ANALOG DEVICES

NAND FLASH CONTROLLER (ADI_NFC) DEVICE DRIVER

DATE: JULY 29, 2008.

Table of Contents

List of Tables

Acronyms

ADI	Analog Devices Inc.
ADSP – BF	Analog Devices Digital Signal Processor – Blackfin
DMA	Direct Memory Access
ECC	Error Correcting/Correction Code
NFC	NAND Flash Controller
NFD	NAND Flash Device (memory)
MLC	Multi-Level-Cell NAND devices
SFTL	Simple Flash Translation Layer
SLC	Single-Level-Cell NAND devices

Document Revision History

Date	Description of Changes
Jan 22, 2008	Initial release
July 29, 2008	Added `ADI_NFC_CMD_SET_ECC_MODE`, `ADI_NFC_CMD_GET_NFD_INFO`, `ADI_NFC_CMD_GET_ELEMENT_SIZE` commands

Table 1 – Revision History

1. Overview

This document describes functionality of NAND Flash Controller (NFC) driver for ADSP – BF52x and ADSP – BF54x NAND Flash Controller. The driver adheres to Analog Devices' Device Driver and System Services Model.

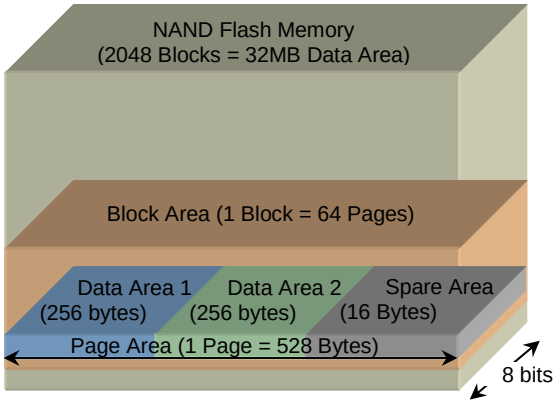
1.1. NAND Flash Memory Overview

NAND flash is a non volatile solid state memory that provides the lowest semiconductor cost per MB making NAND flash the logical choice for the demands of growing storage applications, especially in the hand held devices.

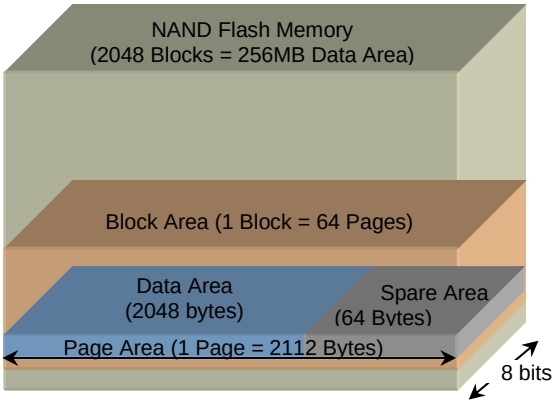
NAND flash memory consists of a large array of transistors connected serially (similar to that of a NAND gate) that are used for data storage. NAND Flash is more of a sequential access device unlike NOR which is designed to be a random access memory. Sequential access results in faster programming and erase times than NOR flash and makes NAND memory ideal for large data storage applications and applications where files are frequently overwritten.

There are two different types of NAND flash devices in the market today: Single-Level-Cell (SLC) devices and Multi-Level-Cell (MLC) devices. MLC chip technology is capable of storing two or more bits of data per memory cell, while SLC chip technology allows just one bit of data per memory cell. MLC chips provide much larger storage capacity than SLC; however, they also suffer from lower bandwidth for read and write operations. Also, the Error Correcting Code (ECC) requirement for MLC is higher than for SLC, as MLC technology is more prone to bit errors. The NAND Flash Controller (NFC) on ADSP-BF52x and ADSP-BF54x processors only support SLC devices.

Similar to hard disks or memory cards, NAND Flash memory is accessed like block device using column, page, and block addressing. A NAND page size is typically of 512 bytes for small page device or 2,048 bytes for large page device. The pages are typically arranged in blocks. A typical block would be 32 pages of 512 bytes or 64 pages of 2,048 bytes. Each NAND page consists of few spare bytes or spare area, typically 12 to 16 bytes, that are mostly used for storage of error detection and correction checksum. The read and program operations execute at the page level, and erase operations are performed at the block level.



Small Page SLC NAND Flash Memory



Large Page SLC NAND Flash Memory

1.2. NAND Flash Controller Driver Overview

The NFC driver supports both ADSP – BF52x and ADSP – BF54x processors.

A NAND Flash Device (memory) contains N number of 'Blocks', where each NFD 'Block' contains M number of 'Pages' and each NFD 'Page' can be split in to Page Data area and Page Spare area. Typically, the Page Data Area is used to store the actual user data and Spare area is used for error correction and other memory management tasks.

The NFC driver treats NFD Page Data area as 256-byte packets (called NFD Data packet).

In sector access mode, the NFC driver treats each sector as 512-byte data packet.

NFC driver features

- Supports x8 and x16 type Large page devices (x16 support is yet to be tested)
- Page/Column/Block address based memory access
- Sector based memory access using physical sector number
- Supports accessing NAND Memory directly by writing to NFC registers (no DMA used)
- Supports NFC DMA based NAND Memory access
- 1-bit Error Correction (supported only in DMA based NFC access)
- Stores hardware generated ECC to NAND memory page - spare area for error correction (supported only in DMA based NFC access)
- Can be configured to treat NFD Spare Area as a Sequential/Non-Sequential memory.
- Enable/Disable Hardware generated Error correction

NFC driver performance

Processor	DMA Frame Memory	Read Access	Write Access
ADSP – BF527 CCLK = 400 MHz SCLK = 133 MHz	No	31.07 Mbits/s (3.88 MB/s)	7.43 Mbits/s (0.93 MB/s)
	Yes	63.42 Mbits/s (7.93 MB/s)	23.84 Mbits/s (2.98 MB/s)
ADSP – BF548 CCLK = 400 MHz SCLK = 109 MHz	No	35.65 Mbits/s (4.45 MB/s)	7.82 Mbits/s (0.98 MB/s)
	Yes	70.36 Mbits/s (8.79 MB/s)	24.77 Mbits/s (3.1 MB/s)

Table 2 – NFC Driver performance results

2. Files

The files listed below comprise the device driver API and source files.

