MTD NAND Driver Programming Interface

Author: Thomas Gleixner

Introduction

The generic NAND driver supports almost all NAND and AG-AND based chips and connects them to the Memory Technology Devices (MTD) subsystem of the Linux Kernel.

This documentation is provided for developers who want to implement board drivers or filesystem drivers suitable for NAND devices.

Known Bugs And Assumptions

None.

Documentation hints

The function and structure docs are autogenerated. Each function and struct member has a short description which is marked with an [XXX] identifier. The following chapters explain the meaning of those identifiers.

Function identifiers [XXX]

The functions are marked with [XXX] identifiers in the short comment. The identifiers explain the usage and scope of the functions. Following identifiers are used:

- [MTD Interface]

  These functions provide the interface to the MTD kernel API. They are not replaceable and provide functionality which is complete hardware independent.

- [NAND Interface]

  These functions are exported and provide the interface to the NAND kernel API.

- [GENERIC]

  Generic functions are not replaceable and provide functionality which is complete hardware independent.

- [DEFAULT]

  Default functions provide hardware related functionality which is suitable for most of the implementations. These functions can be replaced by the board driver if necessary. Those functions are called via pointers in the NAND chip description structure. The board driver can set the functions which should be replaced by board dependent functions before calling nand_scan(). If the function pointer is NULL on entry to nand_scan() then the pointer is set to the default function which is suitable for the detected chip type.

Struct member identifiers [XXX]

The struct members are marked with [XXX] identifiers in the comment. The identifiers explain the usage and scope of the members. Following identifiers are used:

- [INTERN]

  These members are for NAND driver internal use only and must not be modified. Most of these values are calculated from the chip geometry information which is evaluated during nand_scan().

- [REPLACEABLE]

  Replaceable members hold hardware related functions which can be provided by the board driver. The board driver can set the functions which should be replaced by board dependent functions before calling nand_scan(). If the function pointer is NULL on entry to nand_scan() then the pointer is set to the default function which is suitable for the detected chip type.

- [BOARDSPECIFIC]

  Board specific members hold hardware related information which must be provided by the board driver. The board driver must set the function pointers and datafields before calling nand_scan().

- [OPTIONAL]

  Optional members can hold information relevant for the board driver. The generic NAND driver code does not use this information.

Basic board driver

For most boards it will be sufficient to provide just the basic functions and fill out some really board dependent members in the nand chip description structure.

Basic defines

At least you have to provide a nand_chip structure and a storage for the ioremap'ed chip address. You can allocate the nand_chip structure using kmalloc or you can allocate it statically. The NAND chip structure embeds an mtd structure which will be registered to the MTD subsystem. You can extract a pointer to the mtd structure from a nand_chip pointer using the nand_to_mtd() helper.

Kmalloc based example

```
static struct mtd_info *board_mtd;
static void __iomem *baseaddr;
```

Static example

```
static struct nand_chip board_chip;
static void __iomem *baseaddr;
```

Partition defines

If you want to divide your device into partitions, then define a partitioning scheme suitable to your board.

```
#define NUM_PARTITIONS 2
static struct mtd_partition partition_info[] = {
    { .name = "Flash partition 1",
      .offset =  0,
      .size =    8 * 1024 * 1024 },
    { .name = "Flash partition 2",
      .offset =  MTDPART_OFS_NEXT,
      .size =    MTDPART_SIZ_FULL },
};
```

Hardware control function

The hardware control function provides access to the control pins of the NAND chip(s). The access can be done by GPIO pins or by address lines. If you use address lines, make sure that the timing requirements are met.

GPIO based example

```
static void board_hwcontrol(struct mtd_info *mtd, int cmd)
{
    switch(cmd){
        case NAND_CTL_SETCLE: /* Set CLE pin high */ break;
        case NAND_CTL_CLRCLE: /* Set CLE pin low */ break;
        case NAND_CTL_SETALE: /* Set ALE pin high */ break;
        case NAND_CTL_CLRALE: /* Set ALE pin low */ break;
        case NAND_CTL_SETNCE: /* Set nCE pin low */ break;
        case NAND_CTL_CLRNCE: /* Set nCE pin high */ break;
    }
}
```

