NAND Flash Memories

–

Using Linux MTD compatible mode

on Dataman Universal Device Programmers

–

(Quick Guide)

Application Note

February 2011
an_linux_mtd, version 1.03

As embeded devices become still more and more complex, specialised control software becomes still more expensive in terms of both money and time. Therefore there is a strong pressure to use some universal operating system instead of special software creations. Probably the most popular operating system in embeded world of today is Linux due to its open source and free of costs nature.

There are many embeded operating systems based on Linux, but they all use the same Linux kernel. MTD (stands for Memory Technology Devices) subsystem is the portion of Linux kernel responsible for flash memory devices management. You can find more information about MTD on their homepage: http://www.linux-mtd.infradead.org/index.html.

Linux MTD compatible feature provides the compatibility issues with Linux MTD subsystem, as involved in Linux kernel version 2.6.32.8. It allows writing of Linux MTD compatible bad blocks tables and Linux MTD compatible ECC for both, bad blocks tables and user data. See Linux kernel homepage for the most actual source code of Linux MTD subsystem: http://www.kernel.org/.

Actually, Linux MTD compatible feature is supported on 48Pro2, 448Pro2 and 848Pro2 programmers. Linux MTD subsystem specifies ECC algorithm capable to recover one single-bit error in 256-byte wide frame. The feature is enabled only for those devices that the ECC algorithm is suitable for (typically all SLC devices).

OVERVIEW

To enable **Linux MTD compatible** feature, open **Access Method** dialog window (easy accessible through clicking the link in **Device** panel in bottom right of **pg4uw** software window, or shortcut **<Alt+S>**) and set **Invalid Block Management** option to **Linux MTD compatible**, see Figure 1.

Figure 1. Enabling Linux MTD compatible feature

Once **Linux MTD compatible** feature is enabled, special options are taken into account, see Figure 2.

