

# **MTD NAND Driver Programming Interface**

**Thomas Gleixner** <tglx@linutronix.de>

---

# MTD NAND Driver Programming Interface

by Thomas Gleixner

Copyright © 2004 Thomas Gleixner

This documentation is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License version 2 as published by the Free Software Foundation.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

For more details see the file COPYING in the source distribution of Linux.

---

---

# Table of Contents

1. Introduction .....	1
2. Known Bugs And Assumptions .....	2
3. Documentation hints .....	3
Function identifiers [XXX] .....	3
Struct member identifiers [XXX] .....	3
4. Basic board driver .....	5
Basic defines .....	5
Partition defines .....	5
Hardware control function .....	5
Device ready function .....	6
Init function .....	6
Exit function .....	7
5. Advanced board driver functions .....	9
Multiple chip control .....	9
Hardware ECC support .....	10
Functions and constants .....	10
Hardware ECC with syndrome calculation .....	10
Bad block table support .....	11
Flash based tables .....	11
User defined tables .....	12
Spare area (auto)placement .....	13
Placement defined by fs driver .....	14
Automatic placement .....	14
Spare area autoplacement default schemes .....	15
256 byte pagesize .....	15
512 byte pagesize .....	15
2048 byte pagesize .....	16
6. Filesystem support .....	18
7. Tools .....	19
8. Constants .....	20
Chip option constants .....	20
Constants for chip id table .....	20
Constants for runtime options .....	20
ECC selection constants .....	20
Hardware control related constants .....	21
Bad block table related constants .....	21
9. Structures .....	23
struct nand_hw_control .....	24
struct nand_ecc_ctrl .....	25
struct nand_buffers .....	27
struct nand_sdr_timings .....	28
enum nand_data_interface_type .....	31
struct nand_data_interface .....	32
nand_get_sdr_timings .....	33
struct nand_chip .....	34
struct nand_flash_dev .....	38
struct nand_manufacturers .....	39
struct platform_nand_chip .....	40
struct platform_nand_ctrl .....	41
struct platform_nand_data .....	42
nand_opcode_8bits .....	43

10. Public Functions Provided .....	44
nand_wait_ready .....	45
nand_unlock .....	46
nand_lock .....	47
nand_check_erased_ecc_chunk .....	48
nand_read_oob_std .....	49
nand_read_oob_syndrome .....	50
nand_write_oob_std .....	51
nand_write_oob_syndrome .....	52
nand_scan_ident .....	53
nand_scan_tail .....	54
nand_scan .....	55
nand_cleanup .....	56
nand_release .....	57
drivers/mtd/nand/nand_bbt.c .....	58
__nand_calculate_ecc .....	59
nand_calculate_ecc .....	60
__nand_correct_data .....	61
nand_correct_data .....	62
11. Internal Functions Provided .....	63
nand_release_device .....	64
nand_read_byte .....	65
nand_read_byte16 .....	66
nand_read_word .....	67
nand_select_chip .....	68
nand_write_byte .....	69
nand_write_byte16 .....	70
nand_write_buf .....	71
nand_read_buf .....	72
nand_write_buf16 .....	73
nand_read_buf16 .....	74
nand_block_bad .....	75
nand_default_block_markbad .....	76
nand_block_markbad_lowlevel .....	77
nand_check_wp .....	78
nand_block_isreserved .....	79
nand_block_checkbad .....	80
panic_nand_wait_ready .....	81
nand_wait_status_ready .....	82
nand_command .....	83
nand_command_lp .....	84
panic_nand_get_device .....	85
nand_get_device .....	86
panic_nand_wait .....	87
nand_wait .....	88
nand_reset_data_interface .....	89
nand_setup_data_interface .....	90
nand_init_data_interface .....	91
nand_reset .....	92
__nand_unlock .....	93
nand_check_erased_buf .....	94
nand_read_page_raw .....	95
nand_read_page_raw_syndrome .....	96
nand_read_page_swecc .....	97

MTD NAND Driver  
Programming Interface

---

nand_read_subpage .....	98
nand_read_page_hwecc .....	99
nand_read_page_hwecc_oob_first .....	100
nand_read_page_syndrome .....	101
nand_transfer_oob .....	102
nand_setup_read_retry .....	103
nand_do_read_ops .....	104
nand_read .....	105
nand_do_read_oob .....	106
nand_read_oob .....	107
nand_write_page_raw .....	108
nand_write_page_raw_syndrome .....	109
nand_write_page_swecc .....	110
nand_write_page_hwecc .....	111
nand_write_subpage_hwecc .....	112
nand_write_page_syndrome .....	113
nand_write_page .....	114
nand_fill_oob .....	115
nand_do_write_ops .....	116
panic_nand_write .....	117
nand_write .....	118
nand_do_write_oob .....	119
nand_write_oob .....	120
single_erase .....	121
nand_erase .....	122
nand_erase_nand .....	123
nand_sync .....	124
nand_block_isbad .....	125
nand_block_markbad .....	126
nand_onfi_set_features .....	127
nand_onfi_get_features .....	128
nand_suspend .....	129
nand_resume .....	130
nand_shutdown .....	131
check_pattern .....	132
check_short_pattern .....	133
add_marker_len .....	134
read_bbt .....	135
read_abs_bbt .....	136
scan_read_oob .....	137
read_abs_bbts .....	138
create_bbt .....	139
search_bbt .....	140
search_read_bbts .....	141
get_bbt_block .....	142
mark_bbt_block_bad .....	143
write_bbt .....	144
nand_memory_bbt .....	145
check_create .....	146
mark_bbt_region .....	147
verify_bbt_descr .....	148
nand_scan_bbt .....	149
nand_update_bbt .....	150
nand_create_badblock_pattern .....	151

MTD NAND Driver  
Programming Interface

---

nand_default_bbt .....	152
nand_isreserved_bbt .....	153
nand_isbad_bbt .....	154
nand_markbad_bbt .....	155
12. Credits .....	156

---

# Chapter 1. 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.

---

## Chapter 2. Known Bugs And Assumptions

None.



---

# Chapter 3. 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.

---

# Chapter 4. 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->IO_ADDR_W |= CLE_ADDR_BIT; break;
case NAND_CTL_CLRCLE: this->IO_ADDR_W &= ~CLE_ADDR_BIT; break;
case NAND_CTL_SETALE: this->IO_ADDR_W |= ALE_ADDR_BIT; break;
case NAND_CTL_CLRALE: this->IO_ADDR_W &= ~ALE_ADDR_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->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->IO_ADDR_R = baseaddr;
this->IO_ADDR_W = baseaddr;
/* Reference hardware control function */
this->hwcontrol = board_hwcontrol;
/* Set command delay time, see datasheet for correct value */
this->chip_delay = CHIP_DEPENDEND_COMMAND_DELAY;
/* Assign the device ready function, if available */
this->dev_ready = board_dev_ready;
this->eccmode = NAND_ECC_SOFT;

/* Scan to find existence of the device */
if (nand_scan (board_mtd, 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)
{
    /* Release resources, unregister device */
}
```

```
nand_release (board_mtd);

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

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

---

# Chapter 5. 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

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->IO_ADDR_R &= ~BOARD_NAND_ADDR_MASK;
    this->IO_ADDR_W &= ~BOARD_NAND_ADDR_MASK;
    switch (chip) {
    case 0:
        this->IO_ADDR_R |= BOARD_NAND_ADDR_CHIP0;
        this->IO_ADDR_W |= BOARD_NAND_ADDR_CHIP0;
        break;
    ....
    case n:
        this->IO_ADDR_R |= BOARD_NAND_ADDR_CHIPn;
        this->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 6 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 `nand.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

---

# Chapter 6. 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.



---

# Chapter 7. 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.

---

# Chapter 8. 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 nand.h. They are ored together to describe the chip functionality.

```
/* Buswidth 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 nand.h. They are ored 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 6 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 exists */
#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
```

---

# Chapter 9. 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.

## Name

struct nand\_hw\_control — Control structure for hardware controller (e.g ECC generator) shared among independent devices

## Synopsis

```
struct nand_hw_control {  
    spinlock_t lock;  
    struct nand_chip * active;  
    wait_queue_head_t wq;  
};
```

## Members

lock	protection lock
active	the mtd device which holds the controller currently
wq	wait queue to sleep on if a NAND operation is in progress used instead of the per chip wait queue when a hw controller is available.