2.1. Include Files

The driver sources include the following include files:

- **<services/services.h>**
  This file contains all definitions, function prototypes etc. for all the System Services.

- **<drivers/adi_dev.h>**
  This file contains all definitions, function prototypes etc. for the Device Manager and general device driver information.

- **<drivers/nfc/adi_nfc.h>**
  This file contains all definitions, function prototypes etc. specific to NFC driver.

2.2. Source Files

The driver sources are contained in the following files, as located in the default installation directory:

- **<Blackfin/lib/src/drivers/nfc/adi_nfc.c>**
  This file contains all the source code for the NFC Device Driver. All source code is written in ‘C’. There are no assembly level functions in this driver.

- **<Blackfin/lib/src/drivers/nfc/adi_nfc_reg.h>**
  This file contains definitions of all MMR addresses and the bit-field access macros specific to NFC.

3. Lower Level Drivers

The NFC driver does not use any lower level device drivers.

4. Resources Required

Device drivers typically consume some amount of system resources. This section describes the resources required by the device driver.

Unless explicitly noted in the sections below, this device driver uses the System Services to access and control any required hardware. The information in this section may be helpful in determining the resources this driver requires, such as the number of interrupt handlers or number of DMA channels etc., from the System Services.

Because dynamic memory allocations are not used in the Device Drivers or System Services, all memory used by the Device Drivers and System Services must be supplied by the application. The Device Drivers and System Services supply macros that can be used by the application to size the amount of base memory and/or the amount of incremental memory required to support the needed functionality. Memory for the Device Manager and System Services is provided in the initialization functions (adi_xxx_Init()).

Wherever possible, this device driver uses the System Services to perform the necessary low-level hardware access and control.

Each NFC driver instance requires an additional (driver) memory of size **ADI_DEV_DEVICE_MEMORY**

4.1. Interrupts

NFC driver requires three additional memory of size **ADI_INT_SECONDARY_MEMORY** to handle its DMA interrupts (DMA data interrupt and DMA error interrupt), and NFC peripheral interrupt.

4.2. DMA

NFC device requires additional DMA memory of size **ADI_DMA_CHANNEL_MEMORY** to handle its DMA channel.

4.3. Semaphores

NFC driver requires memory for one semaphore of size **ADI_SEM_SEMAPHORE_MEMORY** to manage NFD data transfers.

4.4. Timers

Timer service is not used by this driver.

4.5. Real-Time Clock

RTC service is not used by this driver

4.6. Programmable Flags

No Programmable Flags are used by this driver

4.7. Pins

Blackfin NFC port pins connected to NAND Flash Memory.
Refer to corresponding device reference manuals for further information.

5. Supported Features of the Device Driver

This section describes what features are supported by the device driver.

5.1. Directionality

The driver supports the dataflow directions listed in the table below.

ADI_DEV_DIRECTION	Description
ADI_DEV_DIRECTION_INBOUND	Supports the reception of data in through the device.
ADI_DEV_DIRECTION_OUTBOUND	Supports the transmission of data out through the device.
ADI_DEV_DIRECTION_BIDIRECTIONAL	Supports both the reception of data and transmission of data through the device.

Table 3 – Supported Dataflow Directions

5.2. Dataflow Methods

The driver supports the dataflow methods listed in the table below.

ADI_DEV_MODE	Description
ADI_DEV_MODE_CHAINED	Supports the chained buffer method

Table 4 – Supported Dataflow Methods

5.3. Buffer Types

The driver supports the buffer types listed in the table below.

- **ADI_DEV_1D_BUFFER**
  - Linear one-dimensional buffer
  - pAdditionalInfo – ignored

5.4. Command IDs

This section enumerates the commands that are supported by the driver. The commands are divided into three sections. The first section describes commands that are supported directly by the Device Manager. The next section describes common commands that the driver supports. The remaining section describes driver specific commands.

Commands are sent to the device driver via the adi_dev_Control() function. The adi_dev_Control() function accepts three arguments:

- DeviceHandle – This parameter is a **ADI_DEV_DEVICE_HANDLE** type that uniquely identifies the device driver. This handle is provided to the client in the adi_dev_Open() function call.
- CommandID – This parameter is a u32 data type that specifies the command ID.
- Value – This parameter is a void * whose value is context sensitive to the specific command ID.

The sections below enumerate the command IDs that are supported by the driver and the meaning of the Value parameter for each command ID.

5.4.1. Device Manager Commands

The commands listed below are supported and processed directly by the Device Manager. As such, all device drivers support these commands.

- **ADI_DEV_CMD_TABLE**
  - Table of command pairs being passed to the driver
  - Value – ADI_DEV_CMD_VALUE_PAIR *
- **ADI_DEV_CMD_END**
  - Signifies the end of a command pair table
  - Value – ignored
- **ADI_DEV_CMD_PAIR**
  - Single command pair being passed
  - Value – ADI_DEV_CMD_PAIR *
- **ADI_DEV_CMD_SET_SYNCHRONOUS**
  - Enables/disables synchronous mode for the driver
  - Value – TRUE/FALSE

5.4.2. Common Commands

The command IDs described in this section are common to many device drivers. The list below enumerates all common command IDs that are supported by this device driver.

