d \r\n"),dwNumPage));
OALMSG(TRUE, (TEXT("ImageLength = %d Byte \r\n"), dwImageLength));
OALMSG(TRUE, (TEXT("Start Page = %d, End Page = %d, Page Count = %d\r\n"), dwStartPage, dwStartPage+dwNumPage-1, dwNumPage));

for (dwPage = dwStartPage, nCnt = 0; nCnt < dwNumPage; nCnt++)
{
        if (dwPage < 2)
        {
                if(BYTES_PER_MAIN_PAGE == 2048)//for 2Kpage
                {
                        nSctBitmap = 0xf;
                        nBufCnt = BYTES_PER_SECTOR*4;
                }
                else if(BYTES_PER_MAIN_PAGE == 4096)//for 4Kpage
                {
                        nSctBitmap = 0xff;
                        nBufCnt = BYTES_PER_SECTOR*8;
                }
        }
        else
        {
                nSctBitmap = LEFT_SECTOR_BITMAP_PAGE;
                nBufCnt = BYTES_PER_MAIN_PAGE;
        }

        if (dwPage%PAGES_PER_BLOCK == 0)
        {
                pLowFuncTbl->Erase(0,                    dwPage/PAGES_PER_BLOCK,
enuBOTH_PLANE_BITMAP);
                if (pLowFuncTbl->Sync(0, &nSyncRet) != FIL_SUCCESS)
```

```
                {
                        OALMSG(TRUE, (TEXT("[ERR] FIL Erase Error @ %d block\r\n"),
dwPage/PAGES_PER_BLOCK));
                        OALMSG(TRUE, (TEXT("Write Steploader image to BootMedia
Failed !!!\r\n")));
                        return FALSE;
                }
        }

        OALMSG(OAL_FUNC, (TEXT(" dwPage = 0x%x, pbBuffer = 0x%x \r\n"), dwPage,
pbBuffer));

        if (dwPage < PAGES_PER_BLOCK-2)
        {
        #ifdef _IROMBOOT_
                pLowFuncTbl->Steploader_Write(0, dwPage, nSctBitmap,
enuLEFT_PLANE_BITMAP, pbBuffer, NULL);
        #else
                pLowFuncTbl->Write(0, dwPage, nSctBitmap, enuLEFT_PLANE_BITMAP,
pbBuffer, NULL);
        #endif
        }
        else
        {
                OALMSG(TRUE, (TEXT("Cannot Write image on page %d (Reserved
area) !!!\r\n"), dwPage));
                return FALSE;
        }

        if (pLowFuncTbl->Sync(0, &nSyncRet) != FIL_SUCCESS)
        {
                OALMSG(TRUE, (TEXT("[ERR] FIL Write Error @ %d page\r\n"), dwPage));
                OALMSG(TRUE, (TEXT("Write Steploader image to BootMedia Failed !!!\r\n")));
                return FALSE;
        }
```

```
            // write pages with 0, 1 and 6 to PAGES_PER_BLOCK-3
            dwPage++;
            if(BYTES_PER_MAIN_PAGE == 2048)//for 2Kpage
            {
                        if (IS_MLC && dwPage >= 4 && dwPage < 10) dwPage = 10; //for 8K Stepping
stone
            }
            else if(BYTES_PER_MAIN_PAGE == 4096)//for 4Kpage
            {
                        if (IS_MLC && dwPage >= 2 && dwPage < 10) dwPage = 10; //for 8K Stepping
stone
            }
            pbBuffer += nBufCnt;
}
```

This code will write the page like as below sequence.

00h → 01h → 02h → 03h → (Skip) → 10h → 11h → 12h → 13h → …

From 00h to 03h page data is for NBL1, and from 10h is for NBL2.

The iROM booting can detect and correct bit error for NBL1. To support iROM booting, the NBL1 image have to be written with 8 bit ECC parity code value. The function pLowFuncTbl->Steploader_Write write page data to support iROM booting.

3.6. iROM Booting

In case of S3C6410, this device support iROM booting.

iROM is the internal mask ROM in CPU. It can make it possible to boot up from NAND with 8bit ECC algorithm.

Below pictures are describes boot block assignment.

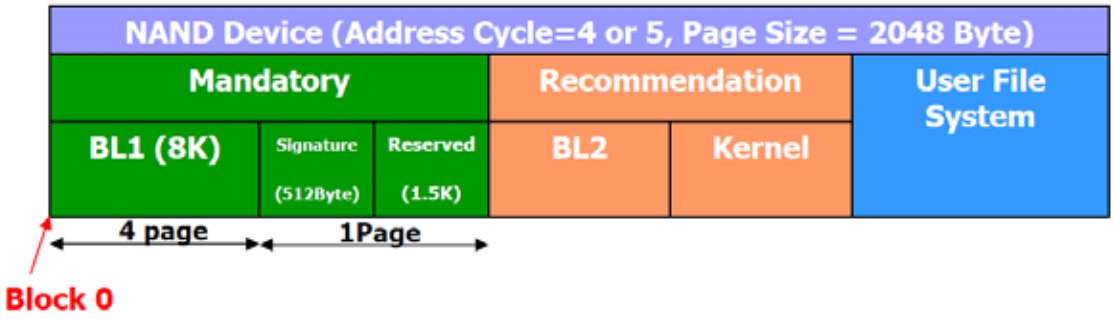


Figure 3-3. NAND Device Boot Block Assignment (Page size = 2048 Byte)

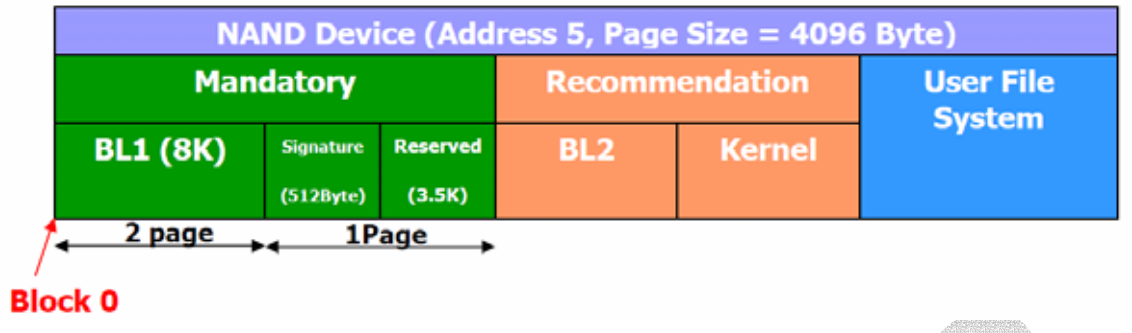


Figure 3-4. NAND Device Boot Block Assignment (Page size = 4096 Byte)

Below tables are describes spare layout for iROM booting. The 8bit ECC algorithm need 13bytes ECC parity code.

0~12(13)	13~25(13)	26~38(13)	39~51(13)
Sector0 ECC	Sector1 ECC	Sector2 ECC	Sector3 ECC

Table 3-1. MLC Spare Area layout 2K page

0~12(13)	13~25(13)	26~38(13)	39~51(13)	52~64(13)	65~77(13)	78~90(13)	91~103(13)
Sector0 ECC	Sector1 ECC	Sector2 ECC	Sector3 ECC	Sector4 ECC	Sector5 ECC	Sector6 ECC	Sector7 ECC

Table 3-2. MLC Page Layout 4K page

4. What have to be modified to change NAND falsh memory map?

4.1. WMR_USER_SUBLKS_RATIO

WMR_USER_SUBLKS_RATIO in WMRConfig.h file.

This value is reserved block ratio for whimory context & reserved section.

The MLC nand flash guaranteed bad block ration is 2.5%.

If you change this value, all block mapping is calculated by automatically.

You can emulate this values using “Whimory_NANDMap.xls” tool in Doc directory.

At now we set this value to 5% in sample BSP.

4.2. WMR_AREA_SIZE

WMR_AREA_SIZE in WMRTypes.h file.

This area is used for 1st Bootloader, 2nd Bootloader, TOC and Eboot.

From 0 to this defined value is not controlled by VFL area. You can only access this area using FIL functions. If you want to change the Eboot area can be controlled by VLF function, you have to move the Eboot area to special area.

4.3. SPECIAL_AREA_SIZE

SPECIAL_AREA_SIZE in WMRTypes.h file.

Special Area is used for OS area. You can change this value to fit your WinCE OS Image size.

And you have to change some definition in the loader.h file for eboot as below values.

The reserved block have to same value with SPECIAL_AREA_START number.

For example, if the SPECIAL_AREA_START number is 5, the RESERVED_BOOT_BLOCKS number have to be 5.

