Using the NAND Flash Controller on Blackfin® Processors

Contributed by Gurudath Vasanth

Rev 1 – September 15, 2008

Introduction

Analog Devices Blackfin® processors are targeted to be widely used in many handheld products such as PDAs, smart phones and PDA phones, digital cameras, and many other portable media devices. These products usually store multi-media content such as movie clips, pictures, and music. NAND flash is most commonly used for storing such high-density data. The processor's NAND flash controller simplifies the memory interface design and eliminates the need for external on-board glue logic. It also reduces the overhead on application software and thus improves the performance to a large extent.

This EE-Note discusses the on-chip NAND flash controller (NFC) on ADSP-BF52x and ADSP-BF54x Blackfin processors. It covers software driver development and the algorithms that are required to manage the NAND flash memories. Example code associated with this EE-Note includes block erase, page program, page read, random data input, random data output, and copyback operations in addition to managing the bad blocks. The driver code is written for the flash devices from ST Microelectronics available on ADSP-BF527 EZ-KIT Lite® and ADSP-BF548 EZ-KIT Lite boards. The EE-Note concludes with guidelines to re-use the drivers for SLC NAND flash devices from other vendors. Appendix A summarizes the differences in NFC between ADSP-BF52x and ADSP-BF54x processors.

NAND Flash

NAND flash is arranged as blocks, and each block contains many pages. For example, NAND02G-B2C from ST Microelectronics used on the ADSP-BF548 EZ-KIT Lite board contains 2048 blocks, and each block contains 64 pages. The page size is 2048 bytes of main area used to store data and 64 bytes of spare area which is commonly used to store ECC, software flags, and so on.

Unlike hard disks, NAND flash memories can only be programmed one page at a time and they can only be re-programmed after the entire block containing the page has been erased. Also, NAND flash has a restriction of a limited number of erasures per block. Usually, the number of program/erase cycles will vary from 10,000 to 100,000 times, after which a block is not reliable to store data. Due to these constraints, a sophisticated algorithm is used to make the NAND flash appear to the system as an ideal storage device. This section discusses bad block management and wear leveling algorithms that extend the life of the NAND flash devices. For more information on NAND flash memories, refer to *http://en.wikipedia.org/wiki/Flash_memory* [6].

Bad Block Management

Some of the blocks of NAND flash shipped from the manufacturer are marked as “bad” and some blocks can become bad during the course of use. NAND flash driver code must recognize these bad blocks and be able to avoid using them by the application software.

For example, on ST Microelectronics flash devices, any block where the 1st and 6th bytes (or 1st word, in the spare area of the 1st page) does not contain 0xFF, is treated as a bad block. So, these blocks should be identified first before any erase operation is performed. A table containing the information of invalid blocks should be maintained. Any blocks that become invalid during the course of use should be updated in the bad block table.

The associated code first checks for the existence of a bad block table. If it is not created, the code checks for the invalid blocks marked by the manufacturer. This table is usually kept in block 0 as all the vendors guarantee block 0 as a valid block. (Associated code provides users with an option to keep the bad block table in any block). This table is loaded to internal memory for further updating when the erase/program operation fails or when there is error in the error checking and correction (ECC) itself during a read operation. Figure 1 shows the flowchart of bad block management implementation in the sample code.

Since the NAND boot starts from block 0, the bad block table in the associated code can be kept in a block other than block 0, so as to evaluate NAND boot feature as well. By default, the table is kept in block 5.

The NAND flash file system sample code provided with the VisualDSP++® 5.0 Update 4 development tools, by default, reserves a 10-block region at the start of the array, which is not managed by the file translation layer. This is intended to allow the user to evaluate NAND boot or for other purposes. Hence, it is recommended that users use this space to evaluate the sample code provided with this EE-Note. Refer to the `readme` file of the NAND flash file system example in the VisualDSP++ installation for more details.

