```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE book PUBLIC "-//OASIS//DTD DocBook XML V4.1.2//EN"
        "http://www.oasis-open.org/docbook/xml/4.1.2/docbookx.dtd" []>

<book id="MTD-NAND-Guide">
 <bookinfo>
  <title>MTD NAND Driver Programming Interface</title>

  <authorgroup>
   <author>
    <firstname>Thomas</firstname>
    <surname>Gleixner</surname>
    <affiliation>
     <address>
      <email>tglx@linutronix.de</email>
     </address>
    </affiliation>
   </author>
  </authorgroup>

  <copyright>
   <year>2004</year>
   <holder>Thomas Gleixner</holder>
  </copyright>

  <legalnotice>
   <para>
     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.
   </para>

   <para>
     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.
   </para>

   <para>
     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
   </para>

   <para>
     For more details see the file COPYING in the source
     distribution of Linux.
   </para>
  </legalnotice>
 </bookinfo>

<toc></toc>

  <chapter id="intro">
```

```
      <title>Introduction</title>
  <para>
        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.
  </para>
  <para>
        This documentation is provided for developers who want to implement
        board drivers or filesystem drivers suitable for NAND devices.
  </para>
  </chapter>

  <chapter id="bugs">
     <title>Known Bugs And Assumptions</title>
  <para>
        None.
  </para>
  </chapter>

  <chapter id="dochints">
     <title>Documentation hints</title>
     <para>
     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.
     </para>
     <sect1 id="Function_identifiers_XXX">
        <title>Function identifiers [XXX]</title>
        <para>
        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:
        </para>
        <itemizedlist>
                <listitem><para>
                [MTD Interface]</para><para>
                These functions provide the interface to the MTD kernel API.
                They are not replacable and provide functionality
                which is complete hardware independent.
                </para></listitem>
                <listitem><para>
                [NAND Interface]</para><para>
                These functions are exported and provide the interface to the
NAND kernel API.
                </para></listitem>
                <listitem><para>
                [GENERIC]</para><para>
                Generic functions are not replacable and provide functionality
                which is complete hardware independent.
                </para></listitem>
                <listitem><para>
                [DEFAULT]</para><para>
                Default functions provide hardware related functionality which
is suitable
                for most of the implementations. These functions can be replaced
```

```
by the
                board driver if neccecary. 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.
                </para></listitem>
        </itemizedlist>
     </sect1>
     <sect1 id="Struct_member_identifiers_XXX">
        <title>Struct member identifiers [XXX]</title>
        <para>
        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:
        </para>
        <itemizedlist>
                <listitem><para>
                [INTERN]</para><para>
                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().
                </para></listitem>
                <listitem><para>
                [REPLACEABLE]</para><para>
                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.
                </para></listitem>
                <listitem><para>
                [BOARDSPECIFIC]</para><para>
                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().
                </para></listitem>
                <listitem><para>
                [OPTIONAL]</para><para>
                Optional members can hold information relevant for the board
driver. The
                generic NAND driver code does not use this information.
```

```
                </para></listitem>
        </itemizedlist>
     </sect1>
  </chapter>

  <chapter id="basicboarddriver">
        <title>Basic board driver</title>
        <para>
                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.
        </para>
        <sect1 id="Basic_defines">
                <title>Basic defines</title>
                <para>
                        At least you have to provide a mtd structure and
                        a storage for the ioremap'ed chip address.
                        You can allocate the mtd structure using kmalloc
                        or you can allocate it statically.
                        In case of static allocation you have to allocate
                        a nand_chip structure too.
                </para>
                <para>
                        Kmalloc based example
                </para>
                <programlisting>
static struct mtd_info *board_mtd;
static void __iomem *baseaddr;
                </programlisting>
                <para>
                        Static example
                </para>
                <programlisting>
static struct mtd_info board_mtd;
static struct nand_chip board_chip;
static void __iomem *baseaddr;
                </programlisting>
        </sect1>
        <sect1 id="Partition_defines">
                <title>Partition defines</title>
                <para>
                        If you want to divide your device into partitions, then
                        enable the configuration switch CONFIG_MTD_PARTITIONS
and define
                        a partitioning scheme suitable to your board.
                </para>
                <programlisting>
#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 },
};
```

```
                </programlisting>
        </sect1>
        <sect1 id="Hardware_control_functions">
                <title>Hardware control function</title>
                <para>
                        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.
                </para>
                <para>
                        <emphasis>GPIO based example</emphasis>
                </para>
                <programlisting>
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;
        }
}
                </programlisting>
                <para>
                        <emphasis>Address lines based example.</emphasis> It's
assumed that the
                        nCE pin is driven by a chip select decoder.
                </para>
                <programlisting>
static void board_hwcontrol(struct mtd_info *mtd, int cmd)
{
        struct nand_chip *this = (struct nand_chip *) mtd->priv;
        switch(cmd){
                case NAND_CTL_SETCLE: this->IO_ADDR_W |= CLE_ADRR_BIT;  break;
                case NAND_CTL_CLRCLE: this->IO_ADDR_W &amp;= ~CLE_ADRR_BIT;
break;
                case NAND_CTL_SETALE: this->IO_ADDR_W |= ALE_ADRR_BIT;  break;
                case NAND_CTL_CLRALE: this->IO_ADDR_W &amp;= ~ALE_ADRR_BIT;
break;
        }
}
                </programlisting>
        </sect1>
        <sect1 id="Device_ready_function">
                <title>Device ready function</title>
                <para>
                        If the hardware interface has the ready busy pin of the
NAND chip connected to a
                        GPIO or other accesible 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.
</para>
</sect1>
<sect1 id="Init_function">
<title>Init function</title>
<para>
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 neccecary
information about the device.
</para>
<programlisting>
```
static int __init board_init (void)
{
	struct nand_chip *this;
	int err = 0;