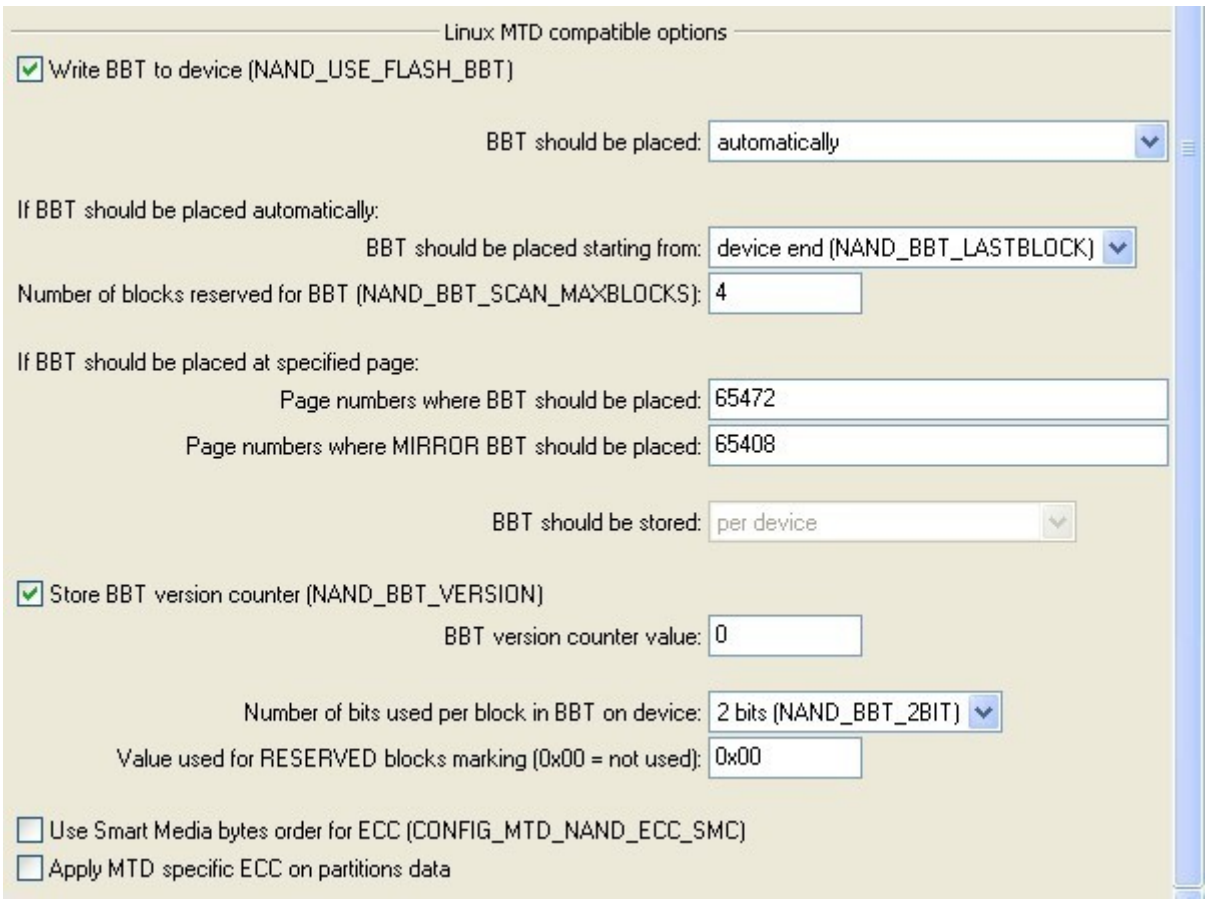


Figure 2. Special Linux MTD compatible options

ACCESS METHOD OPTIONS VALIDITY

Linux MTD subsystem uses **Skip invalid blocks** method to treat with invalid blocks in device. This method is set automatically.

Linux MTD compatible feature is intended for use with partitioned devices. From that reason, partition definition file must be loaded even if there is only single partition in device.

Setting of **Spare Area Usage** option is ignored. Linux MTD subsystem is rather low-level driver and typically accepts spare area content from higher layers (typically some file system data). Linux MTD is capable to write its own specific ECC control sums into locations specified in driver. All other spare area locations are left untouched. Anyhow, your input data file always must contain spare area data. If your higher layers don't use spare area, you can add blank spare area data automatically on file(s) load using **Add blank spare area** feature available in **Load file** dialog.

Setting of **Required Good Blocks Area** is accepted and is applied globally (i. e. over whole device, not over individual partitions).

Setting of **Max. Allowed Number of Invalid Blocks in Device** is accepted and is applied globally.

Setting of **Quick Program** option is accepted and is applied globally.

Reserved Block Area Options are irrelevant when skipping invalid blocks and respective setting is not taken into account.

Setting of **Invalid Block Indication Options** is accepted and is applied globally.

Setting of **Tolerant Verification Options** is accepted and is applied globally.

OPERATIONS EXECUTION

Blank check

The device is always blank checked entirely, from the first up to the last address location, including spare area.

If **Blank check before programming** is enabled (and is not skipped due to enabling both, Blank check and Erase before programming), it will be performed in the same way, not taking partitioning into account.

Read

Read operation is performed on per partition basis, i. e. individual partitions are processed step by step, from the first partition up to the last one.

Note: Partition unused (padding) area is not read. Buffer data at corresponding locations are not altered. Before reading the device, the programmer builds its own invalid blocks table by scanning the device for invalid blocks marks. I. e. searching for BBT stored in device is not supported and nor BBT nor mirror BBT is read.

Verify

Verify operation is performed on per partition basis, i. e. individual partitions are processed step by step, from the first partition up to the last one.

If **Verify after reading** or **Verify after programming** is used, the partition will be read or programmed firstly, and after then it will be verified.

Note: Partition unused (padding) area is not verified. Buffer data at corresponding locations are ignored. Before verifying the device, the programmer builds its own invalid blocks table by scanning the device for invalid blocks marks. I. e. searching for BBT stored in device is not supported and nor BBT nor mirror BBT is read or verified.

Program

Program operation is performed on per partition basis, i. e. individual partitions are processed step by step, from the first partition up to the last one. After all partitions programming is finished, if enabled, BBT and mirror BBT are stored and verified in device.

Note: Partition unused (padding) area is not programmed. Buffer data at corresponding locations are ignored.