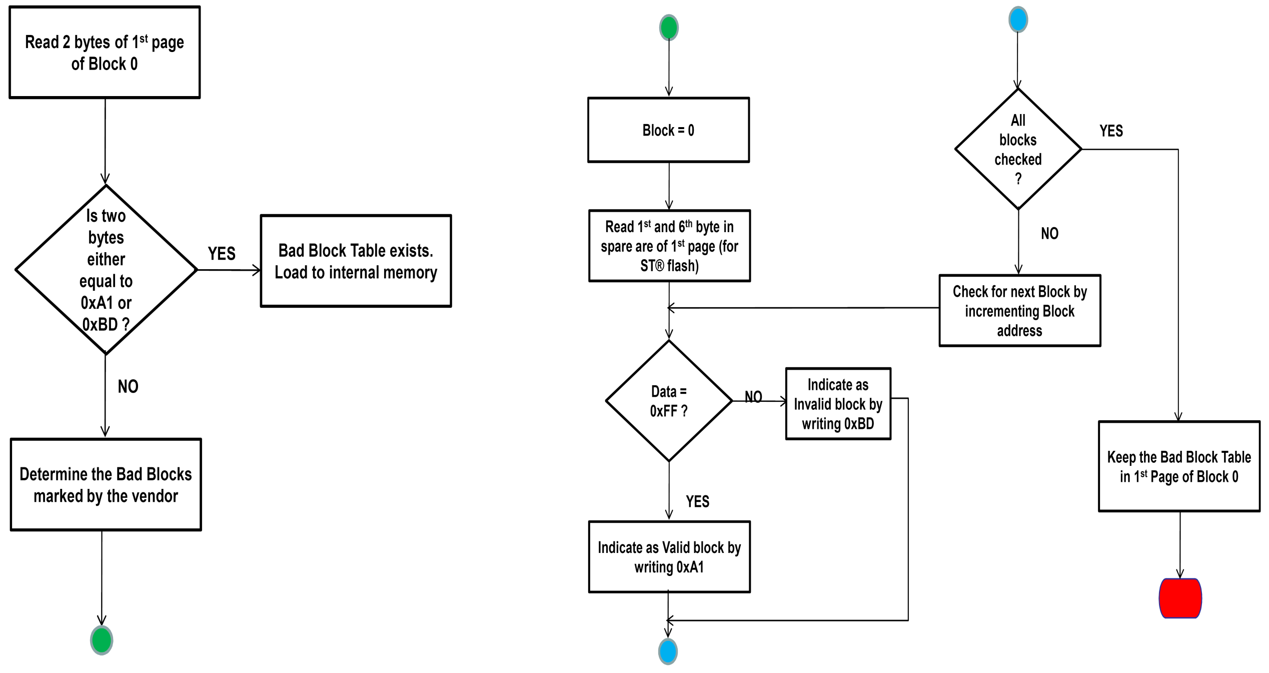

Read 2 bytes of 1st page of Block 0
Is two bytes either equal to 0xA1 or 0xBD ?
YES
Bad Block Table exists. Load to internal memory
NO
Determine the Bad Blocks marked by the vendor
Block = 0
Read 1st and 6th byte in spare are of 1st page (for ST® flash)
Data = 0xFF ?
NO
Indicate as Invalid block by writing 0xBD
YES
Indicate as Valid block by writing 0xA1
All blocks checked ?
YES
NO
Check for next Block by incrementing Block address
Keep the Bad Block Table in 1st Page of Block 0

Figure 1. Flowchart of bad block detection

Wear Leveling

NAND flash has a special property in that any page to be written into a block must be erased. Usually, the number of program/erase cycles will vary from 10,000 to 100,000 times, after which the memory cells wear out. For write-intensive applications, it is recommended to implement a wear-leveling algorithm to monitor and spread the number of write cycles per block. This way all the blocks will undergo an equal number of erase cycles, thereby extending the life of the device.

There are many methods to implement a wear-leveling algorithm. The basic requirement is to spread the number of program/erase cycles across all the blocks. For example, a round-robin method can be used to select the physical blocks. Another method is to maintain a vector having the number of program/erase cycles undergone by each block. The algorithm should map the virtual block to the youngest physical block. A highly reliable system should implement a sophisticated algorithm to extend the life and data retention of the device.

NAND Flash File System

Most of the portable and handheld systems require high density to store large multimedia files, and NAND flash suits these applications. The main characteristics of NAND flash is that bits can be cleared only by erasing a large block of memory, and each block can sustain a limited number of erase cycles. Due to these limitations, the existing file systems cannot be used directly to support NAND flash. Sophisticated algorithms must be used in file systems to overcome these limitations and effectively use the flash memory. This can be achieved either by having a translation layer between the existing file system and flash device to mask the differences, or by developing a customized file system to suit the flash properties. A file translation layer would basically perform block-to-sector mapping, reclamation (garbage collection), and wear leveling. This layer communicates with the low-level drivers which includes ECC and bad block management.