- **ADI_DEV_GET_PERIPHERAL_DMA_SUPPORT**
  - Determines if the device driver is supported by peripheral DMA
  - Value – u32 * (location where TRUE or FALSE is stored)
- **ADI_DEV_CMD_SET_DATAFLOW_METHOD**
  - Specifies the dataflow method the device is to use. The list of dataflow types supported by the device driver is specified in section 5.2.
  - Value – **ADI_DEV_MODE** enumeration
- **ADI_DEV_CMD_SET_STREAMING**
  - Enables/disables the streaming mode of the driver.
  - Value – TRUE/FALSE
- **ADI_DEV_CMD_GET_INBOUND_DMA_CHANNEL_ID**
  - Returns the DMA channel ID value for the device driver's inbound DMA channel
  - Value – u32 * (location where the channel ID is stored)
- **ADI_DEV_CMD_GET_OUTBOUND_DMA_CHANNEL_ID**
  - Returns the DMA channel ID value for the device driver's outbound DMA channel
  - Value – u32 * (location where the channel ID is stored)
- **ADI_DEV_CMD_SET_INBOUND_DMA_CHANNEL_ID**
  - Sets the DMA channel ID value for the device driver's inbound DMA channel
  - Value – ADI_DMA_CHANNEL_ID (DMA channel ID)
- **ADI_DEV_CMD_SET_OUTBOUND_DMA_CHANNEL_ID**
  - Sets the DMA channel ID value for the device driver's outbound DMA channel
  - Value – ADI_DMA_CHANNEL_ID (DMA channel ID)
- **ADI_DEV_CMD_GET_INBOUND_DMA_PMAP_ID**
  - Returns the PMAP ID for the device driver's inbound DMA channel
  - Value – u32 * (location where the PMAP value is stored)
- **ADI_DEV_CMD_GET_OUTBOUND_DMA_PMAP_ID**
  - Returns the PMAP ID for the device driver's outbound DMA channel
  - Value – u32 * (location where the PMAP value is stored)
- **ADI_DEV_CMD_SET_DATAFLOW**
  - Enables/disables dataflow through the device
  - Value – TRUE/FALSE

- **ADI_DEV_CMD_SET_ERROR_REPORTING**
  - Enables/Disables error reporting from the device driver
  - Value – TRUE/FALSE
- **ADI_DEV_CMD_FREQUENCY_CHANGE_PROLOG**
  - Notifies device driver immediately prior to a CCLK/SCLK frequency change
  - Value – **ADI_DEV_FREQUENCIES *** (new frequencies)
- **ADI_DEV_CMD_FREQUENCY_CHANGE_EPILOG**
  - Notifies device driver immediately following a CCLK/SCLK frequency change
  - Value – **ADI_DEV_FREQUENCIES *** (new frequencies)

5.4.3. Device Driver Specific Commands

The command IDs listed below are supported and processed by the device driver. These command IDs are unique to this device driver.

- **ADI_NFC_CMD_PASS_NFD_INFO**
  - Pass NAND Flash Device specific information
  - Value – **ADI_NFD_INFO_TABLE *** (address of NFD Information table).
  - Default = NULL.
  - Note: This Command is MANDATORY and MUST be issued soon after opening the driver. NFC driver assumes that the given information table is not volatile and defined as global.
  - Refer: Section 5.8.1 for list of fields in `ADI_NFD_INFO_TABLE` structure.

- **ADI_NFC_CMD_SET_256BYTES_ALIGN_BUFFER**
  - Pass address of 256 bytes buffer for error correction (accessing NFD page spare area) and to align NFD Data to NFD Data Packet sized boundary (256 bytes)
  - Value – u8* (start address of a 256 bytes array)
  - Default = NULL
  - Note: This Command is MANDATORY and MUST be issued before submitting a read / write buffer to NFC driver. NFC driver assumes that the given buffer is not volatile and defined as global.

- **ADI_NFC_CMD_SET_DMA_FRAME_BUFFER**
  - Pass address of DMA Frame buffer that can be used to build descriptor chains to access a NFD page. It is not mandatory to provide DMA Frame buffer, but doing so will improve the NFC driver performance.
  - Value – **ADI_NFC_DMA_FRAME_BUFFER *** (address of a DMA Frame buffer structure)
  - Default = NULL (Driver will use 1D buffer reserved area to build DMA descriptor). NFC driver assumes that the given buffer is not volatile and defined as global.
  - Refer: Section 5.8.4 for list of fields in `ADI_NFC_DMA_FRAME_BUFFER` structure

- **ADI_NFC_CMD_SET_ECC_MODE**
  - Set ECC storage mode
  - Value – ADI_NFC_ECC_MODE
  - Default = ADI_NFC_ECC_MODE_DISABLE
  - Note: NFC driver treats the whole NFD page area as user data area when the error correction is disabled.
  - Refer: Section 5.7.2 for list of ECC modes supported by the driver.

- **ADI_NFC_CMD_SET_ECC_START_LOC**
  - Set Nth byte/word of Data packet spare area that holds/to hold 1st byte/word of ECC. Rest of the ECC bytes/words will be stored in N+1 and N+2 locations of NFD spare area.
  - Value – u16
  - Default = 0 (ECC read from/written to NFD spare area starting from byte/word 0)

- **ADI_NFC_CMD_SEND_NFD_ACCESS_REQUEST**
  - Send a NFD access request using Block/Page/Column address. NFC Driver uses DMA to service this request.
  - Value – **ADI_NFD_ADDRESS ***
  - Default = NULL
  - Refer: Section 5.8.2 for list of fields in `ADI_NFD_ADDRESS` structure

- **ADI_NFC_CMD_SEND_SECTOR_ACCESS_REQUEST**
  - Send a NFD access request using physical sector number. NFC Driver uses DMA to service this request.
  - Value – u32 (start sector number to access)

- **ADI_NFC_CMD_ERASE_NFD_BLOCK**
  - Erase a NFD Block
  - Value – u16 (NFD Block number to erase)

- **ADI_NFC_CMD_READ_NFD_DIRECT**
  - Reads # bytes (for x8) or # words (for x16) from NFD starting from the given Block/Page/Column address. The driver does not use DMA to service this request; in turn it uses NFC registers to access NFD memory.
  - Value – **ADI_NFD_DIRECT_ACCESS ***
  - Refer: Section 5.8.3 for list of fields in `ADI_NFD_DIRECT_ACCESS` structure