Address lines based example. It's assumed that the nCE pin is driven by a chip select decoder.

```
static void board_hwcontrol(struct mtd_info *mtd, int cmd)
{
    struct nand_chip *this = mtd_to_nand(mtd);
    switch(cmd){
        case NAND_CTL_SETCLE: this->legacy.IO_ADDR_W |= CLE_ADRR_BIT;  break;
        case NAND_CTL_CLRCLE: this->legacy.IO_ADDR_W &= ~CLE_ADRR_BIT; break;
        case NAND_CTL_SETALE: this->legacy.IO_ADDR_W |= ALE_ADRR_BIT;  break;
        case NAND_CTL_CLRALE: this->legacy.IO_ADDR_W &= ~ALE_ADRR_BIT; break;
    }
}
```

Device ready function

If the hardware interface has the ready busy pin of the NAND chip connected to a GPIO or other accessible I/O pin, this function is used to read back the state of the pin. The function has no arguments and should return 0, if the device is busy (R/B pin is low) and 1, if the device is ready (R/B pin is high). If the hardware interface does not give access to the ready busy pin, then the function must not be defined and the function pointer this->legacy.dev_ready is set to NULL.

Init function

The init function allocates memory and sets up all the board specific parameters and function pointers. When everything is set up nand_scan() is called. This function tries to detect and identify then chip. If a chip is found all the internal data fields are initialized accordingly. The structure(s) have to be zeroed out first and then filled with the necessary information about the device.

```
static int __init board_init (void)
{
    struct nand_chip *this;
    int err = 0;

    /* Allocate memory for MTD device structure and private data */
    this = kzalloc(sizeof(struct nand_chip), GFP_KERNEL);
    if (!this) {
        printk ("Unable to allocate NAND MTD device structure.\n");
        err = -ENOMEM;
        goto out;
    }

    board_mtd = nand_to_mtd(this);

    /* map physical address */
    baseaddr = ioremap(CHIP_PHYSICAL_ADDRESS, 1024);
    if (!baseaddr) {
        printk("Ioremap to access NAND chip failed\n");
        err = -EIO;
        goto out_mtd;
    }

    /* Set address of NAND IO lines */
    this->legacy.IO_ADDR_R = baseaddr;
    this->legacy.IO_ADDR_W = baseaddr;
    /* Reference hardware control function */
    this->hwcontrol = board_hwcontrol;
    /* Set command delay time, see datasheet for correct value */
    this->legacy.chip_delay = CHIP_DEPENDEND_COMMAND_DELAY;
    /* Assign the device ready function, if available */
    this->legacy.dev_ready = board_dev_ready;
    this->eccmode = NAND_ECC_SOFT;

    /* Scan to find existence of the device */
    if (nand_scan (this, 1)) {
        err = -ENXIO;
        goto out_ior;
    }

    add_mtd_partitions(board_mtd, partition_info, NUM_PARTITIONS);
    goto out;

out_ior:
    iounmap(baseaddr);
out_mtd:
    kfree (this);
out:
    return err;
}
module_init(board_init);
```

Exit function

The exit function is only necessary if the driver is compiled as a module. It releases all resources which are held by the chip driver and unregisters the partitions in the MTD layer.

```
#ifdef MODULE
static void __exit board_cleanup (void)
{
    /* Unregister device */
    WARN_ON(mtd_device_unregister(board_mtd));
    /* Release resources */
    nand_cleanup(mtd_to_nand(board_mtd));

    /* unmap physical address */
    iounmap(baseaddr);

    /* Free the MTD device structure */
    kfree (mtd_to_nand(board_mtd));
}
module_exit(board_cleanup);
#endif
```

Advanced board driver functions

This chapter describes the advanced functionality of the NAND driver. For a list of functions which can be overridden by the board driver see the documentation of the nand_chip structure.

Multiple chip control

Multiple chip control

The nand driver can control chip arrays. Therefore the board driver must provide an own select_chip function. This function must (de)select the requested chip. The function pointer in the nand_chip structure must be set before calling nand_scan(). The maxchip parameter of nand_scan() defines the maximum number of chips to scan for. Make sure that the select_chip function can handle the requested number of chips.

The nand driver concatenates the chips to one virtual chip and provides this virtual chip to the MTD layer.

Note: The driver can only handle linear chip arrays of equally sized chips. There is no support for parallel arrays which extend the buswidth.