Erase

The device is always erased entirely, from the first up to the last block.

If **Erase before programming** is enabled, it will be performed in the same way, not taking partitioning into account.

PROCEDURE DESCRIPTION

Setting-up Linux MTD compatible mode consists of four basic steps:

Selection of Linux MTD compatible

Open **Access Method** dialog window and set **Invalid Block Management** option to **Linux MTD compatible**, see Figure 1.

Set other general options, if necessary.

Set Linux MTD specific options, see chapter Setting Linux MTD compatible specific options for detailed information.

After clicking OK, internal software environment will be prepared for multi-partition mode. Also, special buffer for Partition Table will be created, see Figure 3. The buffer is not editable, the only way how to define partition table is loading relevant definition file.

After Linux MTD compatible method is deselected, internal software environment is switched back to standard mode and special buffer is destroyed. In consequence, partition table data are lost and must be load again after re-enabling Linux MTD compatible mode again.

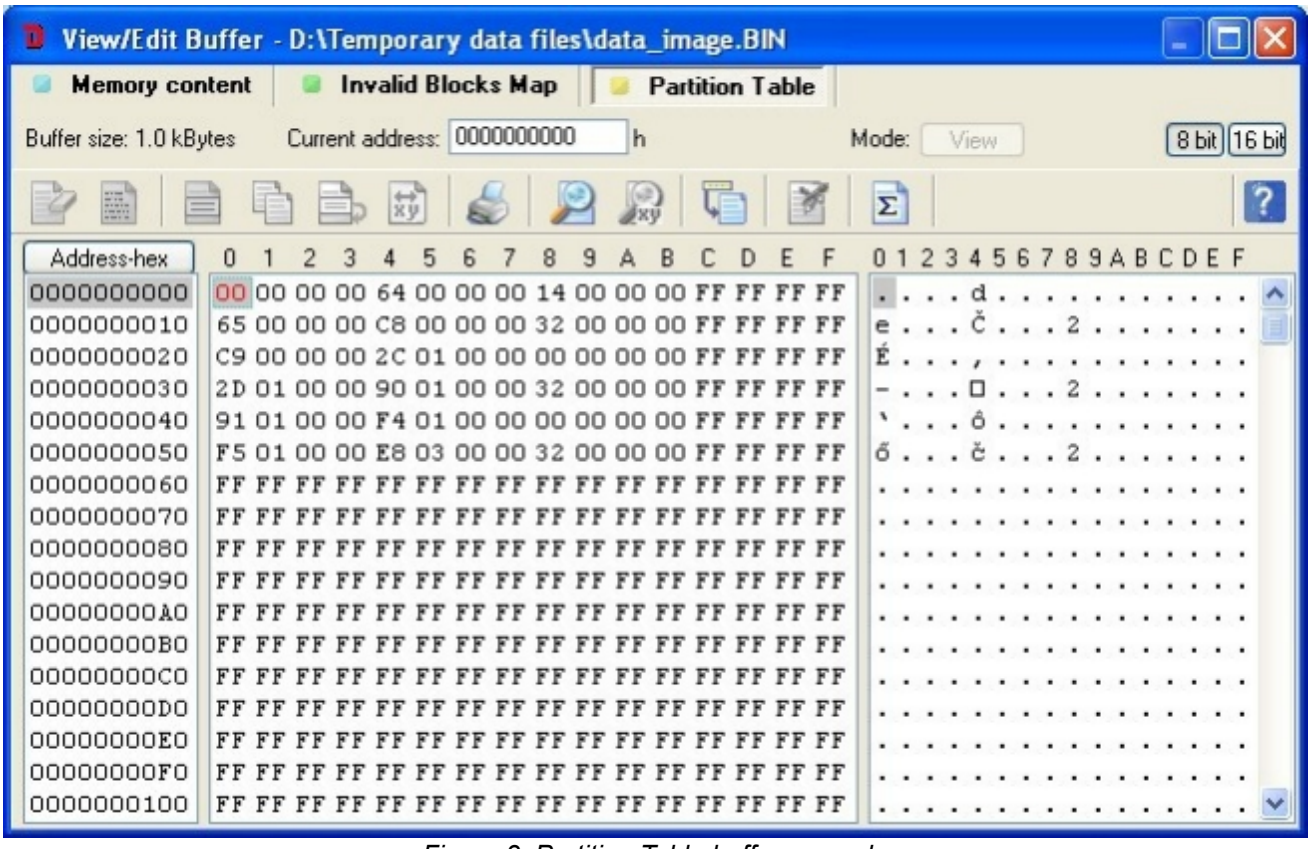