- **ADI_NFC_CMD_WRITE_NFD_DIRECT**
  - Writes # bytes (for x8) or # words (for x16) to NFD starting from the given Block/Page/Column address. The driver does not use DMA to service this request; in turn it uses NFC registers to access NFD memory.
  - Value – **ADI_NFD_DIRECT_ACCESS ***
  - Refer: Section 5.8.3 for list of fields in `ADI_NFD_DIRECT_ACCESS` structure

- **ADI_NFC_CMD_TERMINATE_NFD_ACCESS**
  - Terminate all NFC/NFD related access requests that are in progress.
  - Value – NULL

- **ADI_NFC_CMD_GET_NFD_STATUS**
  - Get present value of NFD status register.
  - Value – u8* (location to store status register value read from NFD)

- **ADI_NFC_CMD_GET_NFD_ELECTRONIC_SIGNATURE**
  - Get NFD Electronic signature.
  - Value – u32* (location to store Electronic signature read from NFD)

- **ADI_NFC_CMD_GET_NFD_INFO**
  - Gets address of a structure holding NAND Flash Device specific information
  - Value – **ADI_NFD_INFO_TABLE *** (location to store address of NFD Information table)
  - Refer: Section 5.8.1 for list of fields in `ADI_NFD_INFO_TABLE` structure.

- **ADI_NFC_CMD_GET_ELEMENT_SIZE**
  - Get NFD Data element Size.
  - Value – u32* (location to store size of data element)

5.5. Callback Events

This section enumerates the callback events the device driver is capable of generating. The events are divided into two sections. The first section describes events that are common to many device drivers. The next section describes driver specific event IDs. The client should prepare its callback function to process each event described in these two sections.

The callback function is of the type **`ADI_DCB_CALLBACK_FN`**. The callback function is passed three parameters. These parameters are:

- ClientHandle – This void * parameter is the value that is passed to the device driver as a parameter in the adi_dev_Open() function.
- EventID – This is a u32 data type that specifies the event ID.
- Value – This parameter is a void * whose value is context sensitive to the specific event ID.

The sections below enumerate the event IDs that the device driver can generate and the meaning of the Value parameter for each event ID.

5.5.1. Common Events

The events described in this section are common to many device drivers. The list below enumerates all common event IDs that are supported by this device driver.

- **`ADI_DEV_EVENT_BUFFER_PROCESSED`**
  - Notifies callback function that a chained buffer has been processed by the device driver.
  - Value – For chained dataflow method, this value is the CallbackParameter value that was supplied in the buffer that was passed to the adi_dev_Read() or adi_dev_Write() function.
- **`ADI_DEV_EVENT_DMA_ERROR_INTERRUPT`**
  - Notifies the callback function that a DMA error occurred.
  - Value – Null.

5.5.2. Device Driver Specific Events

The events listed below are supported and processed by the device driver. These event IDs are unique to this device driver.

This driver doesn't have any unique events.

5.6. Return Codes

All API functions of the device driver return status indicating either successful completion of the function or an indication that an error has occurred. This section enumerates the return codes that the device driver is capable of returning to the client. A return value of **ADI_DEV_RESULT_SUCCESS** indicates success, while any other value indicates an error or some other informative result. The value **ADI_DEV_RESULT_SUCCESS** is always equal to the value zero. All other return codes are a non-zero value.

The return codes are divided into two sections. The first section describes return codes that are common to many device drivers. The next section describes driver specific return codes. The client should prepare to process each of the return codes described in these sections.

Typically, the application should check the return code for **ADI_DEV_RESULT_SUCCESS**, taking appropriate corrective action if **ADI_DEV_RESULT_SUCCESS** is not returned. For example:

```
if (adi_dev_Xxxx(…) == ADI_DEV_RESULT_SUCCESS) {
      // normal processing
} else {
      // error processing
}
```

5.6.1. Common Return Codes

The return codes described in this section are common to many device drivers. The list below enumerates all common return codes that are supported by this device driver.

- **ADI_DEV_RESULT_SUCCESS**
  - The function executed successfully.
- **ADI_DEV_RESULT_NOT_SUPPORTED**
  - The function is not supported by the driver.
- **ADI_DEV_RESULT_DEVICE_IN_USE**
  - The requested device is already in use.
- **ADI_DEV_RESULT_NO_MEMORY**
  - There is insufficient memory available.
- **ADI_DEV_RESULT_BAD_DEVICE_NUMBER**
  - The device number is invalid.
- **ADI_DEV_RESULT_DIRECTION_NOT_SUPPORTED**
  - The device cannot be opened in the direction specified.
- **ADI_DEV_RESULT_BAD_DEVICE_HANDLE**
  - The handle to the device driver is invalid.
- **ADI_DEV_RESULT_BAD_MANAGER_HANDLE**
  - The handle to the Device Manager is invalid.
- **ADI_DEV_RESULT_BAD_PDD_HANDLE**
  - The handle to the physical driver is invalid.
- **ADI_DEV_RESULT_INVALID_SEQUENCE**
  - The action requested is not within a valid sequence.
- **ADI_DEV_RESULT_ATTEMPTED_READ_ON_OUTBOUND_DEVICE**
  - The client attempted to provide an inbound buffer for a device opened for outbound traffic only.
- **ADI_DEV_RESULT_ATTEMPTED_WRITE_ON_INBOUND_DEVICE**
  - The client attempted to provide an outbound buffer for a device opened for inbound traffic only.
- **ADI_DEV_RESULT_DATAFLOW_UNDEFINED**
  - The dataflow method has not yet been declared.
- **ADI_DEV_RESULT_DATAFLOW_INCOMPATIBLE**
  - The dataflow method is incompatible with the action requested.