GPIO based example

```
static void board_select_chip (struct mtd_info *mtd, int chip)
{
    /* Deselect all chips, set all nCE pins high */
    GPIO(BOARD_NAND_NCE) |= 0xff;
    if (chip >= 0)
        GPIO(BOARD_NAND_NCE) &= ~ (1 << chip);
}
```

Address lines based example. Its assumed that the nCE pins are connected to an address decoder.

```
static void board_select_chip (struct mtd_info *mtd, int chip)
{
    struct nand_chip *this = mtd_to_nand(mtd);

    /* Deselect all chips */
    this->legacy.IO_ADDR_R &= ~BOARD_NAND_ADDR_MASK;
    this->legacy.IO_ADDR_W &= ~BOARD_NAND_ADDR_MASK;
    switch (chip) {
    case 0:
        this->legacy.IO_ADDR_R |= BOARD_NAND_ADDR_CHIP0;
        this->legacy.IO_ADDR_W |= BOARD_NAND_ADDR_CHIP0;
        break;
    ....
    case n:
        this->legacy.IO_ADDR_R |= BOARD_NAND_ADDR_CHIPn;
        this->legacy.IO_ADDR_W |= BOARD_NAND_ADDR_CHIPn;
        break;
    }
}
```

Hardware ECC support

Functions and constants

The nand driver supports three different types of hardware ECC.

- NAND_ECC_HW3_256

  Hardware ECC generator providing 3 bytes ECC per 256 byte.
- NAND_ECC_HW3_512

  Hardware ECC generator providing 3 bytes ECC per 512 byte.
- NAND_ECC_HW6_512

  Hardware ECC generator providing 6 bytes ECC per 512 byte.
- NAND_ECC_HW8_512

  Hardware ECC generator providing 8 bytes ECC per 512 byte.

If your hardware generator has a different functionality add it at the appropriate place in nand_base.c

The board driver must provide following functions:

- enable_hwecc

  This function is called before reading / writing to the chip. Reset or initialize the hardware generator in this function. The function is called with an argument which let you distinguish between read and write operations.
- calculate_ecc

  This function is called after read / write from / to the chip. Transfer the ECC from the hardware to the buffer. If the option NAND_HWECC_SYNDROME is set then the function is only called on write. See below.
- correct_data

  In case of an ECC error this function is called for error detection and correction. Return 1 respectively 2 in case the error can be corrected. If the error is not correctable return -1. If your hardware generator matches the default algorithm of the

nand_ecc software generator then use the correction function provided by nand_ecc instead of implementing duplicated code.

Hardware ECC with syndrome calculation

Many hardware ECC implementations provide Reed-Solomon codes and calculate an error syndrome on read. The syndrome must be converted to a standard Reed-Solomon syndrome before calling the error correction code in the generic Reed-Solomon library.

The ECC bytes must be placed immediately after the data bytes in order to make the syndrome generator work. This is contrary to the usual layout used by software ECC. The separation of data and out of band area is not longer possible. The nand driver code handles this layout and the remaining free bytes in the oob area are managed by the autoplacement code. Provide a matching oob-layout in this case. See rts_from4.c and diskonchip.c for implementation reference. In those cases we must also use bad block tables on FLASH, because the ECC layout is interfering with the bad block marker positions. See bad block table support for details.

Bad block table support

Most NAND chips mark the bad blocks at a defined position in the spare area. Those blocks must not be erased under any circumstances as the bad block information would be lost. It is possible to check the bad block mark each time when the blocks are accessed by reading the spare area of the first page in the block. This is time consuming so a bad block table is used.

The nand driver supports various types of bad block tables.

- Per device

  The bad block table contains all bad block information of the device which can consist of multiple chips.

- Per chip

  A bad block table is used per chip and contains the bad block information for this particular chip.

- Fixed offset

  The bad block table is located at a fixed offset in the chip (device). This applies to various DiskOnChip devices.

- Automatic placed

  The bad block table is automatically placed and detected either at the end or at the beginning of a chip (device)

- Mirrored tables

  The bad block table is mirrored on the chip (device) to allow updates of the bad block table without data loss.

nand_scan() calls the function nand_default_bbt(). nand_default_bbt() selects appropriate default bad block table descriptors depending on the chip information which was retrieved by nand_scan().

The standard policy is scanning the device for bad blocks and build a ram based bad block table which allows faster access than always checking the bad block information on the flash chip itself.