## Name

struct nand\_ecc\_ctrl — Control structure for ECC

## Synopsis

```
struct nand_ecc_ctrl {
    nand_ecc_modes_t mode;
    enum nand_ecc_algo algo;
    int steps;
    int size;
    int bytes;
    int total;
    int strength;
    int prepad;
    int postpad;
    unsigned int options;
    void * priv;
    void (* hwctl) (struct mtd_info *mtd, int mode);
    int (* calculate) (struct mtd_info *mtd, const uint8_t *dat, uint8_t *ecc_code);
    int (* correct) (struct mtd_info *mtd, uint8_t *dat, uint8_t *read_ecc, uint8_t *buf);
    int (* read_page_raw) (struct mtd_info *mtd, struct nand_chip *chip, uint8_t *buf, int page);
    int (* write_page_raw) (struct mtd_info *mtd, struct nand_chip *chip, const uint8_t *buf, int page);
    int (* read_page) (struct mtd_info *mtd, struct nand_chip *chip, uint8_t *buf, int page);
    int (* read_subpage) (struct mtd_info *mtd, struct nand_chip *chip, uint32_t offs, int page);
    int (* write_subpage) (struct mtd_info *mtd, struct nand_chip *chip, uint32_t offs, int page);
    int (* write_page) (struct mtd_info *mtd, struct nand_chip *chip, const uint8_t *buf, int page);
    int (* write_oob_raw) (struct mtd_info *mtd, struct nand_chip *chip, int page);
    int (* read_oob_raw) (struct mtd_info *mtd, struct nand_chip *chip, int page);
    int (* read_oob) (struct mtd_info *mtd, struct nand_chip *chip, int page);
    int (* write_oob) (struct mtd_info *mtd, struct nand_chip *chip, int page);
};
```

## Members

mode	ECC mode
algo	ECC algorithm
steps	number of ECC steps per page
size	data bytes per ECC step
bytes	ECC bytes per step
total	total number of ECC bytes per page
strength	max number of correctible bits per ECC step
prepad	padding information for syndrome based ECC generators
postpad	padding information for syndrome based ECC generators
options	ECC specific options (see NAND_ECC_XXX flags defined above)

priv	pointer to private ECC control data
hwctl	function to control hardware ECC generator. Must only be provided if an hardware ECC is available
calculate	function for ECC calculation or readback from ECC hardware
correct	function for ECC correction, matching to ECC generator (sw/hw). Should return a positive number representing the number of corrected bitflips, -EBADMSG if the number of bitflips exceed ECC strength, or any other error code if the error is not directly related to correction. If -EBADMSG is returned the input buffers should be left untouched.
read_page_raw	function to read a raw page without ECC. This function should hide the specific layout used by the ECC controller and always return contiguous in-band and out-of-band data even if they're not stored contiguously on the NAND chip (e.g. NAND_ECC_HW_SYNDROME interleaves in-band and out-of-band data).
write_page_raw	function to write a raw page without ECC. This function should hide the specific layout used by the ECC controller and consider the passed data as contiguous in-band and out-of-band data. ECC controller is responsible for doing the appropriate transformations to adapt to its specific layout (e.g. NAND_ECC_HW_SYNDROME interleaves in-band and out-of-band data).
read_page	function to read a page according to the ECC generator requirements; returns maximum number of bitflips corrected in any single ECC step, 0 if bitflips uncorrectable, -EIO hw error
read_subpage	function to read parts of the page covered by ECC; returns same as read_page
write_subpage	function to write parts of the page covered by ECC.
write_page	function to write a page according to the ECC generator requirements.
write_oob_raw	function to write chip OOB data without ECC
read_oob_raw	function to read chip OOB data without ECC
read_oob	function to read chip OOB data
write_oob	function to write chip OOB data



## Name

struct nand\_buffers — buffer structure for read/write

## Synopsis

```
struct nand_buffers {  
    uint8_t * ecccalc;  
    uint8_t * ecccode;  
    uint8_t * databuf;  
};
```

## Members

ecccalc	buffer pointer for calculated ECC, size is oobsize.
ecccode	buffer pointer for ECC read from flash, size is oobsize.
databuf	buffer pointer for data, size is (page size + oobsize).

## Description

Do not change the order of buffers. databuf and oobrbuf must be in consecutive order.

## Name

struct nand\_sdr\_timings — SDR NAND chip timings

## Synopsis

```
struct nand_sdr_timings {
    u32 tALH_min;
    u32 tADL_min;
    u32 tALS_min;
    u32 tAR_min;
    u32 tCEA_max;
    u32 tCH_min;
    u32 tCHZ_max;
    u32 tCLH_min;
    u32 tCLR_min;
    u32 tCLS_min;
    u32 tCOH_min;
    u32 tCS_min;
    u32 tDH_min;
    u32 tDS_min;
    u32 tFEAT_max;
    u32 tIR_min;
    u32 tITC_max;
    u32 tRC_min;
    u32 tREA_max;
    u32 tREH_min;
    u32 tRHOH_min;
    u32 tRHW_min;
    u32 tRHZ_max;
    u32 tRLOH_min;
    u32 tRP_min;
    u32 tRR_min;
    u64 tRST_max;
    u32 tWB_max;
    u32 tWC_min;
    u32 tWH_min;
    u32 tWHR_min;
    u32 tWP_min;
    u32 tWW_min;
};
```

## Members

tALH_min	ALE hold time
tADL_min	ALE to data loading time
tALS_min	ALE setup time
tAR_min	ALE to RE# delay
tCEA_max	CE# access time

tCH_min	CE# hold time
tCHZ_max	CE# high to output hi-Z
tCLH_min	CLE hold time
tCLR_min	CLE to RE# delay
tCLS_min	CLE setup time
tCOH_min	CE# high to output hold
tCS_min	CE# setup time
tDH_min	Data hold time
tDS_min	Data setup time
tFEAT_max	Busy time for Set Features and Get Features
tIR_min	Output hi-Z to RE# low
tITC_max	Interface and Timing Mode Change time
tRC_min	RE# cycle time
tREA_max	RE# access time
tREH_min	RE# high hold time
tRHOH_min	RE# high to output hold
tRHW_min	RE# high to WE# low
tRHZ_max	RE# high to output hi-Z
tRLOH_min	RE# low to output hold
tRP_min	RE# pulse width
tRR_min	Ready to RE# low (data only)
tRST_max	Device reset time, measured from the falling edge of R/B# to the rising edge of R/B#.
tWB_max	WE# high to SR[6] low
tWC_min	WE# cycle time
tWH_min	WE# high hold time
tWHR_min	WE# high to RE# low
tWP_min	WE# pulse width
tWW_min	WP# transition to WE# low

## Description

This struct defines the timing requirements of a SDR NAND chip. These information can be found in every NAND datasheets and the timings meaning are described in the ONFI specifications: [www.onfi.org/~media/ONFI/specs/onfi\\_3\\_1\\_spec.pdf](http://www.onfi.org/~media/ONFI/specs/onfi_3_1_spec.pdf) (chapter 4.15 Timing Parameters)

All these timings are expressed in picoseconds.

## Name

enum nand\_data\_interface\_type — NAND interface timing type

## Synopsis

```
enum nand_data_interface_type {  
    NAND_SDR_IFACE  
};
```

## Constants

NAND\_SDR\_IFACE Single Data Rate interface

## Name

struct nand\_data\_interface — NAND interface timing

## Synopsis

```
struct nand_data_interface {  
    enum nand_data_interface_type type;  
    union timings;  
};
```

## Members

type	type of the timing
timings	The timing, type according to <i>type</i>

## Name

`nand_get_sdr_timings` — get SDR timing from data interface

## Synopsis

```
const struct nand_sdr_timings * nand_get_sdr_timings (const struct  
nand_data_interface * conf);
```

## Arguments

*conf* The data interface

## Name

struct nand\_chip — NAND Private Flash Chip Data

## Synopsis

```
struct nand_chip {
    struct mtd_info mtd;
    void __iomem * IO_ADDR_R;
    void __iomem * IO_ADDR_W;
    uint8_t (* read_byte) (struct mtd_info *mtd);
    ul6 (* read_word) (struct mtd_info *mtd);
    void (* write_byte) (struct mtd_info *mtd, uint8_t byte);
    void (* write_buf) (struct mtd_info *mtd, const uint8_t *buf, int len);
    void (* read_buf) (struct mtd_info *mtd, uint8_t *buf, int len);
    void (* select_chip) (struct mtd_info *mtd, int chip);
    int (* block_bad) (struct mtd_info *mtd, loff_t ofs);
    int (* block_markbad) (struct mtd_info *mtd, loff_t ofs);
    void (* cmd_ctrl) (struct mtd_info *mtd, int dat, unsigned int ctrl);
    int (* dev_ready) (struct mtd_info *mtd);
    void (* cmdfunc) (struct mtd_info *mtd, unsigned command, int column,int page_ad
    int(* waitfunc) (struct mtd_info *mtd, struct nand_chip *this);
    int (* erase) (struct mtd_info *mtd, int page);
    int (* scan_bbt) (struct mtd_info *mtd);
    int (* errstat) (struct mtd_info *mtd, struct nand_chip *this, int state,int sta
    int (* write_page) (struct mtd_info *mtd, struct nand_chip *chip,uint32_t offset
    int (* onfi_set_features) (struct mtd_info *mtd, struct nand_chip *chip,int feat
    int (* onfi_get_features) (struct mtd_info *mtd, struct nand_chip *chip,int feat
    int (* setup_read_retry) (struct mtd_info *mtd, int retry_mode);
    int (* setup_data_interface) (struct mtd_info *mtd,const struct nand_data_interf
    int chip_delay;
    unsigned int options;
    unsigned int bbt_options;
    int page_shift;
    int phys_erase_shift;
    int bbt_erase_shift;
    int chip_shift;
    int numchips;
    uint64_t chipsize;
    int pagemask;
    int pagebuf;
    unsigned int pagebuf_bitflips;
    int subpagesize;
    uint8_t bits_per_cell;
    uint16_t ecc_strength_ds;
    uint16_t ecc_step_ds;
    int onfi_timing_mode_default;
    int badblockpos;
    int badblockbits;
    int onfi_version;
    int jedec_version;
    union {unnamed_union};
    int read_retries;
```



```
flstate_t state;
uint8_t * oob_poi;
struct nand_hw_control * controller;
struct nand_ecc_ctrl ecc;
struct nand_buffers * buffers;
struct nand_hw_control hwcontrol;
uint8_t * bbt;
struct nand_bbt_descr * bbt_td;
struct nand_bbt_descr * bbt_md;
struct nand_bbt_descr * badblock_pattern;
void * priv;
};
```