Many file systems such as JFFS (Journaling Flash File System), JFFS2, and Trimble File Systems are feasible to use with NAND flash. Among these, YAFFS (Yet Another Flash File System) by Aleph One is only one file system that is designed specifically for NAND flash memories under any operating system. It uses an efficient map structure to map file locations to physical address of NAND flash.

NAND Flash Drivers

This section discusses how to implement the basic drivers such as erase/program/read. The discussion starts with an overview of NFC on ADSP-BF52x and ADSP-BF54x processors.

A Quick Look at NFC

The key features of NFC include:

- Support for SLC NAND flash devices with page sizes of 256 and 512 bytes. Larger page sizes are supported through software
- Supports DMA transfer between internal memory and NAND flash device

- Supports error checking and correction (ECC) that allows the detection of multiple-bit and correction of 1-bit errors

Along with these features, NFC provides a configurable set-up time for writing and reading data to easily suit the requirements of the flash device interfaced with the processor. The page size, NAND width, and set-up times are configured in the `NFC_CTL` register. The module is enabled by configuring the `PORTx_FER` and `PORTx_MUX` registers appropriately. For more information on NFC of ADSP-BF52x and ADSP-BF54x processors, refer to *ADSP-BF52x Blackfin Processor Hardware Reference (Volume 2 of 2)* [8] and the *ADSP-BF54x Blackfin Processor Peripheral Hardware Reference.* [9]

Block Erase

Before writing to the page, the block to which the page belongs must be erased. First, the block erase set-up must be issued by writing to the `NFC_CMD` register. This is followed by the address cycle providing the block address by writing to the `NFC_ADDR` register. Finally, the block erase confirm code must be issued. An interrupt can be set up to detect the rising edge on the `ND_RB/` signal, which indicates that the block erase operation is completed. Figure 2 shows the flow chart for the block erase operation.

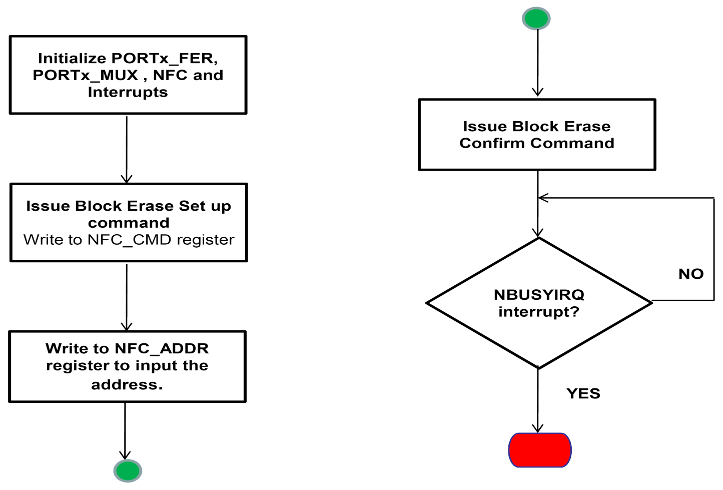

Initialize PORTx_FER, PORTx_MUX , NFC and Interrupts
Issue Block Erase Set up command Write to NFC_CMD register
Write to NFC_ADDR register to input the address.
Issue Block Erase Confirm Command
NBUSYIRQ interrupt?
NO
YES

Figure 2. Flow chart of block erase operation

```
BlockErase(BlockNo);                          // Pass the BlockNo to be erased
```

Listing 1. API to erase a block

Page Write

The page program operation is the standard sequential operation to program data to the memory array. All the flash commands are issued by writing to the `NFC_CMD` command register.

First, the page program set-up code command must be issued. This is followed by the address cycle by writing to the `NFC_ADDR` register. The number of address bus cycles to be issued depends on the size of the NAND flash device used. For example, a 2-Gbit NAND flash device requires 5 address bus cycles. The flash device data sheet gives the details about the address bus cycle required for the specific operation.

The page write start bit in the `NFC_PGCTL` register must be set; this initiates the DMA transfers to complete the page write. After writing all of the data, software can append the ECC values from the ECC registers to store them in the spare area of the NAND flash. The ECC for a 256-byte page will be available in the `NFC_ECC0` and `NFC_ECC1` registers. For a 512-byte page, the ECC values in the `NFC_ECC2` and `NFC_ECC3` registers should also be considered. Finally, page program confirm command is issued to initiate the programming, which asserts the `ND_RB/` signal.