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

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

	/* Get pointer to private data */
	this = (struct nand_chip *) ();
	/* Link the private data with the MTD structure */
	board_mtd->priv = this;

	/* 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 (board_mtd);
out:
        return err;
}
module_init(board_init);
```

```
                </programlisting>
        </sect1>
        <sect1 id="Exit_function">
                <title>Exit function</title>
                <para>
                        The exit function is only neccecary 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.
                </para>
                <programlisting>
```

```
#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 (board_mtd);
}
module_exit(board_cleanup);
#endif
```

```
                </programlisting>
        </sect1>
  </chapter>

  <chapter id="boarddriversadvanced">
        <title>Advanced board driver functions</title>
        <para>
                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.
        </para>
        <sect1 id="Multiple_chip_control">
                <title>Multiple chip control</title>
                <para>
                        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.
                </para>
                <para>
                        The nand driver concatenates the chips to one virtual
                        chip and provides this virtual chip to the MTD layer.
                </para>
                <para>
                        <emphasis>Note: The driver can only handle linear chip
arrays
                        of equally sized chips. There is no support for
                        parallel arrays which extend the buswidth.</emphasis>
                </para>
                <para>
                        <emphasis>GPIO based example</emphasis>
                </para>
                <programlisting>
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) &amp;= ~ (1 &lt;&lt; chip);
}
                </programlisting>
                <para>
                        <emphasis>Address lines based example.</emphasis>
                        Its assumed that the nCE pins are connected to an
                        address decoder.
                </para>
                <programlisting>
static void board_select_chip (struct mtd_info *mtd, int chip)
{
        struct nand_chip *this = (struct nand_chip *) mtd->priv;