Flash based tables

It may be desired or necessary to keep a bad block table in FLASH. For AG-AND chips this is mandatory, as they have no factory marked bad blocks. They have factory marked good blocks. The marker pattern is erased when the block is erased to be reused. So in case of powerloss before writing the pattern back to the chip this block would be lost and added to the bad blocks. Therefore we scan the chip(s) when we detect them the first time for good blocks and store this information in a bad block table before erasing any of the blocks.

The blocks in which the tables are stored are protected against accidental access by marking them bad in the memory bad block table. The bad block table management functions are allowed to circumvent this protection.

The simplest way to activate the FLASH based bad block table support is to set the option NAND_BBT_USE_FLASH in the bbt_option field of the nand chip structure before calling nand_scan(). For AG-AND chips is this done by default. This activates the default FLASH based bad block table functionality of the NAND driver. The default bad block table options are

- Store bad block table per chip
- Use 2 bits per block
- Automatic placement at the end of the chip
- Use mirrored tables with version numbers
- Reserve 4 blocks at the end of the chip

User defined tables

User defined tables are created by filling out a nand_bbt_descr structure and storing the pointer in the nand_chip structure member bbt_td before calling nand_scan(). If a mirror table is necessary a second structure must be created and a pointer to this structure must be stored in bbt_md inside the nand_chip structure. If the bbt_md member is set to NULL then only the main table is used and no scan for the mirrored table is performed.

The most important field in the nand_bbt_descr structure is the options field. The options define most of the table properties. Use the predefined constants from rawnand.h to define the options.

- Number of bits per block

  The supported number of bits is 1, 2, 4, 8.

- Table per chip

  Setting the constant NAND_BBT_PERCHIP selects that a bad block table is managed for each chip in a chip array. If this option is not set then a per device bad block table is used.

- Table location is absolute

  Use the option constant NAND_BBT_ABSPAGE and define the absolute page number where the bad block table starts in the field pages. If you have selected bad block tables per chip and you have a multi chip array then the start page must be given for each chip in the chip array. Note: there is no scan for a table ident pattern performed, so the fields pattern, veroffs, offs, len can be left uninitialized

- Table location is automatically detected

  The table can either be located in the first or the last good blocks of the chip (device). Set NAND_BBT_LASTBLOCK to place the bad block table at the end of the chip (device). The bad block tables are marked and identified by a pattern which is stored in the spare area of the first page in the block which holds the bad block table. Store a pointer to the pattern in the pattern field. Further the length of the pattern has to be stored in len and the offset in the spare area must be given in the offs member of the nand_bbt_descr structure. For mirrored bad block tables different patterns are mandatory.

- Table creation

  Set the option NAND_BBT_CREATE to enable the table creation if no table can be found during the scan. Usually this is done only once if a new chip is found.

- Table write support

  Set the option NAND_BBT_WRITE to enable the table write support. This allows the update of the bad block table(s) in case a block has to be marked bad due to wear. The MTD interface function block_markbad is calling the update function of the bad block table. If the write support is enabled then the table is updated on FLASH.

  Note: Write support should only be enabled for mirrored tables with version control.

- Table version control

  Set the option NAND_BBT_VERSION to enable the table version control. It's highly recommended to enable this for mirrored tables with write support. It makes sure that the risk of losing the bad block table information is reduced to the loss of the information about the one worn out block which should be marked bad. The version is stored in 4 consecutive bytes in the spare area of the device. The position of the version number is defined by the member veroffs in the bad block table descriptor.

- Save block contents on write

  In case that the block which holds the bad block table does contain other useful information, set the option NAND_BBT_SAVECONTENT. When the bad block table is written then the whole block is read the bad block table is updated and the block is erased and everything is written back. If this option is not set only the bad block table is written and everything else in the block is ignored and erased.

- Number of reserved blocks

  For automatic placement some blocks must be reserved for bad block table storage. The number of reserved blocks is defined in the maxblocks member of the bad block table description structure. Reserving 4 blocks for mirrored tables should be a reasonable number. This also limits the number of blocks which are scanned for the bad block table ident pattern.

Spare area (auto)placement

The nand driver implements different possibilities for placement of filesystem data in the spare area,

- Placement defined by fs driver
- Automatic placement

The default placement function is automatic placement. The nand driver has built in default placement schemes for the various chiptypes. If due to hardware ECC functionality the default placement does not fit then the board driver can provide a own placement scheme.

File system drivers can provide a own placement scheme which is used instead of the default placement scheme.