Once the operation is complete, the `ND_RB/` signal de-asserts and a rising edge on this signal will set the bit in the `NFC_STAT` register. An interrupt can be set up to detect the rising edge of the `ND_RB/` signal, which indicates the page write operation is completed.

Large page size is supported in software. For example, a 2K-byte page can be treated as four 512-byte pages. The ECC generated for every 512-byte page is temporarily stored and finally written to the spare area during page write operation. Software must reset the ECC registers by writing to the `NFC_RST` register before starting the next 512-byte transfer.

Figure 3 shows the flow chart for a page write operation. The example code associated with this EE-Note demonstrates a page write operation to a flash device that has a 2048-byte page size.

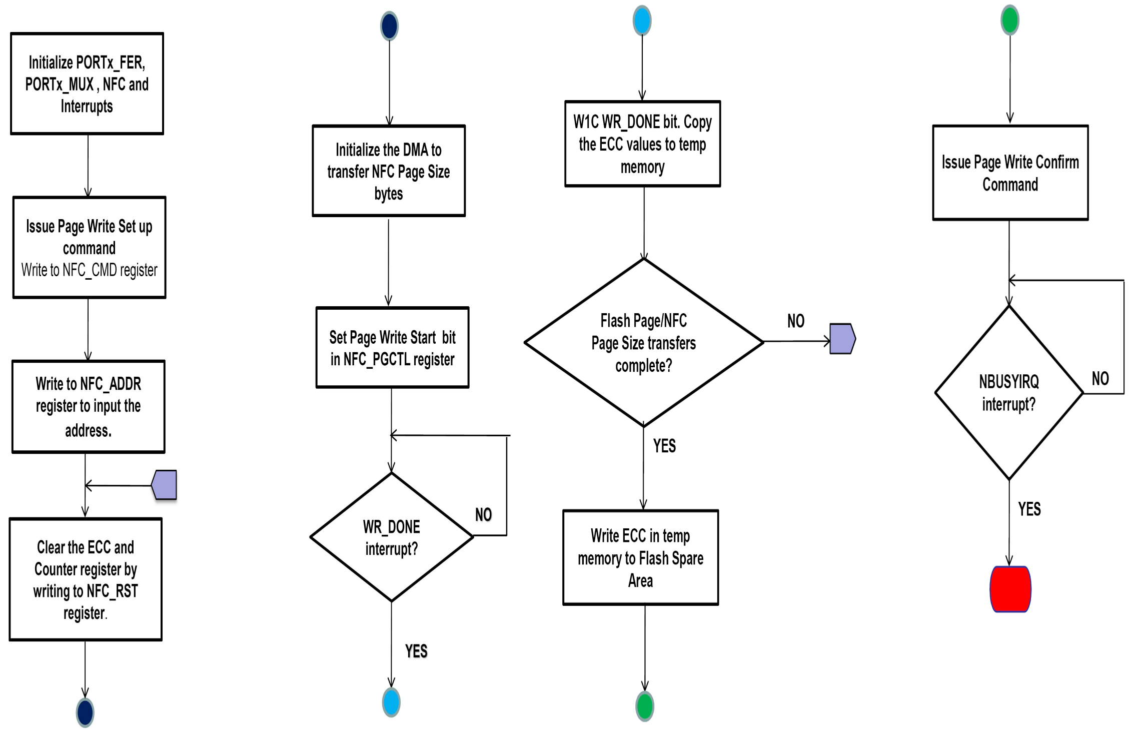

Initialize PORTx_FER, PORTx_MUX , NFC and Interrupts
Issue Page Write Set up command Write to NFC_CMD register
Write to NFC_ADDR register to input the address.
Clear the ECC and Counter register by writing to NFC_RST register.
Initialize the DMA to transfer NFC Page Size bytes
Set Page Write Start bit in NFC_PGCTL register
WR_DONE interrupt?
NO
YES
W1C WR_DONE bit. Copy the ECC values to temp memory
Flash Page/NFC Page Size transfers complete?
NO
YES
Write ECC in temp memory to Flash Spare Area
Issue Page Write Confirm Command
NBUSYIRQ interrupt?
NO
YES

Figure 3. Flow chart of page write operation

```
PageWrite(PageNo, BlockNo, &WriteData);   // &WriteData is the address in internal
                                          // memory where data to be written to
                                          // flash is stored.
```

Listing 2. API to perform page write operation

Random Data Input

During a sequential input operation, the next sequential address to be programmed can be replaced by a random address, by issuing a random data input command. This way, it allows writing to a particular location in a page. The data to be written into NAND flash is stored in the `NFC_DATA_WR` register. Figure 4 shows the flow chart of the random data input operation. The code associated with this EE-Note provides an API to perform this operation.