        /* Deselect all chips */
        this->IO_ADDR_R &amp;= ~BOARD_NAND_ADDR_MASK;
        this->IO_ADDR_W &amp;= ~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;
        }
}
                </programlisting>
        </sect1>
        <sect1 id="Hardware_ECC_support">
                <title>Hardware ECC support</title>
                <sect2 id="Functions_and_constants">
                        <title>Functions and constants</title>
                        <para>
                                The nand driver supports three different types
of
                                hardware ECC.
                                <itemizedlist>
                                <listitem><para>NAND_ECC_HW3_256</para><para>
                                Hardware ECC generator providing 3 bytes ECC per
                                256 byte.
                                </para> </listitem>
                                <listitem><para>NAND_ECC_HW3_512</para><para>
                                Hardware ECC generator providing 3 bytes ECC per
                                512 byte.
                                </para> </listitem>
                                <listitem><para>NAND_ECC_HW6_512</para><para>
                                Hardware ECC generator providing 6 bytes ECC per
                                512 byte.
                                </para> </listitem>
                                <listitem><para>NAND_ECC_HW8_512</para><para>
                                Hardware ECC generator providing 6 bytes ECC per
                                512 byte.
                                </para> </listitem>
                                </itemizedlist>
                                If your hardware generator has a different
functionality
                                add it at the appropriate place in nand_base.c
                        </para>
                        <para>
                                The board driver must provide following
functions:
                                <itemizedlist>
                                <listitem><para>enable_hwecc</para><para>
                                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.
                                </para> </listitem>
                                <listitem><para>calculate_ecc</para><para>
                                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.
</para> </listitem>
<listitem><para>correct_data</para><para>
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.
</para> </listitem>
</itemizedlist>
</para>
</sect2>
<sect2 id="Hardware_ECC_with_syndrome_calculation">
<title>Hardware ECC with syndrome calculation</title>
<para>
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.
</para>
<para>
The ECC bytes must be placed immidiately 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

```
                                mtdnand.tmpl.txt
also
                                use bad block tables on FLASH, because the ECC
layout is
                                interferring with the bad block marker
positions.
                                See bad block table support for details.
                        </para>
                </sect2>
        </sect1>
        <sect1 id="Bad_Block_table_support">
                <title>Bad block table support</title>
                <para>
                        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.
                </para>
                <para>
                        The nand driver supports various types of bad block
                        tables.
                        <itemizedlist>
                        <listitem><para>Per device</para><para>
                        The bad block table contains all bad block information
                        of the device which can consist of multiple chips.
                        </para> </listitem>
                        <listitem><para>Per chip</para><para>
                        A bad block table is used per chip and contains the
                        bad block information for this particular chip.
                        </para> </listitem>
                        <listitem><para>Fixed offset</para><para>
                        The bad block table is located at a fixed offset
                        in the chip (device). This applies to various
                        DiskOnChip devices.
                        </para> </listitem>
                        <listitem><para>Automatic placed</para><para>
                        The bad block table is automatically placed and
                        detected either at the end or at the beginning
                        of a chip (device)
                        </para> </listitem>
                        <listitem><para>Mirrored tables</para><para>
                        The bad block table is mirrored on the chip (device) to
                        allow updates of the bad block table without data loss.
                        </para> </listitem>
                        </itemizedlist>
                </para>
                <para>
                        nand_scan() calls the function nand_default_bbt().
                        nand_default_bbt() selects appropriate default
                        bad block table desriptors depending on the chip
information
                        which was retrieved by nand_scan().
                </para>
```

```
		<para>
			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.
		</para>
		<sect2 id="Flash_based_tables">
			<title>Flash based tables</title>
			<para>
				It may be desired or neccecary 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.
			</para>
			<para>
				The blocks in which the tables are stored are
procteted against
				accidental access by marking them bad in the
memory bad block
				table. The bad block table management functions
are allowed
				to circumvernt this protection.
			</para>
			<para>
				The simplest way to activate the FLASH based bad
block table support
				is to set the option NAND_USE_FLASH_BBT in the
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
				<itemizedlist>
				<listitem><para>Store bad block table per
chip</para></listitem>
				<listitem><para>Use 2 bits per
block</para></listitem>
				<listitem><para>Automatic placement at the end
of the chip</para></listitem>
				<listitem><para>Use mirrored tables with version
```

```
numbers</para></listitem>
                        <listitem><para>Reserve 4 blocks at the end of
the chip</para></listitem>
                        </itemizedlist>
                </para>
        </sect2>
        <sect2 id="User_defined_tables">
                <title>User defined tables</title>
                <para>
                        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 neccecary 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.
                </para>
                <para>
                        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.
                        <itemizedlist>
                        <listitem><para>Number of bits per block</para>
                        <para>The supported number of bits is 1, 2, 4,
8.</para></listitem>
                        <listitem><para>Table per chip</para>
                        <para>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.</para></listitem>
                        <listitem><para>Table location is
absolute</para>
                        <para>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
fields
uninitialized</para></listitem>
detected</para>
first or the last good
NAND_BBT_LASTBLOCK to place
(device). The
pattern which
the block which
the pattern
pattern has to be
must be given
stucture. For mirrored
mandatory.</para></listitem>

the table creation
Usually this is done only


the table write support.
in case a block has
function block_markbad
table. If the write
FLASH.</para>

mirrored tables with


the table version control.
mirrored tables with write
the bad block
```