Figure 3. Partition Table buffer example

Loading partition table definition file

After enabling Linux MTD compatible mode, the partition table must be defined. There is only one way how to define the partition table - using menu **File >> Load Partition table...**, see Figure 4 and Figure 5. There are several partition definition file formats supported, ask your distributor for more information. Comma separated values format (the simplest one) is described later in chapter Using Comma Separated Vales Format.

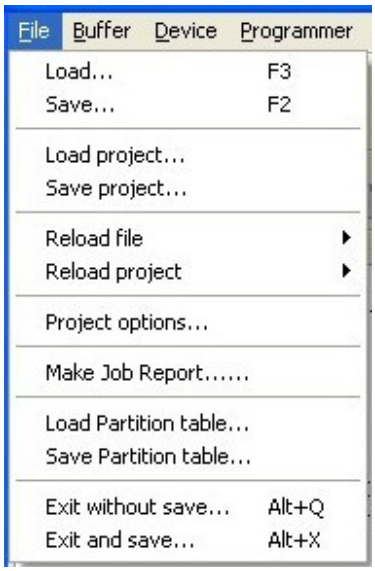


Figure 4. Load Partition table menu

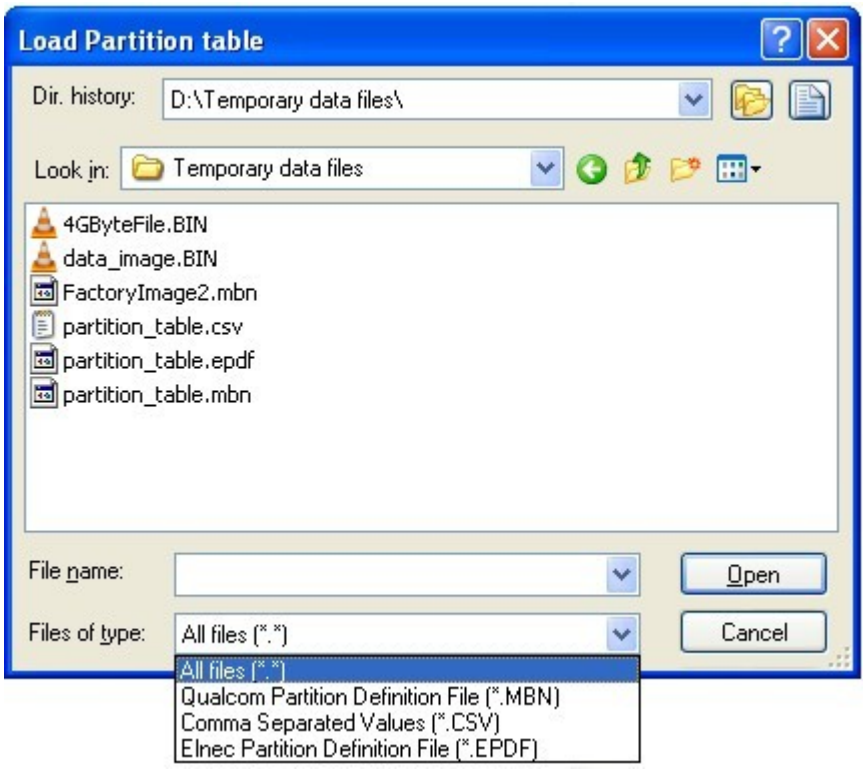


Figure 5. Load Partition table dialog

Loading data image file

If you don't use any special partitioning format (e. g. Qualcomm multiply partition), data image file should correspond with a copy of NAND flash device without any invalid blocks. It must contain data for all partitions placed at correct locations. Also, it must contain spare area data. If your Linux higher layers don't use spare area and/or your development tool doesn't produce image with spare area included, you can add blank spare area automatically by enabling **Add blank spare area** feature in File load dialog.