```
//
// OEM Reserved (Nand) Blocks for TOC and various bootloaders
//

// NAND Boot (loads into SteppingStone) @ Block 0
#define NBOOT_BLOCK                 (0)
#define NBOOT_BLOCK_SIZE            (1)
#define NBOOT_SECTOR                BLOCK_TO_SECTOR(NBOOT_BLOCK)

// TOC @ Block 1
#define TOC_BLOCK                   (1)
```

```
#define TOC_BLOCK_SIZE                  (1)
#define TOC_BLOCK_RESERVED              (0)
#define TOC_SECTOR                      BLOCK_TO_SECTOR(TOC_BLOCK)

// Eboot @ Block 2
#define EBOOT_BLOCK                     (2)
#define EBOOT_BLOCK_SIZE                (3)
#define EBOOT_BLOCK_RESERVED            (0)
#define EBOOT_SECTOR                    BLOCK_TO_SECTOR(EBOOT_BLOCK)

#define RESERVED_BOOT_BLOCKS            (NBOOT_BLOCK_SIZE + TOC_BLOCK_SIZE
+TOC_BLOCK_RESERVED + EBOOT_BLOCK_SIZE + EBOOT_BLOCK_RESERVED)

// Images start after OEM Reserved Blocks
#define IMAGE_START_BLOCK               RESERVED_BOOT_BLOCKS
#define IMAGE_START_SECTOR              BLOCK_TO_SECTOR(IMAGE_START_BLOCK)
```

4.4. BLOCKS_PER_BANK

BLOCKS_PER_BANK number is automatically set in FIL code.

For example, S3C6410_FIL.c file defined below nand flash.