for a table ident pattern performed, so the
pattern, veroffs, offs, len can be left
<listitem><para>Table location is automatically
<para>The table can either be located in the
blocks of the chip (device). Set
the bad block table at the end of the chip
bad block tables are marked and identified by a
is stored in the spare area of the first page in
holds the bad block table. Store a pointer to
in the pattern field. Further the length of the
stored in len and the offset in the spare area
in the offs member of the nand_bbt_descr
bad block tables different patterns are

```
<listitem><para>Table creation</para>
<para>Set the option NAND_BBT_CREATE to enable
if no table can be found during the scan.
once if a new chip is found. </para></listitem>
<listitem><para>Table write support</para>
<para>Set the option NAND_BBT_WRITE to enable
This allows the update of the bad block table(s)
to be marked bad due to wear. The MTD interface
is calling the update function of the bad block
support is enabled then the table is updated on
<para>
Note: Write support should only be enabled for
version control.
</para></listitem>
<listitem><para>Table version control</para>
<para>Set the option NAND_BBT_VERSION to enable
It's highly recommended to enable this for
support. It makes sure that the risk of loosing
```

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.</para></listitem>
<listitem><para>Save block contents on
write</para>
<para>
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.
</para></listitem>
<listitem><para>Number of reserved blocks</para>
<para>
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 babd 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.
</para></listitem>
</itemizedlist>
</para>
</sect2>
</sect1>
<sect1 id="Spare_area_placement">
<title>Spare area (auto)placement</title>
<para>
The nand driver implements different possibilities for
placement of filesystem data in the spare area,
<itemizedlist>
<listitem><para>Placement defined by fs
driver</para></listitem>
<listitem><para>Automatic placement</para></listitem>
</itemizedlist>
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.
</para>
<para>
File system drivers can provide a own placement scheme
which
is used instead of the default placement scheme.
</para>
<para>
Placement schemes are defined by a nand_oobinfo
structure
<programlisting>
struct nand_oobinfo {
int useecc;
int eccbytes;
int eccpos[24];
int oobfree[8][2];
};
</programlisting>
<itemizedlist>
<listitem><para>useecc</para><para>
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.
</para></listitem>
<listitem><para>eccbytes</para><para>
The eccbytes member defines the number of ecc
bytes per page.
</para></listitem>
<listitem><para>eccpos</para><para>
The eccpos array holds the byte offsets in the
spare area where
the ecc codes are placed.
</para></listitem>
<listitem><para>oobfree</para><para>
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.
</para></listitem>

```
                </itemizedlist>
        </para>
        <sect2 id="Placement_defined_by_fs_driver">
                <title>Placement defined by fs driver</title>
                <para>
                        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
                </para>
                <para>
                        &lt;spare data page 0&gt;&lt;ecc result
0&gt;...&lt;ecc result n&gt;
                </para>
                <para>
                        ...
                </para>
                <para>
                        &lt;spare data page n&gt;&lt;ecc result
0&gt;...&lt;ecc result n&gt;
                </para>
                <para>
                        This is a legacy mode used by YAFFS1.
                </para>
                <para>
                        If the spare area buffer is NULL then only the
ECC placement is
                        done according to the given scheme in the
nand_oobinfo structure.
                </para>
        </sect2>
        <sect2 id="Automatic_placement">
                <title>Automatic placement</title>
                <para>
                        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.
```