- **ADI_DEV_RESULT_BUFFER_TYPE_INCOMPATIBLE**
  - The device does not support the buffer type provided.
- **ADI_DEV_RESULT_CANT_HOOK_INTERRUPT**
  - The Interrupt Manager failed to hook an interrupt handler.
- **ADI_DEV_RESULT_CANT_UNHOOK_INTERRUPT**
  - The Interrupt Manager failed to unhook an interrupt handler.
- **ADI_DEV_RESULT_NON_TERMINATED_LIST**
  - The chain of buffers provided is not NULL terminated.
- **ADI_DEV_RESULT_NO_CALLBACK_FUNCTION_SUPPLIED**
  - No callback function was supplied when it was required.
- **ADI_DEV_RESULT_REQUIRES_UNIDIRECTIONAL_DEVICE**
  - Requires the device be opened for either inbound or outbound traffic only.
- **ADI_DEV_RESULT_REQUIRES_BIDIRECTIONAL_DEVICE**
  - Requires the device be opened for bidirectional traffic only.

5.6.2. Device Driver Specific Return Codes

The return codes listed below are supported and processed by the device driver. These event IDs are unique to this device driver.

- **ADI_NFC_RESULT_CMD_NOT_SUPPORTED** (Hex: 0x402E0001)
  - Command supplied by the client is not supported by NFC device driver
- **ADI_NFC_RESULT_NFD_UNDEFINED** (Hex: 0x402E0002)
  - Given NAND Flash Device type is undefined.
- **ADI_NFC_RESULT_NFD_NOT_SUPPORTED** (Hex: 0x402E0003)
  - Driver does not support the NAND Flash Device type supplied by the client.
- **ADI_NFC_RESULT_NFD_ACCESS_IN_PROGRESS** (Hex: 0x402E0004)
  - Results when client issues as NFD access request before the previous request gets serviced.
- **ADI_NFC_RESULT_NFD_ADDRESS_OUT_OF_BOUNDARY** (Hex: 0x402E0005)
  - Block/Page/Column address or Sector number issued by the client exceeds NAND Flash Device Memory size.
- **ADI_NFC_RESULT_NFD_DATA_ERROR** (Hex: 0x402E0006)
  - Detected an un-correctable error with data read from NFD.
- **ADI_NFC_RESULT_NO_ALIGNMENT_BUFFER** (Hex: 0x402E0007)
  - Results when client tries to perform a DMA based NFC access without providing a 256-bytes buffer for Spare Area Access & NFD Packet Alignment.

5.7. Enumerations

This section lists the enumerations specific to NFC device driver.

5.7.1. ADI_NFC_NFD_TYPE

Below enumeration table lists the NAND Flash Device types supported by NFC driver

```
/* NAND Flash Device Type */
typedef enum ADI_NFC_NFD_TYPE
{
      ADI_NFD_UNDEFINED,      /* NAND Flash Device type is not defined yet          */
      ADI_NFD_SMALL_PAGE_x8,  /* Small Page NAND Flash Device with 8-bit Data Bus   */
      ADI_NFD_SMALL_PAGE_x16, /* Small Page NAND Flash Device with 16-bit Data Bus  */
      ADI_NFD_LARGE_PAGE_x8,  /* Large Page NAND Flash Device with 8-bit Data Bus   */
      ADI_NFD_LARGE_PAGE_x16, /* Large Page NAND Flash Device with 16-bit Data Bus  */
} ADI_NFC_NFD_TYPE;
```

5.7.2. ADI_NFC_ECC_MODE

Below enumeration table lists the Hardware generated ECC storage modes supported by NFC driver

```
/* NFC ECC Mode */
typedef enum ADI_NFC_ECC_MODE
{

      /* ECC stored non-sequentially in NFD Spare area */
      ADI_NFC_ECC_MODE_NON_SEQUENTIAL = 0,
      /* ECC stored sequentially in NFD Spare area */
      ADI_NFC_ECC_MODE_SEQUENTIAL,
      /* Disable Error correction/Discard HW generated ECC */
      ADI_NFC_ECC_MODE_DISABLE

} ADI_NFC_ECC_MODE;
```

5.8. Data structures

This section lists the data structures specific to NFC device driver.

5.8.1. ADI_NFD_INFO_TABLE

The below structure is used to pass NAND Flash Device specific information to NFC driver

```
/* Structure to hold NAND Flash Device parameters */
typedef struct ADI_NFD_INFO_TABLE
{
    u8                  tWPmin;
    u8                  tCSmin;
    u8                  tRPmin;
    u8                  tREAmax;
    u8                  tCEAmax;
    u16                 PageSize;
    u16                 DataAreaPerPage;
    u16                 PagesPerBlock;
    u16                 TotalBlocks;
    ADI_NFC_NFD_TYPE    NFDType;
} ADI_NFD_INFO_TABLE;
```

tWPmin	tWP Minimum (Write Enable Low to Write Enable High) value mentioned in NAND Flash Device spec (in nano seconds)
tCSmin	tCS Minimum (Chip Enable Low to Write Enable High) value mentioned in NAND Flash Device spec (in nano seconds)
tRPmin	tRP Minimum (Read Enable Low to Read Enable High/Read Enable Pulse Width) value mentioned in NAND Flash Device spec (in nano seconds)
tREAmax	tREA Maximum (Read Enable Low to Output Valid/Read Enable Access time) value mentioned in NAND Flash Device spec (in nano seconds)
tCEAmax	tCEA Maximum (Read Enable Low to Output Valid/Read Enable Access time) value mentioned in NAND Flash Device spec (in nano seconds)
PageSize	Page size (in bytes for x8, in words for x16)
DataAreaPerPage	Data Area size per Page (in bytes for x8, in words for x16)
PagesPerBlock	Number of pages per block
TotalBlocks	Total blocks in this NAND Flash device(including invalid blocks)
NFDType	NAND Flash Device Type