```
PRIVATE const DEVInfo stDEVInfo[] = {
        /*****************************************************************************/
        /* Device ID                                                                 */
        /*     Hidden ID                                                             */
        /*          Blocks                                                           */
        /*                Pages per block                                            */
        /*                     Sectors per page                                      */
        /*                        2X program                                         */
        /*                                 2X read                                   */
        /*                                          2x status                        */
        /*                                                   internal Interleaving   */
        /*                                                                 MLC       */
        /*****************************************************************************/
        /* MLC NAND ID TABLE */
        {0xDC, 0x14, 2048, 128, 4, TRUE32,  FALSE32, FALSE32, FALSE32, TRUE32, FALSE32},    /* 4Gb MLC(K9G4G08) Mono */
        {0xD3, 0x55, 4096, 128, 4, TRUE32,  FALSE32, FALSE32, TRUE32,  TRUE32, FALSE32},    /* 8Gb MLC(K9L8G08) DDP  */
        {0xD3, 0x14, 4096, 128, 4, TRUE32,  FALSE32, FALSE32, FALSE32, TRUE32, FALSE32},    /* 8Gb MLC(K9G8G08) Mono */
        {0xD5, 0x55, 8192, 128, 4, TRUE32,  FALSE32, FALSE32, TRUE32,  TRUE32, FALSE32},    /* 16Gb MLC(K9LAG08) DDP  */
        {0xD5, 0x14, 4096, 128, 8, TRUE32,  TRUE32,  TRUE32,  FALSE32, TRUE32, FALSE32},    /* 16Gb MLC(K9GAG08) Mono */
        {0xD5, 0x94, 4096, 128, 8, TRUE32,  TRUE32,  TRUE32,  FALSE32, TRUE32, TRUE32},     /* 16Gb MLC(K9GAG08) Mono */
        {0xD7, 0x55, 8192, 128, 8, TRUE32,  TRUE32,  TRUE32,  TRUE32,  TRUE32, FALSE32},    /* 32Gb MLC(K9LBG08) DDP  */
        {0xD7, 0xD5, 8192, 128, 8, TRUE32,  TRUE32,  TRUE32,  TRUE32,  TRUE32, TRUE32},     /* 32Gb MLC(K9LBG08) DDP  */
};
```

Figure 4-1. DEVInfo structure

The 1st value is device ID number.

The 2nd value is hidden ID number. For example, this value is different with DDP and Mono die.

The 3rd value is the number of total block.

The 4th value is the number of pages per one block.

The 5th value is the number of sectors per one page. One sector is 512 bytes.

The 6th value is for determine it can support 2 plane programming.

The 7th value is for determine it can support 2 plane read.

The 8th value is for determine it can support 2 plane read status.

The 9th value is for determine it can support internal interleaving function.

The 8th value is for determine it is MLC NAND flash or not.

4.5. Comparison Table

Below table is for compare with Mono die MLC nand (K9G8G08), DDP(K9L8G08), QDP(K9HAG08) for 2Kbyte/Page, Mono(K9GAG08) for 4Kbyte/Page.

	K9G8G08	K9G8G08 (2plane)	K9L8G08 (2plane)	K9HAG08 (2plane)	K9GAG08 (2plane)
BANKS_TOTAL	1	1	2	4	1
SUBLKS_TOTAL	4096	2048	1024	1024	2048
BYTES_PER_SECTOR	512				
SECTORS_PER_SUPAGE	4	8	8	8	16
PAGES_PER_SUBLK	128	128	256	512	128
BYTES_PER_SUBLK	0x40000	0x80000	0x100000	0x200000	0x100000
SPECIAL_AREA_SIZE	200	100	50	25	100

Table 4-1 Comparison table for various MLC nand flash

4.6. Physical page write order.

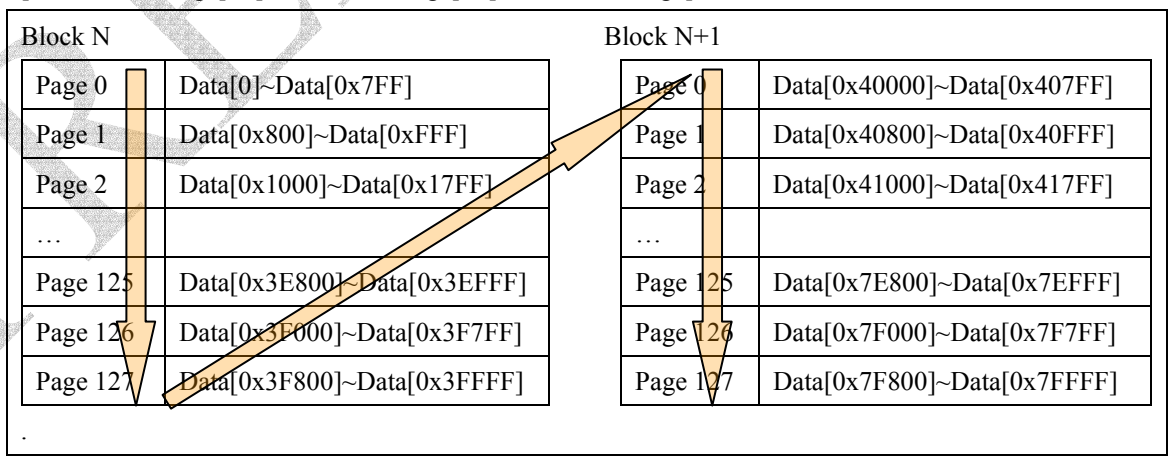
4.6.1. Normal programming

This picture shows the case of normal write. The data will be programmed like as below picture.

The page programming sequence is like as below.

[128 Block 0 Page]→[128 Block 1 Page]→[128 Block 2 Page]→…→[128 Block 127 Page]→

[129 Block 0 Page]→[129 Block 1 Page]→[129 Block 2 Page]→...