To load input data image file, use standard Load command (menu **File >> Load**, shortcut **<F3>** or **Load** command from **Main toolbar**).

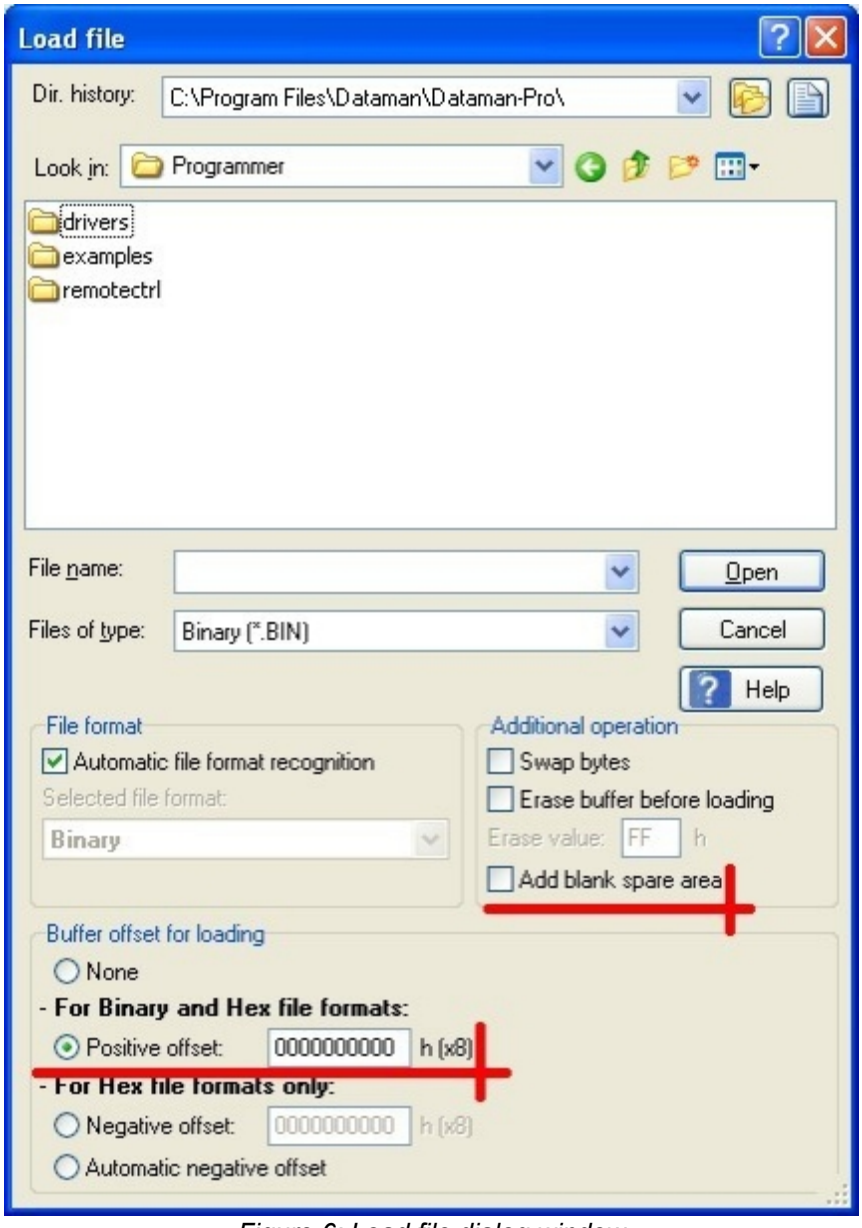


Figure 6: Load file dialog window

If you haven't a single input data image file but you have several input data files for individual partitions instead, you will need to use **Positive offset** setting in File load dialog. You can compute the offset using the following formula:

positive_offset = partition_start_block_number x number_of_pages_in_block x page_size_including_spare

In this way, you can load multiple files into buffer, each one at respective offset corresponding to partition start block address offset.