5.8.2. ADI_NFD_ADDRESS

Below data structure is used to pass Block/Page/Column address of the NAND Flash Device memory to be accessed by NFC driver using DMA.

```
/* NFD Block/Page/Column address to Access */
typedef struct ADI_NFD_ADDRESS
{
      u16   BlockAddress;     /* NAND Memory Block address to access          */
      u16   PageAddress;      /* NAND Memory Page address to access           */
      u16   ColumnAddress;    /* NAND Memory Column address to start with     */
} ADI_NFD_ADDRESS;
```

5.8.3. ADI_NFD_DIRECT_ACCESS

Below data structure is used to issue a direct NFD access request. NFC driver does not use DMA to service a direct access request; in turn it uses NFC registers to access NFD memory. Direct access can be used to access any part of NFD Memory, where as DMA based NFC access can be used only to access NFD Data Memory area in a page.

```
/* Structure to access NFD memory directly */
typedef struct ADI_NFD_DIRECT_ACCESS
{
      u16         BlockAddress;
      u16         PageAddress;
      u16         ColumnAddress;
      void        *pData;
      u32         AccessCount;
} ADI_NFD_DIRECT_ACCESS;
```

BlockAddress	NAND Flash Device Memory Block address to access
PageAddress	NAND Flash Device Page address to access
ColumnAddress	NAND Flash Device Column address to start with
***pData**	pointer to array holding/to hold NFD data (u8 type array for x8 NFD, u16 type array for x16 NFD)
AccessCount	# bytes or # words to read from / write to NFD (in bytes for x8 type NFD, in words for x16 type NFD)

5.8.4. ADI_NFC_DMA_FRAME_BUFFER

Below data structure is used to set NFC DMA Frame buffer. It is not mandatory to provide a DMA frame buffer, but doing so will improve NFC driver performance by 3 to 5 times.

```
/* Data structure to set NFC DMA Frame buffer */
typedef struct ADI_NFC_DMA_FRAME_BUFFER
{
      void        *pDMAFrameMemory;
      u8          NumDMAFrames;
} ADI_NFC_DMA_FRAME_BUFFER;
```

***pDMAFrameMemory**	Pointer to NFC DMA Data Frame memory allocated by the application, where DMA Frame Memory size = `ADI_NFC_DMA_DATA_FRAME_BASE_MEMORY * NumDMAFrames`
NumDMAFrames	Number of NFC DMA Data Frames (must be = NFD Data Size per page / 256), where each data frame must be of size `ADI_NFC_DMA_DATA_FRAME_BASE_MEMORY`

Example:

Consider ST Microelectronics NAND02GW3B2C NAND Flash Device available on ADSP-BF548 Ez-Kit Lite. This device has 2048-bytes of data area per page.

Number of DMA Frames required for NAND02GW3B2C NFD = (2048 / 256) = 8.

DMA Frame memory required to access
one NAND02GW3B2C NFD Page = (`ADI_NFC_DMA_DATA_FRAME_BASE_MEMORY * 8`)

6. Configuring the Device Driver

This section describes the default configuration settings for the device driver and any additional configuration settings required from the client application.

6.1. Entry Point

When opening the device driver with the adi_dev_Open() function call, the client passes a parameter to the function that identifies the specific device driver that is being opened. This parameter is called the entry point. The entry point for this driver is listed below.

- ADI_NFC_EntryPoint

6.2. Default Settings

The table below describes the default configuration settings for the device driver. If the default values are inappropriate for the given system, the application should use the command IDs listed in the table to configure the device driver appropriately.

Item	Default Value	Possible Values	Command ID
ECC Start Location	0	Between 0 and 4 for x8 type Devices, Between 0 and 2 for x16 type devices	ADI_NFC_CMD_SET_ECC_START_LOC

Table 5 – Default Settings

6.3. Additional Required Configuration Settings

In addition to the possible overrides of the default driver settings, the device driver requires the application to specify the additional configuration information listed in the table below.

Item	Possible Values	Command ID
NAND Flash Device Information table	ADI_NFD_INFO_TABLE *	ADI_NFC_CMD_PASS_NFD_INFO
256 bytes spare area access / NFD packet alignment buffer	u8*	ADI_NFC_CMD_SET_256BYTES_ALIGN_BUFFER

Table 6 – Additional Required Settings

6.4. NFD Page Access Modes

NFD Access Modes	Error Correction Mode	NFD Page Area	
		Page Data Area	Page Spare Area
Direct NFD Access (no DMA)	Don't care	Treated as user data area	Treated as user data area
NFD access via NFC DMA	Enabled	Treated as user data area	Used for error correction and other memory management tasks
	Disabled	Treated as user data area	Treated as user data area

Table 7 – NFD Page Access Modes

7. Appendix

7.1. NFD Information table for ADSP – BF526 Ez–Kit

```
/* Global data: NFD Information table for ST Microelectronics NAND02GR3B2CZAB
                NAND Flash Device on ADSP-BF526 Ez-Kit Lite */

extern const ADI_NFD_INFO_TABLE  gaADI_BF527EZKIT_NFD_Info =
{
    25,     /* tWP Minimum value from NAND02GR3B2CZAB spec (25 ns for 1.8V device   */
    35,     /* tCS Minimum value from NAND02GR3B2CZAB spec (35 ns for 1.8V device)  */
    25,     /* tRP Minimum value from NAND02GR3B2CZAB spec (25 ns for 1.8V device)  */
    30,     /* tREA Maximum value from NAND02GR3B2CZAB spec (30 ns for 1.8V device) */
    45,     /* tCEA Maximum value from NAND02GR3B2CZAB spec (45 ns for 1.8V device) */
    2112,   /* Page Size (in bytes)                                                 */
    2048,   /* Data Area size per page (in bytes)                                   */
    64,     /* Pages per Block                                                      */
    2048,   /* Total Blocks in this device (including invalid blocks)               */
    ADI_NFD_LARGE_PAGE_x8,    /* NFD type                                           */
};
```