.

Figure 4-2. 1-Plane Programming

4.6.2. 2 Plane programming

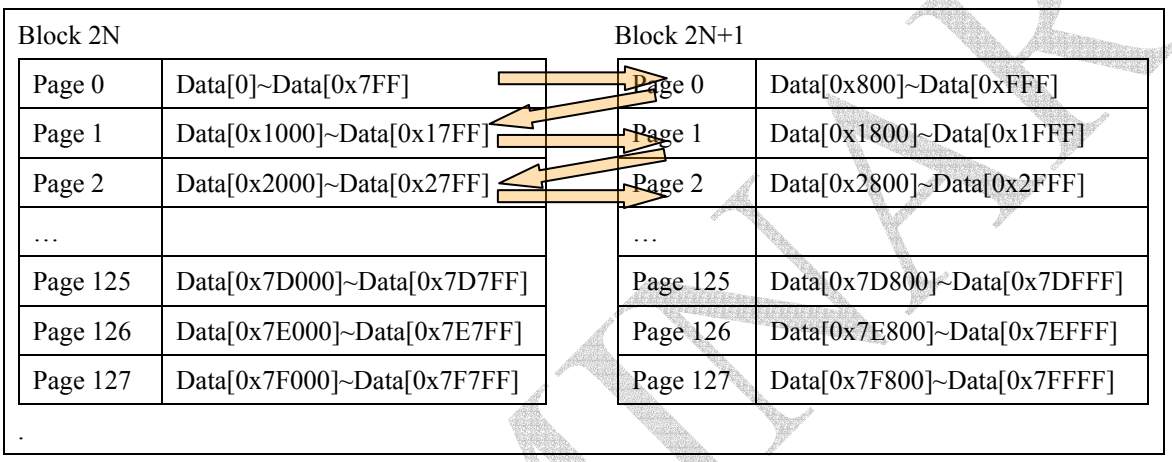
This picture shows the case of 2-Plane programming write. The data will be programmed like as below picture.

The page programming sequence is like as below.

[128 Block 0 Page]→[129 Block 0 Page] →

[128 Block 1 Page]→[129 Block 1 Page] →

[128 Block 2 Page]→[129 Block 2 Page] →…



Block 2N		Block 2N+1	
Page 0	Data[0]~Data[0x7FF]	Page 0	Data[0x800]~Data[0xFFF]
Page 1	Data[0x1000]~Data[0x17FF]	Page 1	Data[0x1800]~Data[0x1FFF]
Page 2	Data[0x2000]~Data[0x27FF]	Page 2	Data[0x2800]~Data[0x2FFF]
…		…	
Page 125	Data[0x7D000]~Data[0x7D7FF]	Page 125	Data[0x7D800]~Data[0x7DFFF]
Page 126	Data[0x7E000]~Data[0x7E7FF]	Page 126	Data[0x7E800]~Data[0x7EFFF]
Page 127	Data[0x7F000]~Data[0x7F7FF]	Page 127	Data[0x7F800]~Data[0x7FFFF]

.

Figure 4-3. 2-Plane Programming

4.6.3. 2 Plane and interleaving programming

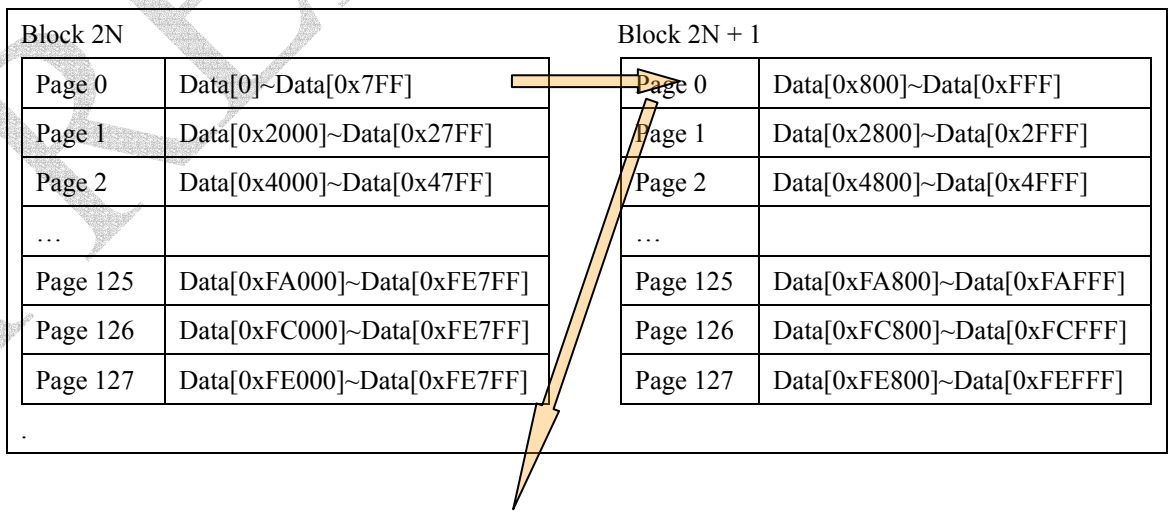
This picture shows the case of 2-Plane and interleaving programming write. The data will be programmed like as below picture.

The page programming sequence is like as below.

[128 Block 0 Page]→[129 Block 0 Page]→[128+4096 Block 0 Page]→[129+4096 Block 0 Page]→

[128 Block 1 Page]→[129 Block 1 Page]→[128+4096 Block 1 Page]→[129+4096 Block 1 Page]→

[128 Block 2 Page]→[129 Block 2 Page]→[128+4096 Block 2 Page]→[129+4096 Block 2 Page]→…



Block 2N		Block 2N + 1	
Page 0	Data[0]~Data[0x7FF]	Page 0	Data[0x800]~Data[0xFFF]
Page 1	Data[0x2000]~Data[0x27FF]	Page 1	Data[0x2800]~Data[0x2FFF]
Page 2	Data[0x4000]~Data[0x47FF]	Page 2	Data[0x4800]~Data[0x4FFF]
…		…	
Page 125	Data[0xFA000]~Data[0xFE7FF]	Page 125	Data[0xFA800]~Data[0xFAFFF]
Page 126	Data[0xFC000]~Data[0xFE7FF]	Page 126	Data[0xFC800]~Data[0xFCFFF]
Page 127	Data[0xFE000]~Data[0xFE7FF]	Page 127	Data[0xFE800]~Data[0xFEFFF]

.

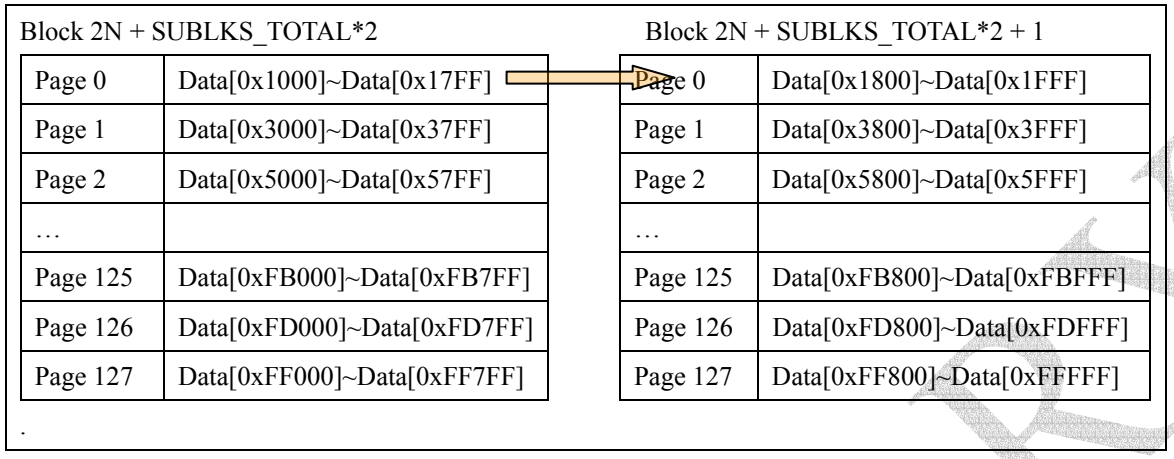


Figure 4-4. 2-Plane, 2Way Programming

The SUBLKS_TOTAL number is block number for 1 plane.

For understanding NAND flash physical map, K9LAG08U0M NAND Flash map is as below.

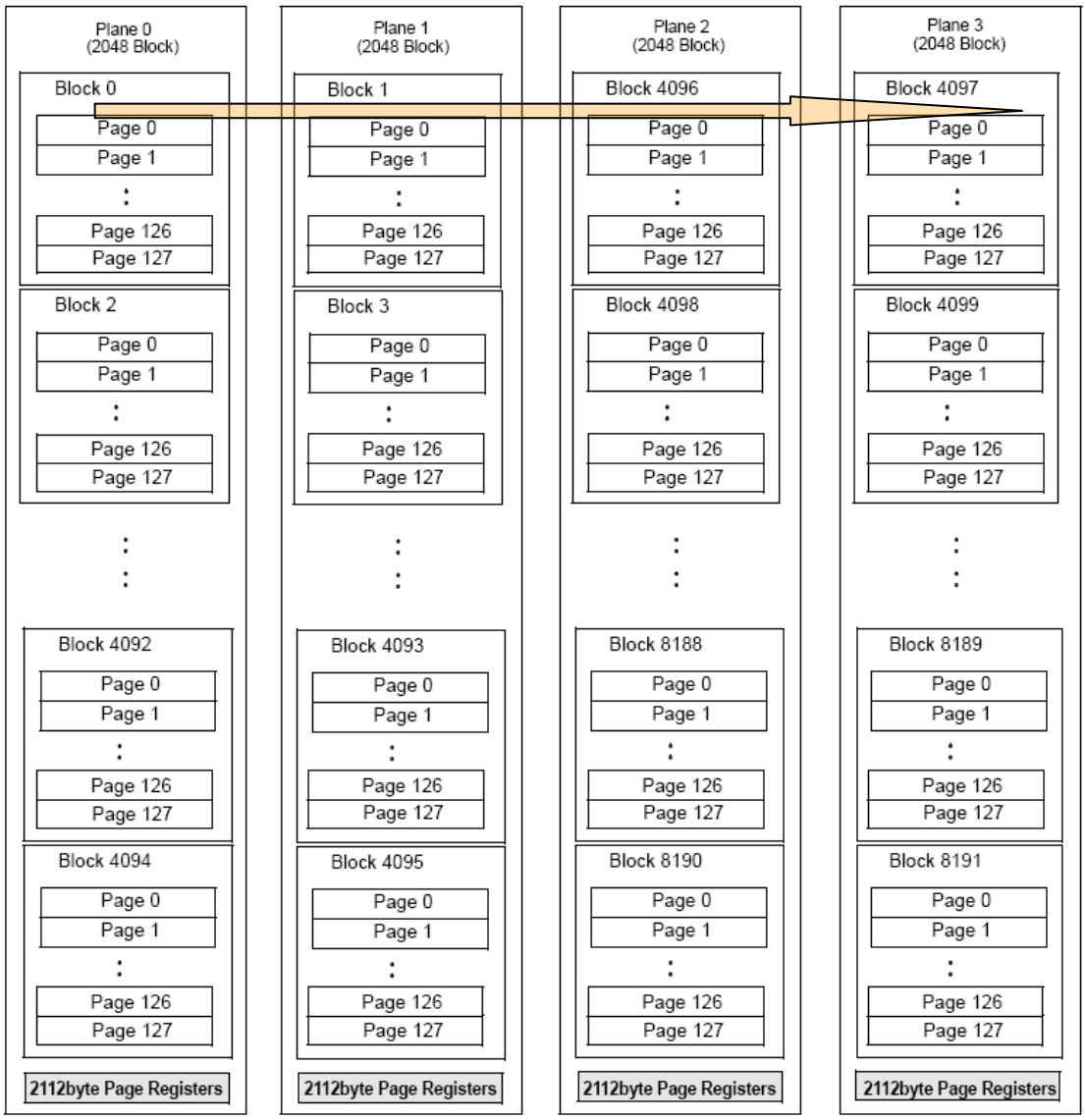


Figure 4-5. K9LAG08U0M Memory Map

K9LAG08U0M is arranged in four 4Gb memory planes. Each plane contains 2,048 blocks and 2112 byte page registers. This allows it to perform simultaneous page program and block erase by selecting one page or block from each plane. The block address map is configured so that two-plane program/erase operations can be executed by dividing the memory array into plane 0~1 or plane 2~3 separately.

For example, two-plane program/erase operation into plane 0 and plane 2 is prohibited. That is to say, two-plane program/erase operation into plane 0 and plane 1 or into plane 2 and plane 3 is allowed

4.6.4. 2 Plane, interleaving, 2CE programming

This picture shows the case of 2-Plane and interleaving programming write. The data will be programmed

like as below picture.

CE0

Block 2N

Page 0	Data[0]~Data[0x7FF]
Page 1	Data[0x4000]~Data[0x47FF]
Page 2	Data[0x8000]~Data[0x87FF]
…	
Page 125	Data[0x1FA000]~Data[0x1FE7FF]
Page 126	Data[0x1FC000]~Data[0x1FE7FF]
Page 127	Data[0x1FC000]~Data[0x1FC7FF]

Block 2N + 1

Page 0	Data[0x800]~Data[0xFFF]
Page 1	Data[0x4800]~Data[0x4FFF]
Page 2	Data[0x8800]~Data[0x8FFF]
…	
Page 125	Data[0x1FA800]~Data[0x1FAFFF]
Page 126	Data[0x1FC800]~Data[0x1FCFFF]
Page 127	Data[0x1FC800]~Data[0x1FCFFF]

.

Block 2N + SUBLKS_TOTAL

Page 0	Data[0x1000]~Data[0x17FF]
Page 1	Data[0x5000]~Data[0x57FF]
Page 2	Data[0x9000]~Data[0x57FF]
…	
Page 125	Data[0x1F6000]~Data[0x1F67FF]
Page 126	Data[0x1F9000]~Data[0x1F97FF]
Page 127	Data[0x1FD000]~Data[0x1FD7FF]

Block 2N + SUBLKS_TOTAL + 1

Page 0	Data[0x1800]~Data[0x1FFF]
Page 1	Data[0x5800]~Data[0x5FFF]