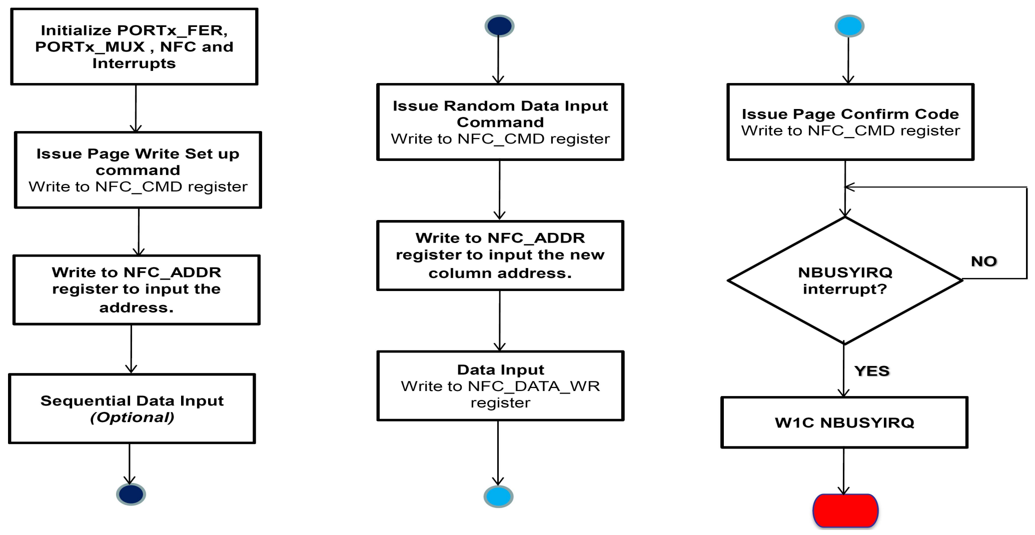

Initialize PORTx_FER, PORTx_MUX , NFC and Interrupts
Issue Page Write Set up command Write to NFC_CMD register
Write to NFC_ADDR register to input the address.
Sequential Data Input (Optional)
Issue Random Data Input Command Write to NFC_CMD register
Write to NFC_ADDR register to input the new column address.
Data Input Write to NFC_DATA_WR register
Issue Page Confirm Code Write to NFC_CMD register
NBUSYIRQ interrupt?
NO
YES
W1C NBUSYIRQ

Figure 4. Flow chart of random data input operation

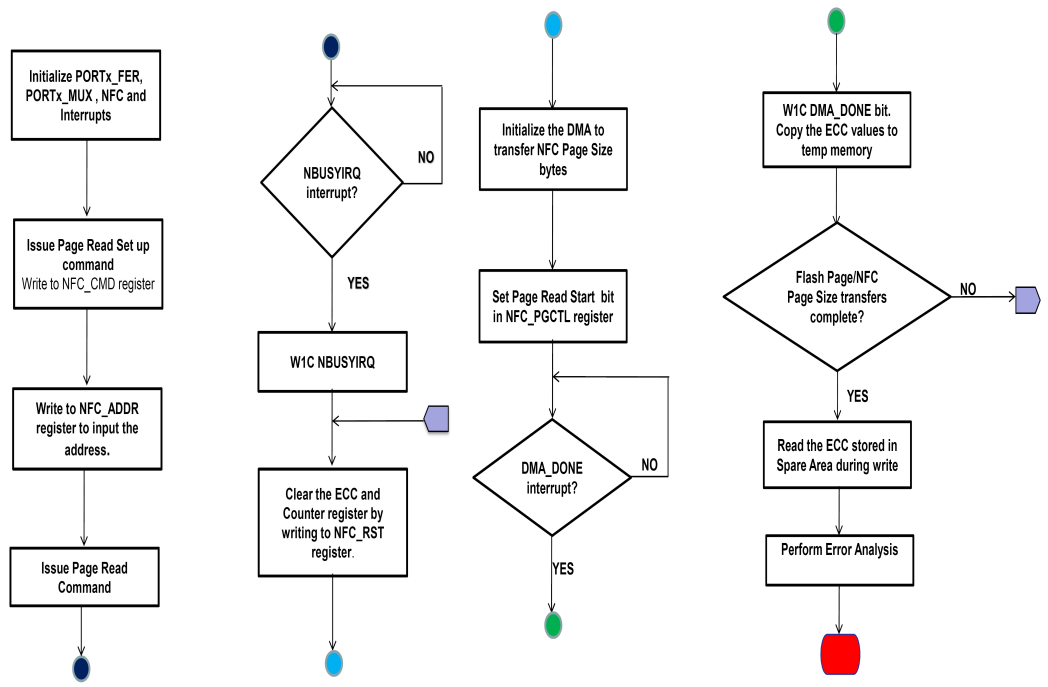
```
RandomDataInput(ByteNo, PageNo, BlockNo, DataInput);  //DataInput is the data to be
                                                      //stored in the Byte of a page
```