mtdnand.tmpl.txt
</para>
<para>
If the spare area buffer is NULL then only the ECC placement is
done according to the default builtin scheme.
</para>
</sect2>
<sect2 id="User_space_placement_selection">
<title>User space placement selection</title>
<para>
All non ecc functions like mtd->read and mtd->write use an internal
structure, which can be set by an ioctl. This structure is preset
to the autoplacement default.
<programlisting>
ioctl (fd, MEMSETOOBSEL, oobsel);
</programlisting>
oobsel is a pointer to a user supplied structure of type
nand_oobconfig. The contents of this structure must match the
criteria of the filesystem, which will be used. See an example in utils/nandwrite.c.
</para>
</sect2>
</sect1>
<sect1 id="Spare_area_autoplacement_default">
<title>Spare area autoplacement default schemes</title>
<sect2 id="pagesize_256">
<title>256 byte pagesize</title>
<informaltable><tgroup cols="3"><tbody>
<row>
<entry>Offset</entry>
<entry>Content</entry>
<entry>Comment</entry>
</row>
<row>
<entry>0x00</entry>
<entry>ECC byte 0</entry>
<entry>Error correction code byte 0</entry>
</row>
<row>
<entry>0x01</entry>
<entry>ECC byte 1</entry>
<entry>Error correction code byte 1</entry>
</row>
<row>
<entry>0x02</entry>
<entry>ECC byte 2</entry>
<entry>Error correction code byte 2</entry>
</row>
<row>
<entry>0x03</entry>
<entry>Autoplace 0</entry>
<entry></entry>
</row>

```
<row>
<entry>0x04</entry>
<entry>Autoplace 1</entry>
<entry></entry>
</row>
<row>
<entry>0x05</entry>
<entry>Bad block marker</entry>
<entry>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</entry>
</row>
<row>
<entry>0x06</entry>
<entry>Autoplace 2</entry>
<entry></entry>
</row>
<row>
<entry>0x07</entry>
<entry>Autoplace 3</entry>
<entry></entry>
</row>
</tbody></tgroup></informaltable>
                </sect2>
                <sect2 id="pagesize_512">
                        <title>512 byte pagesize</title>
<informaltable><tgroup cols="3"><tbody>
<row>
<entry>Offset</entry>
<entry>Content</entry>
<entry>Comment</entry>
</row>
<row>
<entry>0x00</entry>
<entry>ECC byte 0</entry>
<entry>Error correction code byte 0 of the lower 256 Byte data in
this page</entry>
</row>
<row>
<entry>0x01</entry>
<entry>ECC byte 1</entry>
<entry>Error correction code byte 1 of the lower 256 Bytes of data
in this page</entry>
</row>
<row>
<entry>0x02</entry>
<entry>ECC byte 2</entry>
<entry>Error correction code byte 2 of the lower 256 Bytes of data
in this page</entry>
</row>
<row>
<entry>0x03</entry>
<entry>ECC byte 3</entry>
<entry>Error correction code byte 0 of the upper 256 Bytes of data
in this page</entry>
</row>
```

```
<row>
<entry>0x04</entry>
<entry>reserved</entry>
<entry>reserved</entry>
</row>
<row>
<entry>0x05</entry>
<entry>Bad block marker</entry>
<entry>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</entry>
</row>
<row>
<entry>0x06</entry>
<entry>ECC byte 4</entry>
<entry>Error correction code byte 1 of the upper 256 Bytes of data
in this page</entry>
</row>
<row>
<entry>0x07</entry>
<entry>ECC byte 5</entry>
<entry>Error correction code byte 2 of the upper 256 Bytes of data
in this page</entry>
</row>
<row>
<entry>0x08 - 0x0F</entry>
<entry>Autoplace 0 - 7</entry>
<entry></entry>
</row>
</tbody></tgroup></informaltable>
                </sect2>
                <sect2 id="pagesize_2048">
                        <title>2048 byte pagesize</title>
<informaltable><tgroup cols="3"><tbody>
<row>
<entry>Offset</entry>
<entry>Content</entry>
<entry>Comment</entry>
</row>
<row>
<entry>0x00</entry>
<entry>Bad block marker</entry>
<entry>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</entry>
</row>
<row>
<entry>0x01</entry>
<entry>Reserved</entry>
<entry>Reserved</entry>
</row>
<row>
<entry>0x02-0x27</entry>
<entry>Autoplace 0 - 37</entry>
<entry></entry>
</row>
```

```
<row>
<entry>0x28</entry>
<entry>ECC byte 0</entry>
<entry>Error correction code byte 0 of the first 256 Byte data in
this page</entry>
</row>
<row>
<entry>0x29</entry>
<entry>ECC byte 1</entry>
<entry>Error correction code byte 1 of the first 256 Bytes of data
in this page</entry>
</row>
<row>
<entry>0x2A</entry>
<entry>ECC byte 2</entry>
<entry>Error correction code byte 2 of the first 256 Bytes data in
this page</entry>
</row>
<row>
<entry>0x2B</entry>
<entry>ECC byte 3</entry>
<entry>Error correction code byte 0 of the second 256 Bytes of data
in this page</entry>
</row>
<row>
<entry>0x2C</entry>
<entry>ECC byte 4</entry>
<entry>Error correction code byte 1 of the second 256 Bytes of data
in this page</entry>
</row>
<row>
<entry>0x2D</entry>
<entry>ECC byte 5</entry>
<entry>Error correction code byte 2 of the second 256 Bytes of data
in this page</entry>
</row>
<row>
<entry>0x2E</entry>
<entry>ECC byte 6</entry>
<entry>Error correction code byte 0 of the third 256 Bytes of data
in this page</entry>
</row>
<row>
<entry>0x2F</entry>
<entry>ECC byte 7</entry>
<entry>Error correction code byte 1 of the third 256 Bytes of data
in this page</entry>
</row>
<row>
<entry>0x30</entry>
<entry>ECC byte 8</entry>
<entry>Error correction code byte 2 of the third 256 Bytes of data
in this page</entry>
</row>
<row>
<entry>0x31</entry>
```