## Members

mtd	MTD device registered to the MTD framework
IO_ADDR_R	[BOARDSPECIFIC] address to read the 8 I/O lines of the flash device
IO_ADDR_W	[BOARDSPECIFIC] address to write the 8 I/O lines of the flash device.
read_byte	[REPLACEABLE] read one byte from the chip
read_word	[REPLACEABLE] read one word from the chip
write_byte	[REPLACEABLE] write a single byte to the chip on the low 8 I/O lines
write_buf	[REPLACEABLE] write data from the buffer to the chip
read_buf	[REPLACEABLE] read data from the chip into the buffer
select_chip	[REPLACEABLE] select chip nr
block_bad	[REPLACEABLE] check if a block is bad, using OOB markers
block_markbad	[REPLACEABLE] mark a block bad
cmd_ctrl	[BOARDSPECIFIC] hardware specific function for controlling ALE/CLE/nCE. Also used to write command and address
dev_ready	[BOARDSPECIFIC] hardware specific function for accessing device ready/busy line. If set to NULL no access to ready/busy is available and the ready/busy information is read from the chip status register.
cmdfunc	[REPLACEABLE] hardware specific function for writing commands to the chip.
waitfunc	[REPLACEABLE] hardware specific function for wait on ready.
erase	[REPLACEABLE] erase function
scan_bbt	[REPLACEABLE] function to scan bad block table

errstat	[OPTIONAL] hardware specific function to perform additional error status checks (determine if errors are correctable).
write_page	[REPLACEABLE] High-level page write function
onfi_set_features	[REPLACEABLE] set the features for ONFI nand
onfi_get_features	[REPLACEABLE] get the features for ONFI nand
setup_read_retry	[FLASHSPECIFIC] flash (vendor) specific function for setting the read-retry mode. Mostly needed for MLC NAND.
setup_data_interface	[OPTIONAL] setup the data interface and timing
chip_delay	[BOARDSPECIFIC] chip dependent delay for transferring data from array to read regs (tR).
options	[BOARDSPECIFIC] various chip options. They can partly be set to inform nand_scan about special functionality. See the defines for further explanation.
bbt_options	[INTERN] bad block specific options. All options used here must come from bbm.h. By default, these options will be copied to the appropriate nand_bbt_descr's.
page_shift	[INTERN] number of address bits in a page (column address bits).
phys_erase_shift	[INTERN] number of address bits in a physical eraseblock
bbt_erase_shift	[INTERN] number of address bits in a bbt entry
chip_shift	[INTERN] number of address bits in one chip
numchips	[INTERN] number of physical chips
chipsize	[INTERN] the size of one chip for multichip arrays
pagemask	[INTERN] page number mask = number of (pages / chip) - 1
pagebuf	[INTERN] holds the pagenumber which is currently in data_buf.
pagebuf_bitflips	[INTERN] holds the bitflip count for the page which is currently in data_buf.
subpagesize	[INTERN] holds the subpagesize
bits_per_cell	[INTERN] number of bits per cell. i.e., 1 means SLC.
ecc_strength_ds	[INTERN] ECC correctability from the datasheet. Minimum amount of bit errors per <i>ecc_step_ds</i> guaranteed to be correctable. If unknown, set to zero.
ecc_step_ds	[INTERN] ECC step required by the <i>ecc_strength_ds</i> , also from the datasheet. It is the recommended ECC step size, if known; if unknown, set to zero.
onfi_timing_mode_default	[INTERN] default ONFI timing mode. This field is set to the actually used ONFI mode if the chip is ONFI compliant or deduced from the datasheet if the NAND chip is not ONFI compliant.

badblockpos	[INTERN] position of the bad block marker in the oob area.
badblockbits	[INTERN] minimum number of set bits in a good block's bad block marker position; i.e., BBM == 11110111b is not bad when badblockbits == 7
onfi_version	[INTERN] holds the chip ONFI version (BCD encoded), non 0 if ONFI supported.
jedec_version	[INTERN] holds the chip JEDEC version (BCD encoded), non 0 if JEDEC supported.
{unnamed_union}	anonymous
read_retries	[INTERN] the number of read retry modes supported
state	[INTERN] the current state of the NAND device
oob_poi	"poison value buffer," used for laying out OOB data before writing
controller	[REPLACEABLE] a pointer to a hardware controller structure which is shared among multiple independent devices.
ecc	[BOARDSPECIFIC] ECC control structure
buffers	buffer structure for read/write
hwcontrol	platform-specific hardware control structure
bbt	[INTERN] bad block table pointer
bbt_td	[REPLACEABLE] bad block table descriptor for flash lookup.
bbt_md	[REPLACEABLE] bad block table mirror descriptor
badblock_pattern	[REPLACEABLE] bad block scan pattern used for initial bad block scan.
priv	[OPTIONAL] pointer to private chip data

## Name

struct nand\_flash\_dev — NAND Flash Device ID Structure

## Synopsis

```
struct nand_flash_dev {  
    char * name;  
    union ecc;  
    int onfi_timing_mode_default;  
};
```

## Members

name	a human-readable name of the NAND chip
ecc	The ECC step required by the <i>ecc.strength_ds</i> , same as the <i>ecc_step_ds</i> in <i>nand_chip{}</i> , also from the datasheet. For example, the “4bit ECC for each 512Byte” can be set with <i>NAND_ECC_INFO(4, 512)</i> .
onfi_timing_mode_default	the default ONFI timing mode entered after a NAND reset. Should be deduced from timings described in the datasheet.

## Name

struct nand\_manufacturers — NAND Flash Manufacturer ID Structure

## Synopsis

```
struct nand_manufacturers {  
    int id;  
    char * name;  
};
```

## Members

id      manufacturer ID code of device.  
name    Manufacturer name

## Name

struct platform\_nand\_chip — chip level device structure

## Synopsis

```
struct platform_nand_chip {
    int nr_chips;
    int chip_offset;
    int nr_partitions;
    struct mtd_partition * partitions;
    int chip_delay;
    unsigned int options;
    unsigned int bbt_options;
    const char ** part_probe_types;
};
```

## Members

nr_chips	max. number of chips to scan for
chip_offset	chip number offset
nr_partitions	number of partitions pointed to by partitions (or zero)
partitions	mtd partition list
chip_delay	R/B delay value in us
options	Option flags, e.g. 16bit buswidth
bbt_options	BBT option flags, e.g. NAND_BBT_USE_FLASH
part_probe_types	NULL-terminated array of probe types

## Name

struct platform\_nand\_ctrl — controller level device structure

## Synopsis

```
struct platform_nand_ctrl {
    int (* probe) (struct platform_device *pdev);
    void (* remove) (struct platform_device *pdev);
    void (* hwcontrol) (struct mtd_info *mtd, int cmd);
    int (* dev_ready) (struct mtd_info *mtd);
    void (* select_chip) (struct mtd_info *mtd, int chip);
    void (* cmd_ctrl) (struct mtd_info *mtd, int dat, unsigned int ctrl);
    void (* write_buf) (struct mtd_info *mtd, const uint8_t *buf, int len);
    void (* read_buf) (struct mtd_info *mtd, uint8_t *buf, int len);
    unsigned char (* read_byte) (struct mtd_info *mtd);
    void * priv;
};
```

## Members

probe	platform specific function to probe/setup hardware
remove	platform specific function to remove/teardown hardware
hwcontrol	platform specific hardware control structure
dev_ready	platform specific function to read ready/busy pin
select_chip	platform specific chip select function
cmd_ctrl	platform specific function for controlling ALE/CLE/nCE. Also used to write command and address
write_buf	platform specific function for write buffer
read_buf	platform specific function for read buffer
read_byte	platform specific function to read one byte from chip
priv	private data to transport driver specific settings

## Description

All fields are optional and depend on the hardware driver requirements

## Name

struct platform\_nand\_data — container structure for platform-specific data

## Synopsis

```
struct platform_nand_data {  
    struct platform_nand_chip chip;  
    struct platform_nand_ctrl ctrl;  
};
```

## Members

chip	chip level chip structure
ctrl	controller level device structure



## Name

nand\_opcode\_8bits —

## Synopsis

```
int nand_opcode_8bits (unsigned int command);
```

## Arguments

*command* opcode to check

---

# Chapter 10. 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.

## Name

`nand_wait_ready` — [GENERIC] Wait for the ready pin after commands.

## Synopsis

```
void nand_wait_ready (struct mtd_info * mtd);
```

## Arguments

*mtd* MTD device structure

## Description

Wait for the ready pin after a command, and warn if a timeout occurs.

## Name

`nand_unlock` — [REPLACEABLE] unlocks specified locked blocks

## Synopsis

```
int nand_unlock (struct mtd_info * mtd, loff_t ofs, uint64_t len);
```

## Arguments

*mtd* mtd info

*ofs* offset to start unlock from

*len* length to unlock

## Description

Returns unlock status.

## Name

`nand_lock` — [REPLACEABLE] locks all blocks present in the device

## Synopsis

```
int nand_lock (struct mtd_info * mtd, loff_t ofs, uint64_t len);
```

## Arguments

*mtd* mtd info

*ofs* offset to start unlock from

*len* length to unlock

## Description

This feature is not supported in many NAND parts. 'Micron' NAND parts do have this feature, but it allows only to lock all blocks, not for specified range for block. Implementing 'lock' feature by making use of 'unlock', for now.