Listing 3. API to perform random data input

Page Read

For page read operations, first the read set-up command is issued by writing to the `NFC_CMD` register, which is followed by the address cycle. Then the read command code is issued, which asserts the `ND_RB/` signal. When a rising edge on `ND_RB/` is detected, it indicates that the data requested from a particular page is available. To initiate the DMA transfers for a page read operation, the page read start bit in the `NFC_PGCTL` register must be set. The ECC values are calculated for every 256- or 512-byte page. When the page read is complete, the core must read the ECC stored in the spare are during page write operation. The ECC stored during write operation is compared to the new ECC generated during read operation to determine the error.

Figure 5 shows the flow chart for a page read operation. The example code associated with this EE-Note demonstrates a page read operation to a flash device of that has a 2048-byte page size.



Initialize PORTx_FER, PORTx_MUX , NFC and Interrupts
Issue Page Read Set up command Write to NFC_CMD register
Write to NFC_ADDR register to input the address.
Issue Page Read Command
NBUSYIRQ interrupt?
NO
YES
W1C NBUSYIRQ
Clear the ECC and Counter register by writing to NFC_RST register.
Initialize the DMA to transfer NFC Page Size bytes
Set Page Read Start bit in NFC_PGCTL register
DMA_DONE interrupt?
NO
YES
W1C DMA_DONE bit. Copy the ECC values to temp memory
Flash Page/NFC Page Size transfers complete?
NO
YES
Read the ECC stored in Spare Area during write
Perform Error Analysis

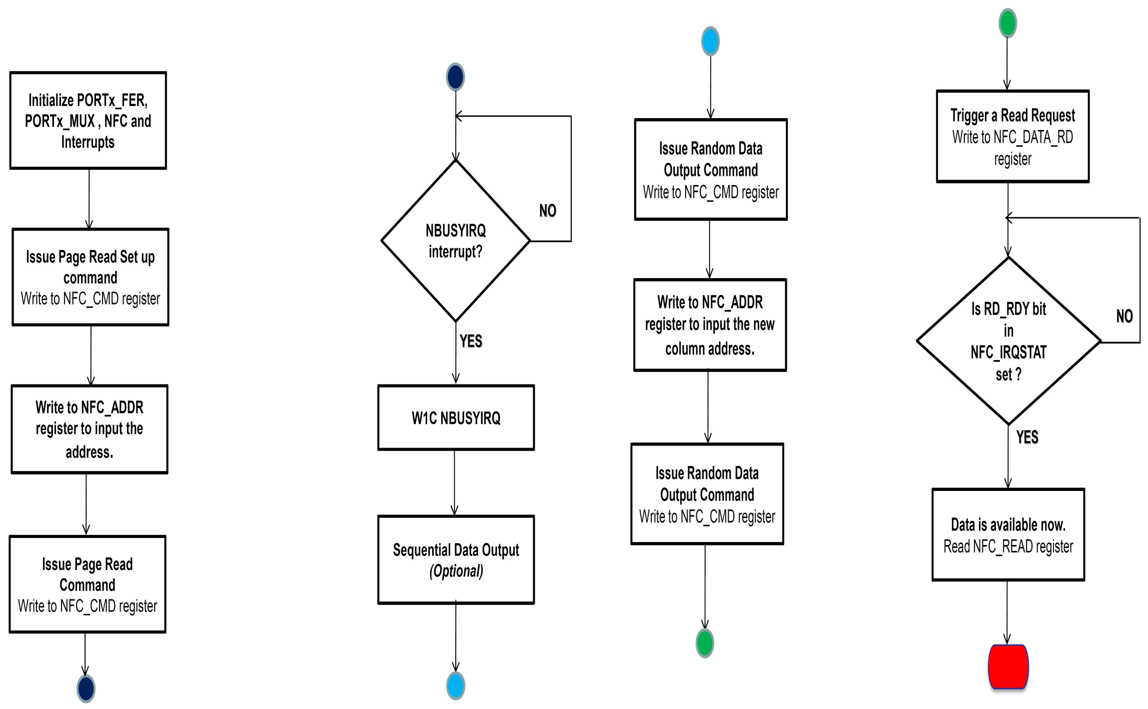
Figure 5. Flow chart of page read operation

```
PageRead(PageNo, BlockNo, &ReadData);     // &ReadData is the address where data to
                                          // read from flash to be stored in
                                          // internal memory.
```