<entry>ECC byte 9</entry>
<entry>Error correction code byte 0 of the fourth 256 Bytes of data
in this page</entry>
</row>
<row>
<entry>0x32</entry>
<entry>ECC byte 10</entry>
<entry>Error correction code byte 1 of the fourth 256 Bytes of data
in this page</entry>
</row>
<row>
<entry>0x33</entry>
<entry>ECC byte 11</entry>
<entry>Error correction code byte 2 of the fourth 256 Bytes of data
in this page</entry>
</row>
<row>
<entry>0x34</entry>
<entry>ECC byte 12</entry>
<entry>Error correction code byte 0 of the fifth 256 Bytes of data
in this page</entry>
</row>
<row>
<entry>0x35</entry>
<entry>ECC byte 13</entry>
<entry>Error correction code byte 1 of the fifth 256 Bytes of data
in this page</entry>
</row>
<row>
<entry>0x36</entry>
<entry>ECC byte 14</entry>
<entry>Error correction code byte 2 of the fifth 256 Bytes of data
in this page</entry>
</row>
<row>
<entry>0x37</entry>
<entry>ECC byte 15</entry>
<entry>Error correction code byte 0 of the sixt 256 Bytes of data
in this page</entry>
</row>
<row>
<entry>0x38</entry>
<entry>ECC byte 16</entry>
<entry>Error correction code byte 1 of the sixt 256 Bytes of data
in this page</entry>
</row>
<row>
<entry>0x39</entry>
<entry>ECC byte 17</entry>
<entry>Error correction code byte 2 of the sixt 256 Bytes of data
in this page</entry>
</row>
<row>
<entry>0x3A</entry>
<entry>ECC byte 18</entry>
<entry>Error correction code byte 0 of the seventh 256 Bytes of