Page 2	Data[0x9800]~Data[0x9FFF]
…	
Page 125	Data[0x1F6800]~Data[0x1F6FFF]
Page 126	Data[0x1F9800]~Data[0x1F9FFF]
Page 127	Data[0x1FD800]~Data[0x1FDFFF]

.

.

CE1

Block 2N

Page 0	Data[0x2000]~Data[0x27FF]
Page 1	Data[0x6000]~Data[0x67FF]
Page 2	Data[0xA000]~Data[0xA7FF]
…	
Page 125	Data[0x1F7000]~Data[0x1F77FF]
Page 126	Data[0x1FA000]~Data[0x1FA7FF]
Page 127	Data[0x1FE000]~Data[0x1FE7FF]

Block 2N + 1

Page 0	Data[0x2800]~Data[0x2FFF]
Page 1	Data[0x6800]~Data[0x6FFF]
Page 2	Data[0xA800]~Data[0xAFFF]
…	
Page 125	Data[0x1F7800]~Data[0x1F7FFF]
Page 126	Data[0x1FA800]~Data[0x1FAFFF]
Page 127	Data[0x1FE800]~Data[0x1FEFFF]

.

Block 2N + SUBLKS_TOTAL

Page 0	Data[0x3000]~Data[0x37FF]
Page 1	Data[0x7000]~Data[0x77FF]
Page 2	Data[0xB000]~Data[0xB7FF]

Block 2N + SUBLKS_TOTAL + 1

Page 0	Data[0x3800]~Data[0x3FFF]
Page 1	Data[0x7800]~Data[0x7FFF]
Page 2	Data[0xB800]~Data[0xBFFF]

…	
Page 125	Data[0x1F8000]~Data[0x1F87FF]
Page 126	Data[0x1FB000]~Data[0x1FB7FF]
Page 127	Data[0x1FF000]~Data[0x1FF7FF]

…	
Page 125	Data[0x1F8800]~Data[0x1F8FFF]
Page 126	Data[0x1FB800]~Data[0x1FBFFF]
Page 127	Data[0x1FF800]~Data[0x1FFFFF]

Figure 4-6. 2-Plane, 4Way Programming

5. PocketMory FIL layer

5.1. Main area and Spare area layout.

- Page Layout for 2Kbytes/Page NAND Flash Device:

0~511	512~1023	1024~1535	1536~2047	2048~2111
Sector 0 512B	Sector 1 512B	Sector 2 512B	Sector 3 512B	Spare Area 64B

Table 5-1. MLC Page Layout 2K page (2048+64 byte)

- Spare Area Layout for 2Kbyte/Page NAND Flash Device:

0(1)	1(1)	2~3(2)	4~15(12)	16~23(8)	24~39(8)	32~39(8)	40~47(8)	48~55(8)	56~63(8)
Bad Mark	Clean Mark	Reserved	Spare Context	Sector0 ECC	Sector1 ECC	Sector2 ECC	Sector3 ECC	Spare ECC	Spare ECC copy

Table 5-2. MLC Spare Area layout 2K page (64 Byte)

- Page Layout for 4Kbytes/Page NAND Flash Device:

0~511	512~1023	1024~1535	1536~2047	2048~2559	2560~3071	3072~3583	3584~4095	4096~4224
Sector 0 512B	Sector 1 512B	Sector 2 512B	Sector 3 512B	Sector 4 512B	Sector 5 512B	Sector 6 512B	Sector 7 512B	Spare Area 128B

Table 5-3. MLC Page Layout 4K page (4096+128 byte)

- Spare Area Layout for 4Kbyte/Page NAND Flash Device: 4bit ECC Device case

0 (1)	1 (1)	2~3 (2)	4~23 (20)	24~31 (8)	32~39 (8)	40~47 (8)	48~55 (8)	56~63 (8)	64~71 (8)	72~79 (8)	80~87 (8)	88~103 (16)	104~119 (16)
Bad Mark	Clean Mark	Reserved	Spare Context	Sector0 ECC	Sector1 ECC	Sector2 ECC	Sector3 ECC	Sector4 ECC	Sector5 ECC	Sector6 ECC	Sector7 ECC	Spare Context ECC	ECC of MECC

Table 5-4. MLC Spare Area layout 4K page (Spare:128 Byte)

The spare area is most important data. In case of 6410, this CPU can support 8bit ECC algorithm and there is available spare area. So “Spare Context” area and “MECC” area are calculated by 8 bit ECC algorithm. The 13bytes of 16bytes are available for 8bit ECC algorithm.

- Spare Area Layout for 4Kbyte/Page NAND Flash Device: 8bit ECC Device case

0 (1)	1 (1)	2~3 (2)	4~23 (20)	24~39 (16)	40~55 (16)	56~71 (16)	72~87 (16)	88~103 (16)	104~119 (16)	120~135 (16)	136~151 (16)	152~167 (16)	168~183 (16)
Bad Mark	Clean Mark	Reserved	Spare Context	Sector0 ECC	Sector1 ECC	Sector2 ECC	Sector3 ECC	Sector4 ECC	Sector5 ECC	Sector6 ECC	Sector7 ECC	Spare ECC	Spare ECC copy

Table 5-5. MLC Spare Area layout 4K page (Spare:218Byte)

The valid data of 8bit ECC algorithm is 13bytes.

Bad Mark : If this value is not 0xFF then this block is Initial or Run-time Bad Block.

Clean Mark : If this page is programmed with any data (include all 0xFF), this value have to mark to 0x00.

Spare Context : VFL context value used by Upper layer.

Sector X ECC : 4-bit ECC parity code for Main area 512 bytes.

Spare ECC : 4-bit ECC parity code for Spare Context 12 bytes.

Spare ECC copy : Back up ECC value.