Placement schemes are defined by a nand_oobinfo structure

```
struct nand_oobinfo {
    int useecc;
    int eccbytes;
    int eccpos[24];
    int oobfree[8][2];
};
```

- useecc

The useecc member controls the ecc and placement function. The header file include/mtd/mtd-abi.h contains constants to select ecc and placement. MTD_NANDECC_OFF switches off the ecc complete. This is not recommended and available for testing and diagnosis only. MTD_NANDECC_PLACE selects caller defined placement, MTD_NANDECC_AUTOPLACE selects automatic placement.

- eccbytes

  The eccbytes member defines the number of ecc bytes per page.

- eccpos

  The eccpos array holds the byte offsets in the spare area where the ecc codes are placed.

- oobfree

  The oobfree array defines the areas in the spare area which can be used for automatic placement. The information is given in the format {offset, size}. offset defines the start of the usable area, size the length in bytes. More than one area can be defined. The list is terminated by an {0, 0} entry.

Placement defined by fs driver

The calling function provides a pointer to a nand_oobinfo structure which defines the ecc placement. For writes the caller must provide a spare area buffer along with the data buffer. The spare area buffer size is (number of pages) * (size of spare area). For reads the buffer size is (number of pages) * ((size of spare area) + (number of ecc steps per page) * sizeof (int)). The driver stores the result of the ecc check for each tuple in the spare buffer. The storage sequence is:

```
<spare data page 0><ecc result 0>...<ecc result n>

...

<spare data page n><ecc result 0>...<ecc result n>
```

This is a legacy mode used by YAFFS1.

If the spare area buffer is NULL then only the ECC placement is done according to the given scheme in the nand_oobinfo structure.

Automatic placement

Automatic placement uses the built in defaults to place the ecc bytes in the spare area. If filesystem data have to be stored / read into the spare area then the calling function must provide a buffer. The buffer size per page is determined by the oobfree array in the nand_oobinfo structure.

If the spare area buffer is NULL then only the ECC placement is done according to the default builtin scheme.

Spare area autoplacement default schemes

256 byte pagesize

Offset	Content	Comment
0x00	ECC byte 0	Error correction code byte 0
0x01	ECC byte 1	Error correction code byte 1
0x02	ECC byte 2	Error correction code byte 2
0x03	Autoplace 0	
0x04	Autoplace 1	
0x05	Bad block marker	If any bit in this byte is zero, then this block is bad. This applies only to the first page in a block. In the remaining pages this byte is reserved
0x06	Autoplace 2	
0x07	Autoplace 3	

512 byte pagesize

Offset	Content	Comment
0x00	ECC byte 0	Error correction code byte 0 of the lower 256 Byte data in this page
0x01	ECC byte 1	Error correction code byte 1 of the lower 256 Bytes of data in this page
0x02	ECC byte 2	Error correction code byte 2 of the lower 256 Bytes of data in this page
0x03	ECC byte 3	Error correction code byte 0 of the upper 256 Bytes of data in this page
0x04	reserved	reserved
0x05	Bad block marker	If any bit in this byte is zero, then this block is bad. This applies only to the first page in a block. In the remaining pages this byte is reserved
0x06	ECC byte 4	Error correction code byte 1 of the upper 256 Bytes of data in this page
0x07	ECC byte 5	Error correction code byte 2 of the upper 256 Bytes of data in this page
0x08 - 0x0F	Autoplace 0 - 7	