data in this page</entry>
</row>
<row>
<entry>0x3B</entry>
<entry>ECC byte 19</entry>
<entry>Error correction code byte 1 of the seventh 256 Bytes of
data in this page</entry>
</row>
<row>
<entry>0x3C</entry>
<entry>ECC byte 20</entry>
<entry>Error correction code byte 2 of the seventh 256 Bytes of
data in this page</entry>
</row>
<row>
<entry>0x3D</entry>
<entry>ECC byte 21</entry>
<entry>Error correction code byte 0 of the eigth 256 Bytes of data
in this page</entry>
</row>
<row>
<entry>0x3E</entry>
<entry>ECC byte 22</entry>
<entry>Error correction code byte 1 of the eigth 256 Bytes of data
in this page</entry>
</row>
<row>
<entry>0x3F</entry>
<entry>ECC byte 23</entry>
<entry>Error correction code byte 2 of the eigth 256 Bytes of data
in this page</entry>
</row>
</tbody></tgroup></informaltable>
		</sect2>
	</sect1>
  </chapter>

  <chapter id="filesystems">
	<title>Filesystem support</title>
	<para>
		The NAND driver provides all neccecary functions for a
		filesystem via the MTD interface.
	</para>
	<para>
		Filesystems must be aware of the NAND pecularities 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.
	</para>
	<para>
		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.

        </para>
        <para>
                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.
        </para>
  </chapter>
  <chapter id="tools">
        <title>Tools</title>
        <para>
                The MTD project provides a couple of helpful tools to handle
NAND Flash.
                <itemizedlist>
                <listitem><para>flasherase, flasheraseall: Erase and format
FLASH partitions</para></listitem>
                <listitem><para>nandwrite: write filesystem images to NAND
FLASH</para></listitem>
                <listitem><para>nanddump: dump the contents of a NAND FLASH
partitions</para></listitem>
                </itemizedlist>
        </para>
        <para>
                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.
        </para>
  </chapter>

  <chapter id="defines">
     <title>Constants</title>
     <para>
     This chapter describes the constants which might be relevant for a driver
developer.
     </para>
     <sect1 id="Chip_option_constants">
        <title>Chip option constants</title>
        <sect2 id="Constants_for_chip_id_table">
                <title>Constants for chip id table</title>
                <para>
                These constants are defined in nand.h. They are ored together to
describe
                the chip functionality.
                <programlisting>
/* Chip can not auto increment pages */
#define NAND_NO_AUTOINCR        0x00000001
/* Buswitdh is 16 bit */
#define NAND_BUSWIDTH_16        0x00000002
/* Device supports partial programming without padding */
#define NAND_NO_PADDING         0x00000004
```