```
typedef struct {
    UINT8   cBadMark;           // 1 bytes bad mark
    UINT8   cCleanMark;         // 1 Byte clean mark
    UINT8   cReserved[2];       // 2 byte Reserved
    INT32   aSpareData[5];      // 20 bytes spare data, use only 12 byte for 2KByte/Page
    UINT32  aMECC[8*4];         // 32 bytes ECC for Sec0~Sec3 in Main Area,
                                // 64 bytes ECC for Sec4~Sec7 in Main Area for 4K page,
                                // 128 byte for sec0~7 8Bit ECC data
    UINT32  aSECC[8];           // 8 bytes ECC x 2 for Spare Area
} SECCCxt, *pSECCCxt;
```

Figure 5-1. FIL Spare Data Structure

5.2. Erase, Write, Read procedure.

- FIL Erase:

1) Erase command

2) Write Row-address

3) Erase Confirm command

4) Wait for R/B pin Ready

5) Read Status command

6) Read status

The step 4), 5), 6) is separated with NAND_Sync() function. This function is to support 2-plane programming, internal interleaving function to improve Read/Write/Erase performance.

- FIL Write: 2K Page Size case

1) Program command (sector 0) or Random Data Input command (sector 1,2,3)
2) Write address for Sector
3) 4bit ECC engine initialize for encoding
4) 512 byte Sector data write
5) Wait for ECC encoding finished
6) Keep ECC parity code in Memory
7) Loop 1)~6) for Sector 1,2,3
8) Random Data Input command
9) Write address for Spare area
10) Write 1 byte Bad mark (from Upper Layer)
11) Write 1 byte Clean mark (0x00)
12) Write 2 byte Reserved Data (0xFF)
13) 4bit ECC engine initialize for encoding
14) Write 12 byte Spare Context (from Upper layer)
15) Write 8 byte Sector 0 ECC parity code (7 byte is meaningful)
16) Write 8 byte Sector 1 ECC parity code (7 byte is meaningful)
17) Write 8 byte Sector 2 ECC parity code (7 byte is meaningful)
18) Write 8 byte Sector 3 ECC parity code (7 byte is meaningful)
19) CE pin set to High (CE don't care state start)
20) Write 468 byte Dummy Data (0xFF)
21) CE pin set to Low (CE don't care state end)
22) Wait for ECC encoding finished
23) Write 8 byte Spare ECC parity code (7 byte is meaningful)
24) Write 8 byte Spare ECC parity code again.
25) Program Confirm command
26) FIL Sync()

- FIL Write: 4K Page Size case (4bit ECC)

1) Program command (sector 0) or Random Data Input command (sector 1,2,3,4,5,6,7)
2) Write address for Sector
3) 4bit ECC engine initialize for encoding
4) 512 byte Sector data write

5) Wait for ECC encoding finished
6) Keep ECC parity code in Memory
7) Loop 1)~6) for Sector 1,2,3,4,5,6,7
8) Random Data Input command
9) Write address for Spare area
10) Write 1 byte Bad mark (from Upper Layer)
11) Write 1 byte Clean mark (0x00)
12) Write 2 byte Reserved Data (0xFF)
13) 8bit ECC engine initialize for encoding
14) Write 12 byte Spare Context (from Upper layer)
15) CE pin set to High (CE don't care state start)
16) Write 492 byte Dummy Data (0xFF)
17) CE pin set to Low (CE don't care state end)
18) Wait for ECC encoding finished.
19) Keep ECC parity code in Memory.
20) 8bit ECC engine initialize for encoding
21) Write 8 byte Sector 0 ECC parity code (7 byte is meaningful)
22) Write 8 byte Sector 1 ECC parity code (7 byte is meaningful)
23) Write 8 byte Sector 2 ECC parity code (7 byte is meaningful)
24) Write 8 byte Sector 3 ECC parity code (7 byte is meaningful)
25) Write 8 byte Sector 4 ECC parity code (7 byte is meaningful)
26) Write 8 byte Sector 5 ECC parity code (7 byte is meaningful)
27) Write 8 byte Sector 6 ECC parity code (7 byte is meaningful)
28) Write 8 byte Sector 7 ECC parity code (7 byte is meaningful)
29) CE pin set to High (CE don't care state start)
30) Write 448 byte Dummy Data (0xFF)
31) CE pin set to Low (CE don't care state end)
32) Wait for ECC encoding finished
33) Write 16 byte Spare Context ECC parity code (13 byte is meaningful)
34) Write 16 byte ECC parity code of 8 MECC (13 byte is meaningful)
35) Program Confirm command
36) FIL Sync()

- FIL Write: 4K Page Size case(8bit ECC)

1) Program command (sector 0) or Random Data Input command (sector 1,2,3,4,5,6,7)
2) Write address for Sector

3) 8bit ECC engine initialize for encoding
4) 512 byte Sector data write
5) Wait for ECC encoding finished
6) Keep ECC parity code in Memory
7) Loop 1)~6) for Sector 1,2,3,4,5,6,7
8) Random Data Input command
9) Write address for Spare area
10) Write 1 byte Bad mark (from Upper Layer)
11) Write 1 byte Clean mark (0x00)
12) Write 2 byte Reserved Data (0xFF)
13) 8bit ECC engine initialize for encoding
14) Write 12 byte Spare Context (from Upper layer)
15) Write 8 byte Sector 0 ECC parity code (7 byte is meaningful)
16) Write 8 byte Sector 1 ECC parity code (7 byte is meaningful)
17) Write 8 byte Sector 2 ECC parity code (7 byte is meaningful)
18) Write 8 byte Sector 3 ECC parity code (7 byte is meaningful)
19) Write 8 byte Sector 4 ECC parity code (7 byte is meaningful) in case 4Kbyte/Page
20) Write 8 byte Sector 5 ECC parity code (7 byte is meaningful) in case 4Kbyte/Page
21) Write 8 byte Sector 6 ECC parity code (7 byte is meaningful) in case 4Kbyte/Page
22) Write 8 byte Sector 7 ECC parity code (7 byte is meaningful) in case 4Kbyte/Page
23) CE pin set to High (CE don't care state start)
24) Write 364 byte Dummy Data (0xFF)
25) CE pin set to Low (CE don't care state end)
26) Wait for ECC encoding finished
27) Write 16 byte Spare ECC parity code (13 byte is meaningful)
28) Write 8 byte Spare ECC parity code again.
29) Program Confirm command
30) FIL Sync()

- FIL Read: 2K Page size case

1) Read command
2) Write address for Spare area
3) Read Confirm command
4) Read 1 byte Bad Mark
5) Read 1 byte Clean Mark
6) Read 2 byte Reserved Data

7) ECC engine initialize for decoding
8) Read 12 byte Spare Context
9) Read 8 byte Sector 0 ECC parity code (7 byte is meaningful)
10) Read 8 byte Sector 1 ECC parity code (7 byte is meaningful)
11) Read 8 byte Sector 2 ECC parity code (7 byte is meaningful)
12) Read 8 byte Sector 3 ECC parity code (7 byte is meaningful)
13) CE pin set to High (CE don't care state start)
14) Write 468 byte Dummy Data (0xFF)
15) CE pin set to Low (CE don't care state end)
16) Read 8 byte Spare ECC parity code from NAND flash device (7 byte is meaningful)
17) ECC decoding starts automatically
18) Wait for ECC decoding finished (detection and correction)
19) If there is uncorrectable Error, repeat from 7) to 15) and read 8 byte more for compare with next 8 byte spare ECC copy code.
20) Write address for Main area.
21) Read 512 byte.
22) CE pin set to High (CE don't care state start)
23) Write 7 byte ECC parity code which read from 9) to 12)
24) CE pin set to Low (CE don't care state end)
25) Wait for ECC decoding finished (detection and correction)
26) Read Next Sector with same method. (20 ~ 25)

- FIL Read: 4K Page size case(4bit ECC)

1) Read command
2) Write address for Spare Context ECC area
3) Read Confirm command
4) Read 16bytes Spare Context ECC parity data
5) Read 16bytes ECC parity data of MECC(0~7)
6) Move address to spare area using Random data output.
7) Read 1 byte Bad Mark
8) Read 1 byte Clean Mark
9) Read 2 byte Reserved Data
10) 8bit ECC engine initialize for decoding
11) Read 12 byte Spare Context
12) CE pin set to High (CE don't care state start)
13) Write 492 byte Dummy Data (0xFF)

14) Write ECC Parity Code which read at 4th step
15) CE pin set to Low (CE don't care state end)
16) ECC decoding starts automatically
17) Wait for ECC decoding finished (detection and correction)
18) 8bit ECC engine initialize for decoding
19) Read 8 byte Sector 0 ECC parity code
20) Read 8 byte Sector 1 ECC parity code
21) Read 8 byte Sector 2 ECC parity code
22) Read 8 byte Sector 3 ECC parity code
23) Read 8 byte Sector 4 ECC parity code
24) Read 8 byte Sector 5 ECC parity code
25) Read 8 byte Sector 6 ECC parity code
26) Read 8 byte Sector 7 ECC parity code
27) CE pin set to High (CE don't care state start)
28) Write 448 byte Dummy Data (0xFF)
29) CE pin set to Low (CE don't care state end)
30) Write ECC Parity Code which read at 5th step.
31) ECC decoding starts automatically
32) Wait for ECC decoding finished (detection and correction)
33) Write address for Main area.
34) Read 512 byte.
35) CE pin set to High (CE don't care state start)
36) Write 7 byte ECC parity code which read from 19) to 26)
37) CE pin set to Low (CE don't care state end)
38) Read Next Sector with same method. (34~37)

- FIL Read: 2K Page size case

1) Read command
2) Write address for Spare area
3) Read Confirm command
4) Read 1 byte Bad Mark
5) Read 1 byte Clean Mark
6) Read 2 byte Reserved Data
7) 8bit ECC engine initialize for decoding
8) Read 12 byte Spare Context
9) Read 16 byte Sector 0 ECC parity code (13 byte is meaningful)

10) Read 16 byte Sector 1 ECC parity code (13 byte is meaningful)
11) Read 16 byte Sector 2 ECC parity code (13 byte is meaningful)
12) Read 16 byte Sector 3 ECC parity code (13 byte is meaningful)
13) Read 16 byte Sector 4 ECC parity code (13 byte is meaningful)
14) Read 16 byte Sector 5 ECC parity code (13 byte is meaningful)
15) Read 16 byte Sector 6 ECC parity code (13 byte is meaningful)
16) Read 16 byte Sector 7 ECC parity code (13 byte is meaningful)
17) CE pin set to High (CE don't care state start)
18) Write 364 byte Dummy Data (0xFF)
19) CE pin set to Low (CE don't care state end)
20) Read 13 byte Spare ECC parity code from NAND flash device
21) ECC decoding starts automatically
22) Wait for ECC decoding finished (detection and correction)
23) If there is uncorrectable Error, repeat from 7) to 19) and read 13 byte more for compare with next 13 byte spare ECC copy code.
24) Write address for Main area.
25) Read 512 byte.
26) CE pin set to High (CE don't care state start)
27) Write 7 byte ECC parity code which read from 9) to 16)
28) CE pin set to Low (CE don't care state end)
29) Read Next Sector with same method.

5.3. User setting values

5.3.1. OS_SCAN_RATIO

This value is used for scanning OS area. The OS areas are scanned in VFL_Open and FTL_Scan function. The reason of scanning of OS is that the OS area can be disturbed by "read disturbance" problem. The "Read Disturbance" problem means, some sector 512bytes bit error can be increased more than 4bit or 8bit which described on NAND flash specification by read operation of other sectors. For example, the page number 10 can have "Uncorrectable ECC" error by read other pages in same block. The ECC error bit number is increased, but unfortunately, if this page number 10 is not read until over its spec of ECC then the device met "Uncorrectable ECC" error.

To avoid this problem, the OS area scan function is available by VFL_Open and FTL_Scan.

This OS_SCAN_RATIO is used to set the scan block number for each scan function.

The percentage of scan range is calculated by below numerical formula.

Scan percentage of all OS area = 100 / OS_SCAN_RATIO

If OS_SCAN_RATIO value is 0, the OS area scan function is not working in VFL_Open and FTL_Scan

function.

5.3.2. FS_SCAN_RATIO

The FS_SCAN_RATIO value is used for same purpose of OS_SCAN_RATIO.

This value is defined in FIL function like as below.