7.2. NFD Information table for ADSP – BF527 Ez–Kit

```
/* Global data: NFD Information table for ST Microelectronics NAND04GW3B2BN6E
                NAND Flash Device on ADSP-BF527 Ez-Kit Lite */

extern const ADI_NFD_INFO_TABLE  gaADI_BF527EZKIT_NFD_Info =
{
    15,     /* tWP Minimum value from NAND04GW3B2BN6E spec  (15 ns for 3V device)   */
    25,     /* tCS Minimum value from NAND04GW3B2BN6E spec  (25 ns for 3V device)   */
    15,     /* tRP Minimum value from NAND04GW3B2BN6E spec  (15 ns for 3V device)   */
    25,     /* tREA Maximum value from NAND04GW3B2BN6E spec (25 ns for 3V device)   */
    30,     /* tCEA Maximum value from NAND04GW3B2BN6E spec (30 ns for 3V device)   */
    2112,   /* Page Size (in bytes)                                                 */
    2048,   /* Data Area size per page (in bytes)                                   */
    64,     /* Pages per Block                                                      */
    4096,   /* Total Blocks in this device (including invalid blocks)               */
    ADI_NFD_LARGE_PAGE_x8,    /* NFD type                                           */
};
```

7.3. NFD Information table for ADSP – BF548 Ez–Kit

```
/* Global data: NFD Information table for ST Microelectronics NAND02GW3B2C
                NAND Flash Device on ADSP-BF548 Ez-Kit Lite */

extern const ADI_NFD_INFO_TABLE  gaADI_BF548EZKIT_NFD_Info =
{
    15,     /* tWP Minimum value from NAND02GW3B2C spec  (15 ns for 3V device)      */
    20,     /* tCS Minimum value from NAND02GW3B2C spec  (20 ns for 3V device)      */
    15,     /* tRP Minimum value from NAND02GW3B2C spec  (15 ns for 3V device)      */
    20,     /* tREA Maximum value from NAND02GW3B2C spec (20 ns for 3V device)      */
    25,     /* tCEA Maximum value from NAND02GW3B2C spec (25 ns for 3V device)      */
    2112,   /* Page Size (in bytes)                                                 */
    2048,   /* Data Area size per page (in bytes)                                   */
    64,     /* Pages per Block                                                      */
    2048,   /* Total Blocks in this device (including invalid blocks)               */
    ADI_NFD_LARGE_PAGE_x8,    /* NFD type                                           */
};
```

7.4. Using NFC Device Driver in Applications

This section explains how to use NFC device driver in an application.

7.4.1. Interrupt Manager Data memory allocation

This section explains Interrupt manager memory requirements for applications using this driver. Application must allocate three secondary interrupt memories, one to handle NFC DMA data interrupt, one for NFC DMA error interrupt and the third to handle NFC peripheral interrupts.

Interrupt manager memory required for NFC driver = (`ADI_INT_SECONDARY_MEMORY * 3`)

7.4.2. DMA Manager Data memory allocation

This section explains DMA manager memory requirements for applications using this driver. Application should allocate base memory + memory for one NFC DMA channel + memory for DMA channels used by other devices in the application

7.4.3. Device Manager Data memory allocation

This section explains device manager memory requirements for applications using this driver. The application should allocate base memory + memory for one NFC device instance + memory for other devices used by the application

7.4.4. Semaphore memory allocation

NFC driver requires memory for one semaphore of size **`ADI_SEM_SEMAPHORE_MEMORY`** to manage NFD data transfers.

7.4.5. NFC Driver initialization

Step 1: Initialize system services and device manager

Step 2: Open NFC Device driver with device specific entry point (refer section 6.1 for valid entry point)

Step 3: Pass NFD Information table

```
/*** Example ***/

/* Pass NFD information table for ADSP- BF548 Ez-Kit */
adi_dev_Control ( NFCDriverHandle,
                  ADI_NFC_CMD_PASS_NFD_INFO,
                  (void *) &gaADI_BF548EZKIT_NFD_Info);
```

Step 4: Pass address of 256-bytes buffer for NFD spare area access and NFD data packet alignment

```
/*** Example ***/

/* Global data: define a 256-bytes alignment buffer */
u8    gaNFDAlignBuf[256];
/* Pass address of 256-bytes spare area access / NFD packet alignment buffer */
adi_dev_Control ( NFCDriverHandle,
                  ADI_NFC_CMD_SET_256BYTES_ALIGN_BUFFER,
                  (void *)gaNFDAlignBuf);
```

Step 5: Pass DMA Frame buffer to boost NFC driver performance (refer section 1.2 for performance results)

```
/*** Example ***/

/* Global data:   we require memory for 8 DMA Frames (refer section 5.8.4) to
                  access one NFD page in NAND02GW3B2C (ADSP- BF548 Ez-Kit) */
u8    gaNFDDMAFrameMemory [ADI_NFC_DMA_DATA_FRAME_BASE_MEMORY * 8];

/* define a DMA frame buffer instance */
ADI_NFC_DMA_FRAME_BUFFER      aNFDDMAFrameBuf;

/* Configure DMA Frame buffer structure */
/* Pointer to DMA Frame memory */
aNFDDMAFrameBuf. pDMAFrameMemory     = (void*) gaNFDDMAFrameMemory;
/* Number of DMA Frames required to access 1 NFD page on NAND02GW3B2C */
aNFDDMAFrameBuf. NumDMAFrames        = 8;

/* Pass DMA Frame buffer to NFC driver */
adi_dev_Control ( NFCDriverHandle,
                  ADI_NFC_CMD_SET_DMA_FRAME_BUFFER,
                  (void *)&aNFDDMAFrameBuf);
```

Step 6: Set ECC Start Location error correction

```
/* set ECC start for NAND02GW3B2C NFD data packet within spare area boundary */
adi_dev_Control ( NFCDriverHandle,
                  ADI_NFC_CMD_SET_ECC_START_LOC,
                  (void *) 10);
```