2048 byte pagesize

Offset	Content	Comment
0x00	Bad block marker	If any bit in this byte is zero, then this block is bad. This applies only to the first page in a block. In the remaining pages this byte is reserved
0x01	Reserved	Reserved
0x02-0x27	Autoplace 0 - 37	
0x28	ECC byte 0	Error correction code byte 0 of the first 256 Byte data in this page
0x29	ECC byte 1	Error correction code byte 1 of the first 256 Bytes of data in this page
0x2A	ECC byte 2	Error correction code byte 2 of the first 256 Bytes data in this page
0x2B	ECC byte 3	Error correction code byte 0 of the second 256 Bytes of data in this page
0x2C	ECC byte 4	Error correction code byte 1 of the second 256 Bytes of data in this page
0x2D	ECC byte 5	Error correction code byte 2 of the second 256 Bytes of data in this page
0x2E	ECC byte 6	Error correction code byte 0 of the third 256 Bytes of data in this page
0x2F	ECC byte 7	Error correction code byte 1 of the third 256 Bytes of data in this page
0x30	ECC byte 8	Error correction code byte 2 of the third 256 Bytes of data in this page
0x31	ECC byte 9	Error correction code byte 0 of the fourth 256 Bytes of data in this page
0x32	ECC byte 10	Error correction code byte 1 of the fourth 256 Bytes of data in this page
0x33	ECC byte 11	Error correction code byte 2 of the fourth 256 Bytes of data in this page
0x34	ECC byte 12	Error correction code byte 0 of the fifth 256 Bytes of data in this page
0x35	ECC byte 13	Error correction code byte 1 of the fifth 256 Bytes of data in this page
0x36	ECC byte 14	Error correction code byte 2 of the fifth 256 Bytes of data in this page
0x37	ECC byte 15	Error correction code byte 0 of the sixth 256 Bytes of data in this page
0x38	ECC byte 16	Error correction code byte 1 of the sixth 256 Bytes of data in this page
0x39	ECC byte 17	Error correction code byte 2 of the sixth 256 Bytes of data in this page
0x3A	ECC byte 18	Error correction code byte 0 of the seventh 256 Bytes of data in this page
0x3B	ECC byte 19	Error correction code byte 1 of the seventh 256 Bytes of data in this page
0x3C	ECC byte 20	Error correction code byte 2 of the seventh 256 Bytes of data in this page
0x3D	ECC byte 21	Error correction code byte 0 of the eighth 256 Bytes of data in this page
0x3E	ECC byte 22	Error correction code byte 1 of the eighth 256 Bytes of data in this page
0x3F	ECC byte 23	Error correction code byte 2 of the eighth 256 Bytes of data in this page

Filesystem support

The NAND driver provides all necessary functions for a filesystem via the MTD interface.

Filesystems must be aware of the NAND peculiarities and restrictions. One major restrictions of NAND Flash is, that you cannot write as often as you want to a page. The consecutive writes to a page, before erasing it again, are restricted to 1-3 writes, depending on the manufacturers specifications. This applies similar to the spare area.

Therefore NAND aware filesystems must either write in page size chunks or hold a writebuffer to collect smaller writes until they sum up to pagesize. Available NAND aware filesystems: JFFS2, YAFFS.

The spare area usage to store filesystem data is controlled by the spare area placement functionality which is described in one of the earlier chapters.

Tools

The MTD project provides a couple of helpful tools to handle NAND Flash.

- flasherase, flasheraseall: Erase and format FLASH partitions
- nandwrite: write filesystem images to NAND FLASH
- nanddump: dump the contents of a NAND FLASH partitions

These tools are aware of the NAND restrictions. Please use those tools instead of complaining about errors which are caused by non NAND aware access methods.

Constants

This chapter describes the constants which might be relevant for a driver developer.

Chip option constants

Constants for chip id table

These constants are defined in rawnand.h. They are OR-ed together to describe the chip functionality:

```
/* Buswitdh is 16 bit */
```

```
#define NAND_BUSWIDTH_16    0x00000002
/* Device supports partial programming without padding */
#define NAND_NO_PADDING     0x00000004
/* Chip has cache program function */
#define NAND_CACHEPRG       0x00000008
/* Chip has copy back function */
#define NAND_COPYBACK       0x00000010
/* AND Chip which has 4 banks and a confusing page / block
 * assignment. See Renesas datasheet for further information */
#define NAND_IS_AND     0x00000020
/* Chip has a array of 4 pages which can be read without
 * additional ready /busy waits */
#define NAND_4PAGE_ARRAY    0x00000040
```

Constants for runtime options

These constants are defined in rawnand.h. They are OR-ed together to describe the functionality:

```
/* The hw ecc generator provides a syndrome instead a ecc value on read
 * This can only work if we have the ecc bytes directly behind the
 * data bytes. Applies for DOC and AG-AND Renesas HW Reed Solomon generators */
#define NAND_HWECC_SYNDROME 0x00020000
```