```
// FS_SCAN_RATIO means scan percentage of Total FTL size.
// If FS_SCAN_RATIO value is 0, skip Scan.
if ( stDEVInfo[nScanIdx].nNumOfBlocks <= 1024 )
    FS_SCAN_RATIO = 20;      // 10 means (100/10)% => 10%
else if ( stDEVInfo[nScanIdx].nNumOfBlocks <= 2048 )
    FS_SCAN_RATIO = 50;      // 20 means (100/20)% => 5%
else if ( stDEVInfo[nScanIdx].nNumOfBlocks <= 4096 )
    FS_SCAN_RATIO = 100;     // 50 means (100/50)% => 2%
else if ( stDEVInfo[nScanIdx].nNumOfBlocks <= 8192 )
    FS_SCAN_RATIO = 200;     // 100 means (100/100)% => 1%
else
    FS_SCAN_RATIO = 400;     // 200 means (100/200)% => 0.5%
```

This value can effect to booting time. Because if the FS_SCAN_RATIO value is small (scan range is wide) then it needs more read operation for each scan function.

So we recommend bigger number for huge size NAND flash to reduce booting time.

5.3.3. CRITICAL_READ_CNT

This value is used to determine the block is needed reclaim or not.

The read disturbance problem can be happen by read operation.

If some page is read many times, the other page of same block can be disturbed.

If the disturbed page can be read by VFL layer when the ECC error is less that 4 bit, then this page can be reclaimed. But this disturbed page is not read VFL layer, this page can not be reclaimed.

To prevent this problem, the VFL layer has read count of every block. And if the read count of block is more that specific number, this block enter to queue of reclaim.

This specific number is CRITICAL_READ_CNT.

This number has to be set by FIL code like as below.