Returns lock status.

## Name

`nand_check_erased_ecc_chunk` — check if an ECC chunk contains (almost) only 0xff data

## Synopsis

```
int nand_check_erased_ecc_chunk (void * data, int datalen, void * ecc,  
int ecclen, void * extraoob, int extraooblen, int bitflips_threshold);
```

## Arguments

<i>data</i>	data buffer to test
<i>datalen</i>	data length
<i>ecc</i>	ECC buffer
<i>ecclen</i>	ECC length
<i>extraoob</i>	extra OOB buffer
<i>extraooblen</i>	extra OOB length
<i>bitflips_threshold</i>	maximum number of bitflips

## Description

Check if a data buffer and its associated ECC and OOB data contains only 0xff pattern, which means the underlying region has been erased and is ready to be programmed. The `bitflips_threshold` specify the maximum number of bitflips before considering the region as not erased.

## Note

1/ ECC algorithms are working on pre-defined block sizes which are usually different from the NAND page size. When fixing bitflips, ECC engines will report the number of errors per chunk, and the NAND core infrastructure expect you to return the maximum number of bitflips for the whole page. This is why you should always use this function on a single chunk and not on the whole page. After checking each chunk you should update your `max_bitflips` value accordingly. 2/ When checking for bitflips in erased pages you should not only check the payload data but also their associated ECC data, because a user might have programmed almost all bits to 1 but a few. In this case, we shouldn't consider the chunk as erased, and checking ECC bytes prevent this case. 3/ The `extraoob` argument is optional, and should be used if some of your OOB data are protected by the ECC engine. It could also be used if you support subpages and want to attach some extra OOB data to an ECC chunk.

Returns a positive number of bitflips less than or equal to `bitflips_threshold`, or `-ERROR_CODE` for bitflips in excess of the threshold. In case of success, the passed buffers are filled with 0xff.

## Name

`nand_read_oob_std` — [REPLACEABLE] the most common OOB data read function

## Synopsis

```
int nand_read_oob_std (struct mtd_info * mtd, struct nand_chip * chip,  
int page);
```

## Arguments

*mtd* mtd info structure

*chip* nand chip info structure

*page* page number to read

## Name

`nand_read_oob_syndrome` — [REPLACEABLE] OOB data read function for HW ECC with syndromes

## Synopsis

```
int nand_read_oob_syndrome (struct mtd_info * mtd, struct nand_chip *  
chip, int page);
```

## Arguments

*mtd* mtd info structure

*chip* nand chip info structure

*page* page number to read



## Name

`nand_write_oob_std` — [REPLACEABLE] the most common OOB data write function

## Synopsis

```
int nand_write_oob_std (struct mtd_info * mtd, struct nand_chip * chip,  
int page);
```

## Arguments

*mtd*    mtd info structure

*chip*   nand chip info structure

*page*   page number to write

## Name

`nand_write_oob_syndrome` — [REPLACEABLE] OOB data write function for HW ECC with syndrome  
- only for large page flash

## Synopsis

```
int nand_write_oob_syndrome (struct mtd_info * mtd, struct nand_chip  
* chip, int page);
```

## Arguments

*mtd* mtd info structure

*chip* nand chip info structure

*page* page number to write

## Name

`nand_scan_ident` — [NAND Interface] Scan for the NAND device

## Synopsis

```
int nand_scan_ident (struct mtd_info * mtd, int maxchips, struct  
nand_flash_dev * table);
```

## Arguments

<i>mtd</i>	MTD device structure
<i>maxchips</i>	number of chips to scan for
<i>table</i>	alternative NAND ID table

## Description

This is the first phase of the normal `nand_scan` function. It reads the flash ID and sets up MTD fields accordingly.

## Name

`nand_scan_tail` — [NAND Interface] Scan for the NAND device

## Synopsis

```
int nand_scan_tail (struct mtd_info * mtd);
```

## Arguments

*mtd* MTD device structure

## Description

This is the second phase of the normal `nand_scan` function. It fills out all the uninitialized function pointers with the defaults and scans for a bad block table if appropriate.

## Name

`nand_scan` — [NAND Interface] Scan for the NAND device

## Synopsis

```
int nand_scan (struct mtd_info * mtd, int maxchips);
```

## Arguments

*mtd*            MTD device structure

*maxchips*    number of chips to scan for

## Description

This fills out all the uninitialized function pointers with the defaults. The flash ID is read and the mtd/chip structures are filled with the appropriate values.

## Name

`nand_cleanup` — [NAND Interface] Free resources held by the NAND device

## Synopsis

```
void nand_cleanup (struct nand_chip * chip);
```

## Arguments

*chip* NAND chip object

## Name

`nand_release` — [NAND Interface] Unregister the MTD device and free resources held by the NAND device

## Synopsis

```
void nand_release (struct mtd_info * mtd);
```

## Arguments

*mtd* MTD device structure

## Name

drivers/mtd/nand/nand\_bbt.c — Document generation inconsistency

## Oops

### Warning

The template for this document tried to insert the structured comment from the file `drivers/mtd/nand/nand_bbt.c` at this point, but none was found. This dummy section is inserted to allow generation to continue.



## Name

`__nand_calculate_ecc` — [NAND Interface] Calculate 3-byte ECC for 256/512-byte block

## Synopsis

```
void __nand_calculate_ecc (const unsigned char * buf, unsigned int  
eccsize, unsigned char * code);
```

## Arguments

<i>buf</i>	input buffer with raw data
<i>eccsize</i>	data bytes per ECC step (256 or 512)
<i>code</i>	output buffer with ECC

## Name

`nand_calculate_ecc` — [NAND Interface] Calculate 3-byte ECC for 256/512-byte block

## Synopsis

```
int nand_calculate_ecc (struct mtd_info * mtd, const unsigned char *  
buf, unsigned char * code);
```

## Arguments

*mtd*    MTD block structure

*buf*    input buffer with raw data

*code*   output buffer with ECC

## Name

`__nand_correct_data` — [NAND Interface] Detect and correct bit error(s)

## Synopsis

```
int __nand_correct_data (unsigned char * buf, unsigned char * read_ecc,  
unsigned char * calc_ecc, unsigned int eccsize);
```

## Arguments

<i>buf</i>	raw data read from the chip
<i>read_ecc</i>	ECC from the chip
<i>calc_ecc</i>	the ECC calculated from raw data
<i>eccsize</i>	data bytes per ECC step (256 or 512)

## Description

Detect and correct a 1 bit error for *eccsize* byte block

## Name

nand\_correct\_data — [NAND Interface] Detect and correct bit error(s)

## Synopsis

```
int nand_correct_data (struct mtd_info * mtd, unsigned char * buf,  
unsigned char * read_ecc, unsigned char * calc_ecc);
```

## Arguments

<i>mtd</i>	MTD block structure
<i>buf</i>	raw data read from the chip
<i>read_ecc</i>	ECC from the chip
<i>calc_ecc</i>	the ECC calculated from raw data

## Description

Detect and correct a 1 bit error for 256/512 byte block

---

# Chapter 11. 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.

## Name

nand\_release\_device — [GENERIC] release chip

## Synopsis

```
void nand_release_device (struct mtd_info * mtd);
```

## Arguments

*mtd* MTD device structure

## Description

Release chip lock and wake up anyone waiting on the device.

## Name

`nand_read_byte` — [DEFAULT] read one byte from the chip

## Synopsis

```
uint8_t nand_read_byte (struct mtd_info * mtd);
```

## Arguments

*mtd* MTD device structure

## Description

Default read function for 8bit buswidth

## Name

`nand_read_byte16` — [DEFAULT] read one byte endianness aware from the chip

## Synopsis

```
uint8_t nand_read_byte16 (struct mtd_info * mtd);
```

## Arguments

*mtd* MTD device structure

## Description

Default read function for 16bit buswidth with endianness conversion.



## Name

`nand_read_word` — [DEFAULT] read one word from the chip

## Synopsis

```
u16 nand_read_word (struct mtd_info * mtd);
```

## Arguments

*mtd* MTD device structure

## Description

Default read function for 16bit buswidth without endianness conversion.

## Name

nand\_select\_chip — [DEFAULT] control CE line

## Synopsis

```
void nand_select_chip (struct mtd_info * mtd, int chipnr);
```

## Arguments

*mtd*        MTD device structure

*chipnr*    chipnumber to select, -1 for deselect

## Description

Default select function for 1 chip devices.

## Name

`nand_write_byte` — [DEFAULT] write single byte to chip

## Synopsis

```
void nand_write_byte (struct mtd_info * mtd, uint8_t byte);
```

## Arguments

*mtd*     MTD device structure

*byte*    value to write

## Description

Default function to write a byte to I/O[7:0]

## Name

`nand_write_byte16` — [DEFAULT] write single byte to a chip with width 16

## Synopsis

```
void nand_write_byte16 (struct mtd_info * mtd, uint8_t byte);
```

## Arguments

*mtd*    MTD device structure

*byte*   value to write

## Description

Default function to write a byte to I/O[7:0] on a 16-bit wide chip.

## Name

`nand_write_buf` — [DEFAULT] write buffer to chip

## Synopsis

```
void nand_write_buf (struct mtd_info * mtd, const uint8_t * buf, int  
len);
```

## Arguments

*mtd* MTD device structure

*buf* data buffer

*len* number of bytes to write

## Description

Default write function for 8bit buswidth.