ECC selection constants

Use these constants to select the ECC algorithm:

```
/* No ECC. Usage is not recommended ! */
#define NAND_ECC_NONE       0
/* Software ECC 3 byte ECC per 256 Byte data */
#define NAND_ECC_SOFT       1
/* Hardware ECC 3 byte ECC per 256 Byte data */
#define NAND_ECC_HW3_256    2
/* Hardware ECC 3 byte ECC per 512 Byte data */
#define NAND_ECC_HW3_512    3
/* Hardware ECC 6 byte ECC per 512 Byte data */
#define NAND_ECC_HW6_512    4
/* Hardware ECC 8 byte ECC per 512 Byte data */
#define NAND_ECC_HW8_512    6
```

Hardware control related constants

These constants describe the requested hardware access function when the boardspecific hardware control function is called:

```
/* Select the chip by setting nCE to low */
#define NAND_CTL_SETNCE     1
/* Deselect the chip by setting nCE to high */
#define NAND_CTL_CLRNCE     2
/* Select the command latch by setting CLE to high */
#define NAND_CTL_SETCLE     3
/* Deselect the command latch by setting CLE to low */
#define NAND_CTL_CLRCLE     4
/* Select the address latch by setting ALE to high */
#define NAND_CTL_SETALE     5
/* Deselect the address latch by setting ALE to low */
#define NAND_CTL_CLRALE     6
/* Set write protection by setting WP to high. Not used! */
#define NAND_CTL_SETWP      7
/* Clear write protection by setting WP to low. Not used! */
#define NAND_CTL_CLRWP      8
```

Bad block table related constants

These constants describe the options used for bad block table descriptors:

```
/* Options for the bad block table descriptors */

/* The number of bits used per block in the bbt on the device */
#define NAND_BBT_NRBITS_MSK 0x0000000F
#define NAND_BBT_1BIT       0x00000001
#define NAND_BBT_2BIT       0x00000002
#define NAND_BBT_4BIT       0x00000004
#define NAND_BBT_8BIT       0x00000008
/* The bad block table is in the last good block of the device */
#define NAND_BBT_LASTBLOCK  0x00000010
/* The bbt is at the given page, else we must scan for the bbt */
#define NAND_BBT_ABSPAGE    0x00000020
/* bbt is stored per chip on multichip devices */
#define NAND_BBT_PERCHIP    0x00000080
/* bbt has a version counter at offset veroffs */
```

```
#define NAND_BBT_VERSION    0x00000100
/* Create a bbt if none axists */
#define NAND_BBT_CREATE     0x00000200
/* Write bbt if necessary */
#define NAND_BBT_WRITE      0x00001000
/* Read and write back block contents when writing bbt */
#define NAND_BBT_SAVECONTENT    0x00002000
```

Structures

This chapter contains the autogenerated documentation of the structures which are used in the NAND driver and might be relevant for a driver developer. Each struct member has a short description which is marked with an [XXX] identifier. See the chapter "Documentation hints" for an explanation.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\[linux-master][Documentation][driver-api]mtdnand.rst, line 961)

Unknown directive type "kernel-doc".

```
.. kernel-doc:: include/linux/mtd/rawnand.h
   :internal:
```

Public Functions Provided

This chapter contains the autogenerated documentation of the NAND kernel API functions which are exported. Each function has a short description which is marked with an [XXX] identifier. See the chapter "Documentation hints" for an explanation.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\[linux-master][Documentation][driver-api]mtdnand.rst, line 972)

Unknown directive type "kernel-doc".

```
.. kernel-doc:: drivers/mtd/nand/raw/nand_base.c
   :export:
```

Internal Functions Provided

This chapter contains the autogenerated documentation of the NAND driver internal functions. Each function has a short description which is marked with an [XXX] identifier. See the chapter "Documentation hints" for an explanation. The functions marked with [DEFAULT] might be relevant for a board driver developer.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\[linux-master][Documentation][driver-api]mtdnand.rst, line 984)

Unknown directive type "kernel-doc".

```
.. kernel-doc:: drivers/mtd/nand/raw/nand_base.c
   :internal:
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\[linux-master][Documentation][driver-api]mtdnand.rst, line 987)

Unknown directive type "kernel-doc".

```
.. kernel-doc:: drivers/mtd/nand/raw/nand_bbt.c
   :internal:
```

Credits

The following people have contributed to the NAND driver:

1. Steven J. Hillsjhill@realitydiluted.com
2. David Woodhousedwmw2@infradead.org

3. Thomas Gleixnertglx@linutronix.de

A lot of users have provided bugfixes, improvements and helping hands for testing. Thanks a lot.

The following people have contributed to this document:

1. Thomas Gleixnertglx@linutronix.de