Listing 4. API to perform a page read operation

Random Data Output

The random data output command can be used to read data in any location of a page. After issuing the sequential read command, the column address should be changed and the random data output command output must be issued. A read request is triggered by writing to the `NFC_DATA_RD` register. The `RD_RDY` bit in the `NFC_IRQSTAT` register indicates that the data is available in the `NFC_READ` register, which can be read to internal memory. Figure 6 shows the flow chart for a random data output operation. The code associated with this EE-Note provides an API to perform this operation.



Initialize PORTx_FER, PORTx_MUX , NFC and Interrupts
Issue Page Read Set up command Write to NFC_CMD register
Write to NFC_ADDR register to input the address.
Issue Page Read Command Write to NFC_CMD register
NBUSYIRQ interrupt?
NO
YES
W1C NBUSYIRQ
Sequential Data Output (Optional)
Issue Random Data Output Command Write to NFC_CMD register
Write to NFC_ADDR register to input the new column address.
Issue Random Data Output Command Write to NFC_CMD register
Trigger a Read Request Write to NFC_DATA_RD register
Is RD_RDY bit in NFC_IRQSTAT set ?
NO
YES
Data is available now. Read NFC_READ register

Figure 6. Flow chart of random data output operation

```
RandomDataOutput(PageNo, BlockNo, DataOutput);   //DataOutput is the data read
                                                //from a location in a page
```

Listing 5. API to perform a random data output operation

Error Analysis

NFC employs ECC hardware that uses the Hamming code algorithm. It generates two sets of parity bits for every 256-byte page, which is available in the `NFC_ECC0` and `NFC_ECC1` registers. A 512-byte page is split as two halves, and ECC is calculated for each half. The ECC for first half is stored in the `NFC_ECC0` and `NFC_ECC1` registers, and the ECC for the next half is stored in the `NFC_ECC2` and `NFC_ECC3` registers.

Once the page read operation is done, the software must compare the ECC stored during write operation to the new ECC generated during the read operation to determine the error. This is determined by generating the syndromes which is the XOR operation of the ECC values stored during write ($ECC0_{calculated}$ and $ECC1_{calculated}$) to the ECC generated during the read ($ECC0_{stored}$ and $ECC1_{stored}$) operation. The following equations are used to calculate the syndromes. Software should analyze these syndromes and should take necessary actions.

$$Syndrome0[21:0] = ((ECC1_{calculated}, ECC0_{calculated}) \wedge (ECC1_{stored}, ECC0_{stored}));$$
$$Syndrome1[10:0] = (ECC0_{calculated}) \wedge ECC0_{stored});$$
$$Syndrome2[10:0] = (ECC0_{calculated} \wedge ECC1_{calculated});$$
$$Syndrome3[10:0] = (ECC0_{stored} \wedge ECC1_{stored});$$
$$Syndrome4[10:0] = Syndrome2 \wedge Syndrome3;$$

Listing 6. Syndrome calculation

If `Syndrome0` is equal to zero, the data is valid.

If `Syndrome0` has 11 bits set to ‘1’ and `Syndrome4` is equal to 0x7FF, there is a one-bit correctable error. `Syndrome1` gives the failing bit within the 256-byte block. `Syndrome1[0-2]` indicates failing bit, and `Syndrome1[3-10]` indicates a failing byte, which has to be inverted.