## Name

`nand_read_buf` — [DEFAULT] read chip data into buffer

## Synopsis

```
void nand_read_buf (struct mtd_info * mtd, uint8_t * buf, int len);
```

## Arguments

*mtd* MTD device structure

*buf* buffer to store data

*len* number of bytes to read

## Description

Default read function for 8bit buswidth.

## Name

`nand_write_buf16` — [DEFAULT] write buffer to chip

## Synopsis

```
void nand_write_buf16 (struct mtd_info * mtd, const uint8_t * buf, int  
len);
```

## Arguments

*mtd* MTD device structure

*buf* data buffer

*len* number of bytes to write

## Description

Default write function for 16bit buswidth.

## Name

`nand_read_buf16` — [DEFAULT] read chip data into buffer

## Synopsis

```
void nand_read_buf16 (struct mtd_info * mtd, uint8_t * buf, int len);
```

## Arguments

*mtd* MTD device structure

*buf* buffer to store data

*len* number of bytes to read

## Description

Default read function for 16bit buswidth.



## Name

`nand_block_bad` — [DEFAULT] Read bad block marker from the chip

## Synopsis

```
int nand_block_bad (struct mtd_info * mtd, loff_t ofs);
```

## Arguments

*mtd* MTD device structure

*ofs* offset from device start

## Description

Check, if the block is bad.

## Name

`nand_default_block_markbad` — [DEFAULT] mark a block bad via bad block marker

## Synopsis

```
int nand_default_block_markbad (struct mtd_info * mtd, loff_t ofs);
```

## Arguments

*mtd* MTD device structure

*ofs* offset from device start

## Description

This is the default implementation, which can be overridden by a hardware specific driver. It provides the details for writing a bad block marker to a block.

## Name

`nand_block_markbad_lowlevel` — mark a block bad

## Synopsis

```
int nand_block_markbad_lowlevel (struct mtd_info * mtd, loff_t ofs);
```

## Arguments

*mtd* MTD device structure

*ofs* offset from device start

## Description

This function performs the generic NAND bad block marking steps (i.e., bad block table(s) and/or marker(s)). We only allow the hardware driver to specify how to write bad block markers to OOB (chip->block\_markbad).

We try operations in the following order: (1) erase the affected block, to allow OOB marker to be written cleanly (2) write bad block marker to OOB area of affected block (unless flag NAND\_BBT\_NO\_OOB\_BBM is present) (3) update the BBT Note that we retain the first error encountered in (2) or (3), finish the procedures, and dump the error in the end.

## Name

nand\_check\_wp — [GENERIC] check if the chip is write protected

## Synopsis

```
int nand_check_wp (struct mtd_info * mtd);
```

## Arguments

*mtd* MTD device structure

## Description

Check, if the device is write protected. The function expects, that the device is already selected.

## Name

`nand_block_isreserved` — [GENERIC] Check if a block is marked reserved.

## Synopsis

```
int nand_block_isreserved (struct mtd_info * mtd, loff_t ofs);
```

## Arguments

*mtd* MTD device structure

*ofs* offset from device start

## Description

Check if the block is marked as reserved.

## Name

`nand_block_checkbad` — [GENERIC] Check if a block is marked bad

## Synopsis

```
int nand_block_checkbad (struct mtd_info * mtd, loff_t ofs, int
allowbbt);
```

## Arguments

<i>mtd</i>	MTD device structure
<i>ofs</i>	offset from device start
<i>allowbbt</i>	1, if its allowed to access the bbt area

## Description

Check, if the block is bad. Either by reading the bad block table or calling of the scan function.

## Name

`panic_nand_wait_ready` — [GENERIC] Wait for the ready pin after commands.

## Synopsis

```
void panic_nand_wait_ready (struct mtd_info * mtd, unsigned long timeo);
```

## Arguments

*mtd*     MTD device structure

*timeo*   Timeout

## Description

Helper function for `nand_wait_ready` used when needing to wait in interrupt context.

## Name

`nand_wait_status_ready` — [GENERIC] Wait for the ready status after commands.

## Synopsis

```
void nand_wait_status_ready (struct mtd_info * mtd, unsigned long  
                             timeo);
```

## Arguments

*mtd*     MTD device structure

*timeo*   Timeout in ms

## Description

Wait for status ready (i.e. command done) or timeout.



## Name

`nand_command` — [DEFAULT] Send command to NAND device

## Synopsis

```
void nand_command (struct mtd_info * mtd, unsigned int command, int  
column, int page_addr);
```

## Arguments

<i>mtd</i>	MTD device structure
<i>command</i>	the command to be sent
<i>column</i>	the column address for this command, -1 if none
<i>page_addr</i>	the page address for this command, -1 if none

## Description

Send command to NAND device. This function is used for small page devices (512 Bytes per page).

## Name

`nand_command_lp` — [DEFAULT] Send command to NAND large page device

## Synopsis

```
void nand_command_lp (struct mtd_info * mtd, unsigned int command, int  
column, int page_addr);
```

## Arguments

<i>mtd</i>	MTD device structure
<i>command</i>	the command to be sent
<i>column</i>	the column address for this command, -1 if none
<i>page_addr</i>	the page address for this command, -1 if none

## Description

Send command to NAND device. This is the version for the new large page devices. We don't have the separate regions as we have in the small page devices. We must emulate NAND\_CMD\_READOOB to keep the code compatible.

## Name

`panic_nand_get_device` — [GENERIC] Get chip for selected access

## Synopsis

```
void panic_nand_get_device (struct nand_chip * chip, struct mtd_info *  
mtd, int new_state);
```

## Arguments

<i>chip</i>	the nand chip descriptor
<i>mtd</i>	MTD device structure
<i>new_state</i>	the state which is requested

## Description

Used when in panic, no locks are taken.

## Name

nand\_get\_device — [GENERIC] Get chip for selected access

## Synopsis

```
int nand_get_device (struct mtd_info * mtd, int new_state);
```

## Arguments

*mtd*            MTD device structure

*new\_state*    the state which is requested

## Description

Get the device and lock it for exclusive access

## Name

`panic_nand_wait` — [GENERIC] wait until the command is done

## Synopsis

```
void panic_nand_wait (struct mtd_info * mtd, struct nand_chip * chip,  
unsigned long timeo);
```

## Arguments

*mtd*     MTD device structure

*chip*    NAND chip structure

*timeo*   timeout

## Description

Wait for command done. This is a helper function for `nand_wait` used when we are in interrupt context. May happen when in panic and trying to write an oops through `mtdots`.

## Name

`nand_wait` — [DEFAULT] wait until the command is done

## Synopsis

```
int nand_wait (struct mtd_info * mtd, struct nand_chip * chip);
```

## Arguments

*mtd*    MTD device structure

*chip*   NAND chip structure

## Description

Wait for command done. This applies to erase and program only.

## Name

`nand_reset_data_interface` — Reset data interface and timings

## Synopsis

```
int nand_reset_data_interface (struct nand_chip * chip);
```

## Arguments

*chip*    The NAND chip

## Description

Reset the Data interface and timings to ONFI mode 0.

Returns 0 for success or negative error code otherwise.

## Name

`nand_setup_data_interface` — Setup the best data interface and timings

## Synopsis

```
int nand_setup_data_interface (struct nand_chip * chip);
```

## Arguments

*chip* The NAND chip

## Description

Find and configure the best data interface and NAND timings supported by the chip and the driver. First tries to retrieve supported timing modes from ONFI information, and if the NAND chip does not support ONFI, relies on the `->onfi_timing_mode_default` specified in the `nand_ids` table.

Returns 0 for success or negative error code otherwise.



## Name

`nand_init_data_interface` — find the best data interface and timings

## Synopsis

```
int nand_init_data_interface (struct nand_chip * chip);
```

## Arguments

*chip*    The NAND chip

## Description

Find the best data interface and NAND timings supported by the chip and the driver. First tries to retrieve supported timing modes from ONFI information, and if the NAND chip does not support ONFI, relies on the `->onfi_timing_mode_default` specified in the `nand_ids` table. After this function `nand_chip->data_interface` is initialized with the best timing mode available.

Returns 0 for success or negative error code otherwise.

## Name

`nand_reset` — Reset and initialize a NAND device

## Synopsis

```
int nand_reset (struct nand_chip * chip, int chipnr);
```

## Arguments

*chip*      The NAND chip

*chipnr*    Internal die id

## Description

Returns 0 for success or negative error code otherwise

## Name

`__nand_unlock` — [REPLACEABLE] unlocks specified locked blocks

## Synopsis

```
int __nand_unlock (struct mtd_info * mtd, loff_t ofs, uint64_t len,  
int invert);
```

## Arguments

*mtd*      mtd info

*ofs*      offset to start unlock from

*len*      length to unlock

*invert*   when = 0, unlock the range of blocks within the lower and upper boundary address when = 1,  
unlock the range of blocks outside the boundaries of the lower and upper boundary address

## Description

Returns unlock status.

## Name

`nand_check_erased_buf` — check if a buffer contains (almost) only 0xff data

## Synopsis

```
int nand_check_erased_buf (void * buf, int len, int bitflips_threshold);
```

## Arguments

<i>buf</i>	buffer to test
<i>len</i>	buffer length
<i>bitflips_threshold</i>	maximum number of bitflips

## Description

Check if a buffer contains only 0xff, which means the underlying region has been erased and is ready to be programmed. The `bitflips_threshold` specify the maximum number of bitflips before considering the region is not erased.