```
/* Chip has cache program function */
#define NAND_CACHEPRG           0x00000008
/* Chip has copy back function */
#define NAND_COPYBACK           0x00000010
/* AND Chip which has 4 banks and a confusing page / block
 * assignment. See Renesas datasheet for further information */
#define NAND_IS_AND             0x00000020
/* Chip has a array of 4 pages which can be read without
 * additional ready /busy waits */
#define NAND_4PAGE_ARRAY        0x00000040
                </programlisting>
                </para>
        </sect2>
        <sect2 id="Constants_for_runtime_options">
                <title>Constants for runtime options</title>
                <para>
                These constants are defined in nand.h. They are ored together to
describe
                the functionality.
                <programlisting>
/* Use a flash based bad block table. This option is parsed by the
 * default bad block table function (nand_default_bbt). */
#define NAND_USE_FLASH_BBT      0x00010000
/* 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
                </programlisting>
                </para>
        </sect2>
     </sect1>

     <sect1 id="EEC_selection_constants">
        <title>ECC selection constants</title>
        <para>
        Use these constants to select the ECC algorithm.
        <programlisting>
/* 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
        </programlisting>
        </para>
     </sect1>

     <sect1 id="Hardware_control_related_constants">
        <title>Hardware control related constants</title>
        <para>
```

These constants describe the requested hardware access function when
the boardspecific hardware control function is called
<programlisting>

```
/* 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
```

</programlisting>
</para>
</sect1>

<sect1 id="Bad_block_table_constants">
<title>Bad block table related constants</title>
<para>
These constants describe the options used for bad block
table descriptors.
<programlisting>

```
/* 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
/* The bbt is at the given page, else we must scan for the bbt */
#define NAND_BBT_SEARCH         0x00000040
/* bbt is stored per chip on multichip devices */
#define NAND_BBT_PERCHIP        0x00000080
/* bbt has a version counter at offset veroffs */
#define NAND_BBT_VERSION        0x00000100
/* Create a bbt if none axists */
#define NAND_BBT_CREATE         0x00000200
/* Search good / bad pattern through all pages of a block */
#define NAND_BBT_SCANALLPAGES   0x00000400
/* Scan block empty during good / bad block scan */
#define NAND_BBT_SCANEMPTY      0x00000800
/* Write bbt if neccecary */
#define NAND_BBT_WRITE          0x00001000
/* Read and write back block contents when writing bbt */
```

#define NAND_BBT_SAVECONTENT   0x00002000
        </programlisting>
        </para>
     </sect1>

  </chapter>

  <chapter id="structs">
     <title>Structures</title>
     <para>
     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.
     </para>
!Iinclude/linux/mtd/nand.h
  </chapter>

  <chapter id="pubfunctions">
     <title>Public Functions Provided</title>
     <para>
     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.
     </para>
!Edrivers/mtd/nand/nand_base.c
!Edrivers/mtd/nand/nand_bbt.c
!Edrivers/mtd/nand/nand_ecc.c
  </chapter>

  <chapter id="intfunctions">
     <title>Internal Functions Provided</title>
     <para>
     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.
     </para>
!Idrivers/mtd/nand/nand_base.c
!Idrivers/mtd/nand/nand_bbt.c
<!-- No internal functions for kernel-doc:
X!Idrivers/mtd/nand/nand_ecc.c
-->
  </chapter>

  <chapter id="credits">
     <title>Credits</title>
        <para>

```
                The following people have contributed to the NAND driver:
                <orderedlist>
                        <listitem><para>Steven J.
Hill<email>sjhill@realitydiluted.com</email></para></listitem>
                        <listitem><para>David
Woodhouse<email>dwmw2@infradead.org</email></para></listitem>
                        <listitem><para>Thomas
Gleixner<email>tglx@linutronix.de</email></para></listitem>
                </orderedlist>
                A lot of users have provided bugfixes, improvements and helping
hands for testing.
                Thanks a lot.
        </para>
        <para>
                The following people have contributed to this document:
                <orderedlist>
                        <listitem><para>Thomas
Gleixner<email>tglx@linutronix.de</email></para></listitem>
                </orderedlist>
        </para>
  </chapter>
</book>
```