Step 7: Enable / Disable error correction

```
/*** Example to enable error correction and store ECC sequentially ***/
adi_dev_Control ( NFCDriverHandle,
                  ADI_NFC_CMD_SET_ECC_MODE,
                  (void *) ADI_NFC_ECC_MODE_SEQUENTIAL);

                  (OR)

/*** Example to disable error correction ***/
adi_dev_Control ( NFCDriverHandle,
                  ADI_NFC_CMD_SET_ECC_MODE,
                  (void *) ADI_NFC_ECC_MODE_DISABLE);
```

Step 8: Setup NFC Driver dataflow mode

```
/*** Example ***/

/* NFC driver only supports chained dataflow mode */
adi_dev_Control ( NFCDriverHandle,
                  ADI_DEV_CMD_SET_DATAFLOW_METHOD,
                  (void *) ADI_DEV_MODE_CHAINED);
```

7.4.6. Access NFD directly without using DMA

Step 9: Read / Write NFD

```
/*** Example ***/

/* assume that we have to access 1kB of NAND02GW3B2C NFD data starting from
   1000th Column in 2nd page of 10th NFD block */

/* create a NFD direct access structure */
ADI_NFD_DIRECT_ACCESS   aNFDDirectAccess;
/* buffer to hold or holds NFD data */
u8    aNFDDataBuf [1024];

… …

/* configure NFD direct access structure */
/* NFD Address to access */
aNFDDirectAccess.BlockAddress       = 10;
aNFDDirectAccess.PageAddress        = 2;
aNFDDirectAccess.ColumnAddress      = 1000;
/* Pointer to NFD Data buffer */
aNFDDirectAccess.pData              = (void*)aNFDDataBuf;
/* # bytes to access */
aNFDDirectAccess.AccessCount        = (sizeof(aNFDDataBuf) /\
                                       sizeof(aNFDDataBuf[0])) ;

/* Read 1kB of NFD data */
adi_dev_Control(  NFCDriverHandle,
                  ADI_NFC_CMD_READ_NFD_DIRECT,
                  (void *)&aNFDDirectAccess);

            (OR)

/* write 1kB of data to NFD */
adi_dev_Control(  NFCDriverHandle,
                  ADI_NFC_CMD_WRITE_NFD_DIRECT,
                  (void *)&aNFDDirectAccess);
```

7.4.7. Access NFD using NFC DMA

Step 9: Read / Write NFD

```
/*** Example ***/

/* assume that we have to access 1kB of NAND02GW3B2C NFD data starting from
   1000th Column in 2nd page of 10th NFD block */

/* create a NFD address structure */
ADI_NFD_ADDRESS   aNFDAddress;
/* buffer to hold or holds NFD data */
u8    aNFDDataBuf [1024];
/* 1D buffer to handle NFD Data */
ADI_DEV_1D_BUFFER aNFD1DBuffer;
… …

/* NFD Address to access */
aNFDAddress.BlockAddress      = 10;
aNFDAddress.PageAddress       = 2;
aNFDAddress.ColumnAddress     = 1000;

/* configure NFD 1D buffer */
/* Pointer to NFD Data buffer */
aNFD1DBuffer.Data                   = (void *)aNFDDataBuf;
/* # bytes to access */
aNFD1DBuffer.ElementCount           = (sizeof(aNFDDataBuf) /\
                                       sizeof(aNFDDataBuf[0]));
/* Element width – don't care as NFC driver will fix this */
aNFD1DBuffer.ElementWidth           = 1;
aNFD1DBuffer.CallbackParameter      = NULL;
aNFD1DBuffer.ProcessedFlag          = FALSE;
aNfc1DBuffer.pNext                  = NULL;


/* pass NFD address to access */
adi_dev_Control(  NFCDriverHandle,
                  ADI_NFC_CMD_SEND_NFD_ACCESS_REQUEST,
                  (void*)&aNFDAddress);

/* Read 1kB of NFD data */
adi_dev_Read (    NFCDriverHandle,
                  ADI_DEV_1D,
                  (ADI_DEV_BUFFER *)&aNFD1DBuffer));

            (OR)

/* write 1kB of data to NFD */
adi_dev_Write (   NFCDriverHandle,
                  ADI_DEV_1D,
                  (ADI_DEV_BUFFER *)&aNFD1DBuffer));
```

7.4.8. Access NFD using physical sector address

Step 9: Read / Write NFD

```
/*** Example ***/

/* assume that we have to access 10 sectors in NAND02GW3B2C NFD starting from
   Physical sector 40 */

/* buffer to hold or holds data for 10 NFD sectors */
u8    aNFDDataBuf [512 * 10];
/* 1D buffer to handle NFD Data */
ADI_DEV_1D_BUFFER aNFD1DBuffer;
… …

/* configure NFD 1D buffer */
/* Pointer to NFD Data buffer */
aNFD1DBuffer.Data                   = (void *)aNFDDataBuf;
/* # bytes to access */
aNFD1DBuffer.ElementCount           = (sizeof(aNFDDataBuf) /\
                                       sizeof(aNFDDataBuf[0]));
/* Element width – don't care as NFC driver will fix this */
aNFD1DBuffer.ElementWidth           = 1;
aNFD1DBuffer.CallbackParameter      = NULL;
aNFD1DBuffer.ProcessedFlag          = FALSE;
aNfc1DBuffer.pNext                  = NULL;


/* pass NFD physical start sector to access */
adi_dev_Control(  NFCDriverHandle,
                  ADI_NFC_CMD_SEND_SECTOR_ACCESS_REQUEST,
                  (void *) 40);

/* Read 10 NFD sectors */
adi_dev_Read (    NFCDriverHandle,
                  ADI_DEV_1D,
                  (ADI_DEV_BUFFER *)&aNFD1DBuffer));

            (OR)

/* write to 10 NFD sectors */
adi_dev_Write (   NFCDriverHandle,
                  ADI_DEV_1D,
                  (ADI_DEV_BUFFER *)&aNFD1DBuffer));
```

7.4.9. Close NFC driver

Step 10: Close NFC driver

```
/* Example: Close NFC driver */
adi_dev_Close(NFCDriverHandle);
```