## Note

The logic of this function has been extracted from the `memweight` implementation, except that `nand_check_erased_buf` function exit before testing the whole buffer if the number of bitflips exceed the `bitflips_threshold` value.

Returns a positive number of bitflips less than or equal to `bitflips_threshold`, or `-ERROR_CODE` for bitflips in excess of the threshold.

## Name

`nand_read_page_raw` — [INTERN] read raw page data without ecc

## Synopsis

```
int nand_read_page_raw (struct mtd_info * mtd, struct nand_chip * chip,  
uint8_t * buf, int oob_required, int page);
```

## Arguments

<i>mtd</i>	mtd info structure
<i>chip</i>	nand chip info structure
<i>buf</i>	buffer to store read data
<i>oob_required</i>	caller requires OOB data read to chip->oob_poi
<i>page</i>	page number to read

## Description

Not for syndrome calculating ECC controllers, which use a special oob layout.

## Name

`nand_read_page_raw_syndrome` — [INTERN] read raw page data without ecc

## Synopsis

```
int nand_read_page_raw_syndrome (struct mtd_info * mtd, struct nand_chip  
* chip, uint8_t * buf, int oob_required, int page);
```

## Arguments

<i>mtd</i>	mtd info structure
<i>chip</i>	nand chip info structure
<i>buf</i>	buffer to store read data
<i>oob_required</i>	caller requires OOB data read to chip->oob_poi
<i>page</i>	page number to read

## Description

We need a special oob layout and handling even when OOB isn't used.

## Name

`nand_read_page_swecc` — [REPLACEABLE] software ECC based page read function

## Synopsis

```
int nand_read_page_swecc (struct mtd_info * mtd, struct nand_chip *  
chip, uint8_t * buf, int oob_required, int page);
```

## Arguments

<i>mtd</i>	mtd info structure
<i>chip</i>	nand chip info structure
<i>buf</i>	buffer to store read data
<i>oob_required</i>	caller requires OOB data read to chip->oob_poi
<i>page</i>	page number to read

## Name

nand\_read\_subpage — [REPLACEABLE] ECC based sub-page read function

## Synopsis

```
int nand_read_subpage (struct mtd_info * mtd, struct nand_chip * chip,  
uint32_t data_offs, uint32_t readlen, uint8_t * bufpoi, int page);
```

## Arguments

<i>mtd</i>	mtd info structure
<i>chip</i>	nand chip info structure
<i>data_offs</i>	offset of requested data within the page
<i>readlen</i>	data length
<i>bufpoi</i>	buffer to store read data
<i>page</i>	page number to read



## Name

`nand_read_page_hwecc` — [REPLACEABLE] hardware ECC based page read function

## Synopsis

```
int nand_read_page_hwecc (struct mtd_info * mtd, struct nand_chip *  
chip, uint8_t * buf, int oob_required, int page);
```

## Arguments

<i>mtd</i>	mtd info structure
<i>chip</i>	nand chip info structure
<i>buf</i>	buffer to store read data
<i>oob_required</i>	caller requires OOB data read to chip->oob_poi
<i>page</i>	page number to read

## Description

Not for syndrome calculating ECC controllers which need a special oob layout.

## Name

`nand_read_page_hwecc_oob_first` — [REPLACEABLE] hw ecc, read oob first

## Synopsis

```
int nand_read_page_hwecc_oob_first (struct mtd_info * mtd, struct
nand_chip * chip, uint8_t * buf, int oob_required, int page);
```

## Arguments

<i>mtd</i>	mtd info structure
<i>chip</i>	nand chip info structure
<i>buf</i>	buffer to store read data
<i>oob_required</i>	caller requires OOB data read to chip->oob_poi
<i>page</i>	page number to read

## Description

Hardware ECC for large page chips, require OOB to be read first. For this ECC mode, the `write_page` method is re-used from `ECC_HW`. These methods read/write ECC from the OOB area, unlike the `ECC_HW_SYNDROME` support with multiple ECC steps, follows the “infix ECC” scheme and reads/writes ECC from the data area, by overwriting the NAND manufacturer bad block markings.

## Name

`nand_read_page_syndrome` — [REPLACEABLE] hardware ECC syndrome based page read

## Synopsis

```
int nand_read_page_syndrome (struct mtd_info * mtd, struct nand_chip *  
chip, uint8_t * buf, int oob_required, int page);
```

## Arguments

<i>mtd</i>	mtd info structure
<i>chip</i>	nand chip info structure
<i>buf</i>	buffer to store read data
<i>oob_required</i>	caller requires OOB data read to chip->oob_poi
<i>page</i>	page number to read

## Description

The hw generator calculates the error syndrome automatically. Therefore we need a special oob layout and handling.

## Name

`nand_transfer_oob` — [INTERN] Transfer oob to client buffer

## Synopsis

```
uint8_t * nand_transfer_oob (struct mtd_info * mtd, uint8_t * oob,  
struct mtd_oob_ops * ops, size_t len);
```

## Arguments

*mtd* mtd info structure

*oob* oob destination address

*ops* oob ops structure

*len* size of oob to transfer

## Name

`nand_setup_read_retry` — [INTERN] Set the READ RETRY mode

## Synopsis

```
int nand_setup_read_retry (struct mtd_info * mtd, int retry_mode);
```

## Arguments

*mtd*                    MTD device structure

*retry\_mode*    the retry mode to use

## Description

Some vendors supply a special command to shift the Vt threshold, to be used when there are too many bitflips in a page (i.e., ECC error). After setting a new threshold, the host should retry reading the page.

## Name

nand\_do\_read\_ops — [INTERN] Read data with ECC

## Synopsis

```
int nand_do_read_ops (struct mtd_info * mtd, loff_t from, struct  
mtd_oob_ops * ops);
```

## Arguments

*mtd*    MTD device structure

*from*   offset to read from

*ops*    oob ops structure

## Description

Internal function. Called with chip held.

## Name

`nand_read` — [MTD Interface] MTD compatibility function for `nand_do_read_ecc`

## Synopsis

```
int nand_read (struct mtd_info * mtd, loff_t from, size_t len, size_t  
* retlen, uint8_t * buf);
```

## Arguments

<i>mtd</i>	MTD device structure
<i>from</i>	offset to read from
<i>len</i>	number of bytes to read
<i>retlen</i>	pointer to variable to store the number of read bytes
<i>buf</i>	the databuffer to put data

## Description

Get hold of the chip and call `nand_do_read`.

## Name

nand\_do\_read\_oob — [INTERN] NAND read out-of-band

## Synopsis

```
int nand_do_read_oob (struct mtd_info * mtd, loff_t from, struct
mtd_oob_ops * ops);
```

## Arguments

*mtd* MTD device structure

*from* offset to read from

*ops* oob operations description structure

## Description

NAND read out-of-band data from the spare area.



## Name

nand\_read\_oob — [MTD Interface] NAND read data and/or out-of-band

## Synopsis

```
int nand_read_oob (struct mtd_info * mtd, loff_t from, struct mtd_oob_ops  
* ops);
```

## Arguments

*mtd*    MTD device structure

*from*   offset to read from

*ops*    oob operation description structure

## Description

NAND read data and/or out-of-band data.

## Name

`nand_write_page_raw` — [INTERN] raw page write function

## Synopsis

```
int nand_write_page_raw (struct mtd_info * mtd, struct nand_chip * chip,  
const uint8_t * buf, int oob_required, int page);
```

## Arguments

<i>mtd</i>	mtd info structure
<i>chip</i>	nand chip info structure
<i>buf</i>	data buffer
<i>oob_required</i>	must write chip->oob_poi to OOB
<i>page</i>	page number to write

## Description

Not for syndrome calculating ECC controllers, which use a special oob layout.

## Name

nand\_write\_page\_raw\_syndrome — [INTERN] raw page write function

## Synopsis

```
int nand_write_page_raw_syndrome (struct mtd_info * mtd, struct
nand_chip * chip, const uint8_t * buf, int oob_required, int page);
```

## Arguments

<i>mtd</i>	mtd info structure
<i>chip</i>	nand chip info structure
<i>buf</i>	data buffer
<i>oob_required</i>	must write chip->oob_poi to OOB
<i>page</i>	page number to write

## Description

We need a special oob layout and handling even when ECC isn't checked.

## Name

`nand_write_page_swecc` — [REPLACEABLE] software ECC based page write function

## Synopsis

```
int nand_write_page_swecc (struct mtd_info * mtd, struct nand_chip *  
chip, const uint8_t * buf, int oob_required, int page);
```

## Arguments

<i>mtd</i>	mtd info structure
<i>chip</i>	nand chip info structure
<i>buf</i>	data buffer
<i>oob_required</i>	must write chip->oob_poi to OOB
<i>page</i>	page number to write

## Name

`nand_write_page_hwecc` — [REPLACEABLE] hardware ECC based page write function

## Synopsis

```
int nand_write_page_hwecc (struct mtd_info * mtd, struct nand_chip *  
chip, const uint8_t * buf, int oob_required, int page);
```

## Arguments

<i>mtd</i>	mtd info structure
<i>chip</i>	nand chip info structure
<i>buf</i>	data buffer
<i>oob_required</i>	must write chip->oob_poi to OOB
<i>page</i>	page number to write

## Name

`nand_write_subpage_hwecc` — [REPLACEABLE] hardware ECC based subpage write

## Synopsis

```
int nand_write_subpage_hwecc (struct mtd_info * mtd, struct nand_chip  
* chip, uint32_t offset, uint32_t data_len, const uint8_t * buf, int  
oob_required, int page);
```

## Arguments

<i>mtd</i>	mtd info structure
<i>chip</i>	nand chip info structure
<i>offset</i>	column address of subpage within the page
<i>data_len</i>	data length
<i>buf</i>	data buffer
<i>oob_required</i>	must write chip->oob_poi to OOB
<i>page</i>	page number to write

## Name

`nand_write_page_syndrome` — [REPLACEABLE] hardware ECC syndrome based page write

## Synopsis

```
int nand_write_page_syndrome (struct mtd_info * mtd, struct nand_chip  
* chip, const uint8_t * buf, int oob_required, int page);
```

## Arguments

<i>mtd</i>	mtd info structure
<i>chip</i>	nand chip info structure
<i>buf</i>	data buffer
<i>oob_required</i>	must write chip->oob_poi to OOB
<i>page</i>	page number to write

## Description

The hw generator calculates the error syndrome automatically. Therefore we need a special oob layout and handling.