If `Syndrome0` has one bit that is one, there is an error in the ECC data itself.

If `Syndrome0` has any other value, there is a multiple-bit, unrecoverable error. Software should mark the block containing this page as a bad block.

The sample code associated with this EE-Note performs the error analysis when a page read operation is performed.

Guidelines to Re-Use the Drivers

This section gives the guidelines to re-use the software drivers for other SLC NAND flash devices. As an example, the MT29F4G08AAAWP device from Micron® is considered here and the modified drivers for this flash device is also available with this EE-Note.

- The flash commands are to be included appropriately in the `main.h` header file.
- Flash information about number of blocks and page size must be changed in the `Init.h` header file.
- The number of address cycles to be issued must be changed in the `BlockErase.c`, `PageWrite.c`, and `PageRead.c` files; this depends on the flash density. For example, MT29F4G08AAAWP is a 4G-byte NAND flash and requires 5 address cycles to be issued for page write and page read operations. The flash device data sheet provides the details about addressing information.
- Refer to the flash device data sheet to identify the bad blocks marked by the manufacturer. For example, on flash devices from Micron, the 1st byte in the spare area of the 1st and 2nd page of every block must be read to determine the invalid blocks marked by the manufacturer.

Appendix A

Table 1 lists differences between the NFC on ADSP-BF52x processors and ADSP-BF54x processors.

Parameters	**ADSP-BF52x Processors**	**ADSP-BF54x Processors**
Data width	8 bits. `D0-D7`	8/16 bits: `D0-D15`
NFC signals	Separate signals for write enable (`ND_WR/`) and read enable (`ND_RE/`).	NFC is shared with AMC and ATAPI. Uses `AWE/` and `ARE/` signals for write and read enable, respectively.
DAB bus	16 bits: 2 access for 8-bit flash.	32 bits: 4 access for 8-bit flash. 2 access for 16-bit flash.
DMA channel	When NFC is functionality is selected, the shared DMA channel will be available for NFC automatically.	NFC and SDH share a PMAP assignment on DMA Controller 1. It is enabled in the `DMAC1_PERIMUX` register.

Table 1. Differences in NFC on ADSP-BF52x and ADSP-BF54x processors

References

[1] *NAND01G-B2G NAND02G-B2C Data Sheet, Rev 3*, November 2006, ST Microelectronics

[2] *NAND512-B, NAND01G-B NAND02G-B NAND04G-B NAND08G-B Data Sheet*, February 2005, ST Microelectronics

[3] AN1820 Application Note *How to Use the FTL and HAL Software Modules to Manage Data in Single Level Cell NAND Flash Memories*, October 2004, ST Microelectronics

[4] AN1822 Application Note *Wear Leveling in Single Level Cell NAND Flash Memories*, November 2004, ST Microelectronics

[5] *Algorithms and Data Structures for Flash Memories- Survey of various Flash File Systems by Eran Gal and Sivan Toledo, Tel-Aviv University* August 2005

[6] *http://en.wikipedia.org/wiki/Flash_memory*

[7] *MT29F4G08AAA, MT29F8G08BAA, MT29F8G08DAA, MT29F16G08FAA Data Sheet, Rev B* February 2007, Micron® Technology, Inc

[8] *ADSP-BF52x Blackfin Processor Hardware Reference (Volume 2 of 2).* Rev 0.3, September 2007. Analog Devices, Inc.

[9] *ADSP-BF54x Blackfin Processor Hardware Reference (Volume 1 of 2) Preliminary*. Rev 0.4, August 2008. Analog Devices, Inc.

[10] *ADSP-BF54x Blackfin Processor Hardware Reference (Volume 2 of 2) Preliminary*. Rev 0.4, August 2008. Analog Devices, Inc.

[11] *ADSP-BF522/523/524/525/526/527 Blackfin Embedded Processor Preliminary Data Sheet*, Rev. PrE, August 2008. Analog Devices, Inc.

[12] *ADSP-BF542/BF544/BF547/BF548/BF549 Blackfin Embedded Processor Preliminary Data Sheet,* Rev. PrH, August 2008. Analog Devices, Inc.

Document History

Revision	**Description**
Rev 1 – September 15, 2008 by Gurudath Vasanth	Initial release.