```
// CRITICAL_READ_CNT value is used to avoid read disturbance problem.
// This value is critical count value to determine the count of read for reclaim.
// Every block have each read count value, this value is reset to 0 when boot up the system.
// If each block is read more than CRITICAL_READ_CNT value after boot up, this block is enter to reclaim list.
```

```
// This value is useful if the system is running long times without reboot.
// But system have to calls FTL_Reclaim function during running the system.
CRITICAL_READ_CNT = 100000; // 0; for no operation.
```

If the read count is bigger than CRITICAL_READ_CNT, that block number is enter to queue.

To reclaim this block, the FTL_Reclaim function has to be called.

So some thread have to calls FTL_Reclaim function.

Please refer to below sample code.

```
DWORD WINAPI CallFTL_Reclaim(LPVOID lpParameter)
{
        unsigned long dwStartTick, dwIdleSt, dwStopTick, dwIdleEd, PercentIdle;
        int i;
        while(1)
        {
                dwStartTick = GetTickCount();
                dwIdleSt = GetIdleTime();
                Sleep(1000); // 1 Seconds
                dwStopTick = GetTickCount();
                dwIdleEd = GetIdleTime();
                PercentIdle = ((100*(dwIdleEd -dwIdleSt)) / (dwStopTick - dwStartTick));
                if ( PercentIdle > 90 )
                {
                        RETAILMSG(1,(TEXT("Call FTL_ReadReclaim!!!\r\n")));
                        FTL_ReadReclaim();
                }
        }
}
```

But if your device can calls FTL_Scan() function by periodically, this option is not need. Then please set the CRITICAL_READ_CNT value to 0. Because those solutions are made for same purpose (Avoid Read disturbance problem.).

6. Tested device of current device driver

1. PocketMory SLC solution is not supported. (Only support MLC flash device)
2. We test the BSP with below nand flash part number.
   A. K9G8G08/K9G4G08 (Mono Die, support 2 plane programming)
   B. K9L8G08 (DDP, support 2 plane programming, internal interleaving)
   C. K9LAG08 (DDP, support 2 plane programming, internal interleaving)
   D. K9HAG08 (QDP, support 2 plane programming, internal interleaving, 2CE programming)
   E. K9GAG08U0M (Mono, support 2 plane programming, 4KByte/Page)
   F. K9LBG08 (DDP, support 2 plane programming, internal interleaving, 4KByte/Page)
   G. K9HCG08 (QDP, support 2 plane programming, internal interleaving, 2CE programming, 4K Byte/Page)
   H. K9GAG08U0D (Mono, support 2 plane programming, 4KByte/Page, 8bit ECC device)