## Name

nand\_write\_page — [REPLACEABLE] write one page

## Synopsis

```
int nand_write_page (struct mtd_info * mtd, struct nand_chip * chip,
uint32_t offset, int data_len, const uint8_t * buf, int oob_required,
int page, int cached, int raw);
```

## Arguments

<i>mtd</i>	MTD device structure
<i>chip</i>	NAND chip descriptor
<i>offset</i>	address offset within the page
<i>data_len</i>	length of actual data to be written
<i>buf</i>	the data to write
<i>oob_required</i>	must write chip->oob_poi to OOB
<i>page</i>	page number to write
<i>cached</i>	cached programming
<i>raw</i>	use _raw version of write_page



## Name

`nand_fill_oob` — [INTERN] Transfer client buffer to oob

## Synopsis

```
uint8_t * nand_fill_oob (struct mtd_info * mtd, uint8_t * oob, size_t  
len, struct mtd_oob_ops * ops);
```

## Arguments

*mtd* MTD device structure

*oob* oob data buffer

*len* oob data write length

*ops* oob ops structure

## Name

nand\_do\_write\_ops — [INTERN] NAND write with ECC

## Synopsis

```
int nand_do_write_ops (struct mtd_info * mtd, loff_t to, struct
mtd_oob_ops * ops);
```

## Arguments

*mtd* MTD device structure

*to* offset to write to

*ops* oob operations description structure

## Description

NAND write with ECC.

## Name

`panic_nand_write` — [MTD Interface] NAND write with ECC

## Synopsis

```
int panic_nand_write (struct mtd_info * mtd, loff_t to, size_t len,  
size_t * retlen, const uint8_t * buf);
```

## Arguments

<i>mtd</i>	MTD device structure
<i>to</i>	offset to write to
<i>len</i>	number of bytes to write
<i>retlen</i>	pointer to variable to store the number of written bytes
<i>buf</i>	the data to write

## Description

NAND write with ECC. Used when performing writes in interrupt context, this may for example be called by `mtdoops` when writing an oops while in panic.

## Name

nand\_write — [MTD Interface] NAND write with ECC

## Synopsis

```
int nand_write (struct mtd_info * mtd, loff_t to, size_t len, size_t  
* retlen, const uint8_t * buf);
```

## Arguments

<i>mtd</i>	MTD device structure
<i>to</i>	offset to write to
<i>len</i>	number of bytes to write
<i>retlen</i>	pointer to variable to store the number of written bytes
<i>buf</i>	the data to write

## Description

NAND write with ECC.

## Name

nand\_do\_write\_oob — [MTD Interface] NAND write out-of-band

## Synopsis

```
int nand_do_write_oob (struct mtd_info * mtd, loff_t to, struct
mtd_oob_ops * ops);
```

## Arguments

*mtd* MTD device structure

*to* offset to write to

*ops* oob operation description structure

## Description

NAND write out-of-band.

## Name

nand\_write\_oob — [MTD Interface] NAND write data and/or out-of-band

## Synopsis

```
int nand_write_oob (struct mtd_info * mtd, loff_t to, struct mtd_oob_ops  
* ops);
```

## Arguments

*mtd* MTD device structure

*to* offset to write to

*ops* oob operation description structure

## Name

`single_erase` — [GENERIC] NAND standard block erase command function

## Synopsis

```
int single_erase (struct mtd_info * mtd, int page);
```

## Arguments

*mtd*    MTD device structure

*page*   the page address of the block which will be erased

## Description

Standard erase command for NAND chips. Returns NAND status.

## Name

nand\_erase — [MTD Interface] erase block(s)

## Synopsis

```
int nand_erase (struct mtd_info * mtd, struct erase_info * instr);
```

## Arguments

*mtd*      MTD device structure

*instr*    erase instruction

## Description

Erase one ore more blocks.



## Name

`nand_erase_nand` — [INTERN] erase block(s)

## Synopsis

```
int nand_erase_nand (struct mtd_info * mtd, struct erase_info * instr,  
int allowbbt);
```

## Arguments

<i>mtd</i>	MTD device structure
<i>instr</i>	erase instruction
<i>allowbbt</i>	allow erasing the bbt area

## Description

Erase one ore more blocks.

## Name

nand\_sync — [MTD Interface] sync

## Synopsis

```
void nand_sync (struct mtd_info * mtd);
```

## Arguments

*mtd* MTD device structure

## Description

Sync is actually a wait for chip ready function.

## Name

nand\_block\_isbad — [MTD Interface] Check if block at offset is bad

## Synopsis

```
int nand_block_isbad (struct mtd_info * mtd, loff_t offs);
```

## Arguments

*mtd*    MTD device structure

*offs*   offset relative to mtd start

## Name

nand\_block\_markbad — [MTD Interface] Mark block at the given offset as bad

## Synopsis

```
int nand_block_markbad (struct mtd_info * mtd, loff_t ofs);
```

## Arguments

*mtd* MTD device structure

*ofs* offset relative to mtd start

## Name

`nand_onfi_set_features` — [REPLACEABLE] set features for ONFI nand

## Synopsis

```
int nand_onfi_set_features (struct mtd_info * mtd, struct nand_chip *  
chip, int addr, uint8_t * subfeature_param);
```

## Arguments

<i>mtd</i>	MTD device structure
<i>chip</i>	nand chip info structure
<i>addr</i>	feature address.
<i>subfeature_param</i>	the subfeature parameters, a four bytes array.

## Name

`nand_onfi_get_features` — [REPLACEABLE] get features for ONFI nand

## Synopsis

```
int nand_onfi_get_features (struct mtd_info * mtd, struct nand_chip *  
chip, int addr, uint8_t * subfeature_param);
```

## Arguments

<i>mtd</i>	MTD device structure
<i>chip</i>	nand chip info structure
<i>addr</i>	feature address.
<i>subfeature_param</i>	the subfeature parameters, a four bytes array.

## Name

`nand_suspend` — [MTD Interface] Suspend the NAND flash

## Synopsis

```
int nand_suspend (struct mtd_info * mtd);
```

## Arguments

*mtd* MTD device structure

## Name

`nand_resume` — [MTD Interface] Resume the NAND flash

## Synopsis

```
void nand_resume (struct mtd_info * mtd);
```

## Arguments

*mtd* MTD device structure



## Name

`nand_shutdown` — [MTD Interface] Finish the current NAND operation and prevent further operations

## Synopsis

```
void nand_shutdown (struct mtd_info * mtd);
```

## Arguments

*mtd* MTD device structure

## Name

`check_pattern` — [GENERIC] check if a pattern is in the buffer

## Synopsis

```
int check_pattern (uint8_t * buf, int len, int paglen, struct  
nand_bbt_descr * td);
```

## Arguments

<i>buf</i>	the buffer to search
<i>len</i>	the length of buffer to search
<i>paglen</i>	the pagelength
<i>td</i>	search pattern descriptor

## Description

Check for a pattern at the given place. Used to search bad block tables and good / bad block identifiers.

## Name

`check_short_pattern` — [GENERIC] check if a pattern is in the buffer

## Synopsis

```
int check_short_pattern (uint8_t * buf, struct nand_bbt_descr * td);
```

## Arguments

*buf*    the buffer to search

*td*    search pattern descriptor

## Description

Check for a pattern at the given place. Used to search bad block tables and good / bad block identifiers. Same as `check_pattern`, but no optional empty check.

## Name

`add_marker_len` — compute the length of the marker in data area

## Synopsis

```
u32 add_marker_len (struct nand_bbt_descr * td);
```

## Arguments

*td* BBT descriptor used for computation

## Description

The length will be 0 if the marker is located in OOB area.

## Name

`read_bbt` — [GENERIC] Read the bad block table starting from page

## Synopsis

```
int read_bbt (struct mtd_info * mtd, uint8_t * buf, int page, int num,  
struct nand_bbt_descr * td, int offs);
```

## Arguments

*mtd*    MTD device structure

*buf*    temporary buffer

*page*   the starting page

*num*    the number of bbt descriptors to read

*td*     the bbt description table

*offs*   block number offset in the table

## Description

Read the bad block table starting from page.

## Name

`read_abs_bbt` — [GENERIC] Read the bad block table starting at a given page

## Synopsis

```
int read_abs_bbt (struct mtd_info * mtd, uint8_t * buf, struct
nand_bbt_descr * td, int chip);
```

## Arguments

*mtd*    MTD device structure

*buf*    temporary buffer

*td*    descriptor for the bad block table

*chip*   read the table for a specific chip, -1 read all chips; applies only if NAND\_BBT\_PERCHIP option is set

## Description

Read the bad block table for all chips starting at a given page. We assume that the bbt bits are in consecutive order.

## Name

`scan_read_oob` — [GENERIC] Scan data+OOB region to buffer

## Synopsis

```
int scan_read_oob (struct mtd_info * mtd, uint8_t * buf, loff_t offs,  
size_t len);
```

## Arguments

*mtd*    MTD device structure

*buf*    temporary buffer

*offs*   offset at which to scan

*len*    length of data region to read

## Description

Scan read data from data+OOB. May traverse multiple pages, interleaving page,OOB,page,OOB,... in *buf*. Completes transfer and returns the “strongest” ECC condition (error or bitflip). May quit on the first (non-ECC) error.

## Name

`read_abs_bbts` — [GENERIC] Read the bad block table(s) for all chips starting at a given page

## Synopsis

```
void read_abs_bbts (struct mtd_info * mtd, uint8_t * buf, struct  
nand_bbt_descr * td, struct nand_bbt_descr * md);
```

## Arguments

*mtd* MTD device structure

*buf* temporary buffer

*td* descriptor for the bad block table

*md* descriptor for the bad block table mirror

## Description

Read the bad block table(s) for all chips starting at a given page. We assume that the bbt bits are in consecutive order.



## Name

`create_bbt` — [GENERIC] Create a bad block table by scanning the device

## Synopsis

```
int create_bbt (struct mtd_info * mtd, uint8_t * buf, struct
nand_bbt_descr * bd, int chip);
```

## Arguments

*mtd* MTD device structure

*buf* temporary buffer

*bd* descriptor for the good/bad block search pattern

*chip* create the table for a specific chip, -1 read all chips; applies only if NAND\_BBT\_PERCHIP option is set

## Description

Create a bad block table by scanning the device for the given good/bad block identify pattern.

## Name

`search_bbt` — [GENERIC] scan the device for a specific bad block table

## Synopsis

```
int search_bbt (struct mtd_info * mtd, uint8_t * buf, struct
nand_bbt_descr * td);
```

## Arguments

*mtd* MTD device structure

*buf* temporary buffer

*td* descriptor for the bad block table

## Description

Read the bad block table by searching for a given ident pattern. Search is performed either from the beginning up or from the end of the device downwards. The search starts always at the start of a block. If the option `NAND_BBT_PERCHIP` is given, each chip is searched for a bbt, which contains the bad block information of this chip. This is necessary to provide support for certain DOC devices.

The bbt ident pattern resides in the oob area of the first page in a block.

## Name

`search_read_bbts` — [GENERIC] scan the device for bad block table(s)

## Synopsis

```
void search_read_bbts (struct mtd_info * mtd, uint8_t * buf, struct  
nand_bbt_descr * td, struct nand_bbt_descr * md);
```

## Arguments

*mtd* MTD device structure

*buf* temporary buffer

*td* descriptor for the bad block table

*md* descriptor for the bad block table mirror

## Description

Search and read the bad block table(s).

## Name

`get_bbt_block` — Get the first valid eraseblock suitable to store a BBT

## Synopsis

```
int get_bbt_block (struct nand_chip * this, struct nand_bbt_descr * td,  
struct nand_bbt_descr * md, int chip);
```

## Arguments

*this*    the NAND device

*td*      the BBT description

*md*      the mirror BBT descriptor

*chip*    the CHIP selector

## Description

This functions returns a positive block number pointing a valid eraseblock suitable to store a BBT (i.e. in the range reserved for BBT), or -ENOSPC if all blocks are already used or marked bad. If `td->pages[chip]` was already pointing to a valid block we re-use it, otherwise we search for the next valid one.

## Name

`mark_bbt_block_bad` — Mark one of the block reserved for BBT bad

## Synopsis

```
void mark_bbt_block_bad (struct nand_chip * this, struct nand_bbt_descr  
* td, int chip, int block);
```

## Arguments

*this*     the NAND device

*td*       the BBT description

*chip*     the CHIP selector

*block*    the BBT block to mark

## Description

Blocks reserved for BBT can become bad. This functions is an helper to mark such blocks as bad. It takes care of updating the in-memory BBT, marking the block as bad using a bad block marker and invalidating the associated `td->pages[]` entry.

## Name

`write_bbt` — [GENERIC] (Re)write the bad block table

## Synopsis

```
int write_bbt (struct mtd_info * mtd, uint8_t * buf, struct
nand_bbt_descr * td, struct nand_bbt_descr * md, int chipsel);
```

## Arguments

<i>mtd</i>	MTD device structure
<i>buf</i>	temporary buffer
<i>td</i>	descriptor for the bad block table
<i>md</i>	descriptor for the bad block table mirror
<i>chipsel</i>	selector for a specific chip, -1 for all

## Description

(Re)write the bad block table.

## Name

`nand_memory_bbt` — [GENERIC] create a memory based bad block table

## Synopsis

```
int nand_memory_bbt (struct mtd_info * mtd, struct nand_bbt_descr * bd);
```

## Arguments

*mtd* MTD device structure

*bd* descriptor for the good/bad block search pattern

## Description

The function creates a memory based bbt by scanning the device for manufacturer / software marked good / bad blocks.

## Name

`check_create` — [GENERIC] create and write bbt(s) if necessary

## Synopsis

```
int check_create (struct mtd_info * mtd, uint8_t * buf, struct
nand_bbt_descr * bd);
```

## Arguments

*mtd* MTD device structure

*buf* temporary buffer

*bd* descriptor for the good/bad block search pattern

## Description

The function checks the results of the previous call to `read_bbt` and creates / updates the bbt(s) if necessary. Creation is necessary if no bbt was found for the chip/device. Update is necessary if one of the tables is missing or the version nr. of one table is less than the other.



## Name

`mark_bbt_region` — [GENERIC] mark the bad block table regions

## Synopsis

```
void mark_bbt_region (struct mtd_info * mtd, struct nand_bbt_descr *  
td);
```

## Arguments

*mtd* MTD device structure

*td* bad block table descriptor

## Description

The bad block table regions are marked as “bad” to prevent accidental erasures / writes. The regions are identified by the mark 0x02.

## Name

`verify_bbt_descr` — verify the bad block description

## Synopsis

```
void verify_bbt_descr (struct mtd_info * mtd, struct nand_bbt_descr *  
bd);
```

## Arguments

*mtd* MTD device structure

*bd* the table to verify

## Description

This functions performs a few sanity checks on the bad block description table.

## Name

`nand_scan_bbt` — [NAND Interface] scan, find, read and maybe create bad block table(s)

## Synopsis

```
int nand_scan_bbt (struct mtd_info * mtd, struct nand_bbt_descr * bd);
```

## Arguments

*mtd* MTD device structure

*bd* descriptor for the good/bad block search pattern

## Description

The function checks, if a bad block table(s) is/are already available. If not it scans the device for manufacturer marked good / bad blocks and writes the bad block table(s) to the selected place.

The bad block table memory is allocated here. It must be freed by calling the `nand_free_bbt` function.

## Name

nand\_update\_bbt — update bad block table(s)

## Synopsis

```
int nand_update_bbt (struct mtd_info * mtd, loff_t offs);
```

## Arguments

*mtd*    MTD device structure

*offs*   the offset of the newly marked block

## Description

The function updates the bad block table(s).

## Name

`nand_create_badblock_pattern` — [INTERN] Creates a BBT descriptor structure

## Synopsis

```
int nand_create_badblock_pattern (struct nand_chip * this);
```

## Arguments

*this* NAND chip to create descriptor for

## Description

This function allocates and initializes a `nand_bbt_descr` for BBM detection based on the properties of *this*. The new descriptor is stored in `this->badblock_pattern`. Thus, `this->badblock_pattern` should be NULL when passed to this function.

## Name

`nand_default_bbt` — [NAND Interface] Select a default bad block table for the device

## Synopsis

```
int nand_default_bbt (struct mtd_info * mtd);
```

## Arguments

*mtd* MTD device structure

## Description

This function selects the default bad block table support for the device and calls the `nand_scan_bbt` function.

## Name

`nand_isreserved_bbt` — [NAND Interface] Check if a block is reserved

## Synopsis

```
int nand_isreserved_bbt (struct mtd_info * mtd, loff_t offs);
```

## Arguments

*mtd*    MTD device structure

*offs*   offset in the device

## Name

nand\_isbad\_bbt — [NAND Interface] Check if a block is bad

## Synopsis

```
int nand_isbad_bbt (struct mtd_info * mtd, loff_t offs, int allowbbt);
```

## Arguments

<i>mtd</i>	MTD device structure
<i>offs</i>	offset in the device
<i>allowbbt</i>	allow access to bad block table region



## Name

`nand_markbad_bbt` — [NAND Interface] Mark a block bad in the BBT

## Synopsis

```
int nand_markbad_bbt (struct mtd_info * mtd, loff_t offs);
```

## Arguments

*mtd*    MTD device structure

*offs*   offset of the bad block

---

# Chapter 12. Credits

The following people have contributed to the NAND driver:

1. Steven J. Hill<sjhill@realitydiluted.com>
2. David Woodhouse<dwmw2@infradead.org>
3. Thomas Gleixner<tglx@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 Gleixner<tglx@linutronix.de>