Example: Partition should start from block 30, devices has 64 pages in block and one page contains 2048 bytes of data area and 64 bytes of spare area.

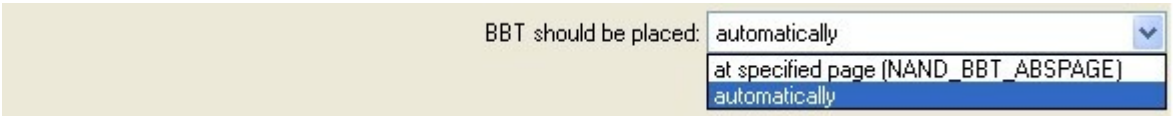
positive_offset = 30 x 64 x (2048+64) = 4055040_{dec} = 3DE000h

Setting Linux MTD compatible specific options

Write BBT to device

If enabled, both BBT and mirror BBT are stored into nand flash device using options specified further. This option corresponds to NAND_USE_FLASH_BBT option in driver and is enabled by default.

BBT should be placed



Decides, whether BBT and mirror BBT should be placed automatically (default setting) or at specified page. This option corresponds to NAND_BBT_ABSPAGE option in driver.

If BBT should be placed automatically, following two options are taken into account:

BBT should be placed starting from

Decides, whether BBT and mirror BBT should be placed starting from the last block (default setting) or from the first block in device or chip (dependent on BBT should be stored setting). This option corresponds to NAND_BBT_LASTBLOCK option in driver.

If BBT should be placed automatically and starting from device end, the first good block from device end (last good block) is used for BBT and the preceding one (last-1 good block) is used for mirror BBT. Example for device with 1024 blocks: block #1023 will be used for BBT and block #1022 will be used for mirror BBT.

If BBT should be placed automatically and starting from device start, the first good block in device will be used for BBT and the second good block will be used for mirror BBT. Example for device with 1024 blocks: block #0000 will be used for BBT and block #0001 will be used for mirror BBT.

If BBT should be stored on per chip basis, the situation will be similar, but consider each chip individually instead of device.

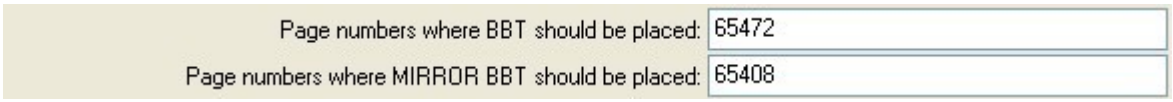
Number of blocks reserved for BBT

Specifies the number of blocks reserved for BBT and mirror BBT storage. These blocks are treated like being invalid when programming partition data and don't need to be considered when specifying the partitions boundaries (except of adding some more padding blocks). This option corresponds to NAND_BBT_SCAN_MAXBLOCKS option in driver and is set to 4 by default.

If BBT should be placed at specified page, following two options are taken into account:

Page numbers where BBT should be placed

Page numbers where MIRROR BBT should be placed

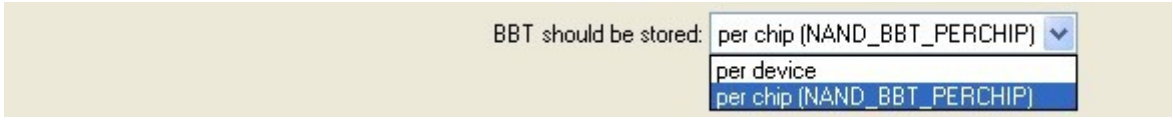


Specify, where BBT or mirror BBT should be stored, respectively. These options correspond to parameter nand_bbt_descr→pages[ ]. Depending on BBT should be stored option setting, there must be specified for both options:

- one page in range of device, if BBT should be stored per device is set,
- one page in range of chip for each chip in device, if BBT should be stored per chip is set.

By default, these settings correspond with the situation of automatic placement from device end as described above for option BBT should be placed starting from.

BBT should be stored



Specifies, whether there will be stored only one BBT and mirror BBT pair covering whole device, or each chip in device will carry its own BBT and mirror BBT pair (default setting). This option corresponds to NAND_BBT_PERCHIP option in driver.

Store BBT version counter

BBT version counter value

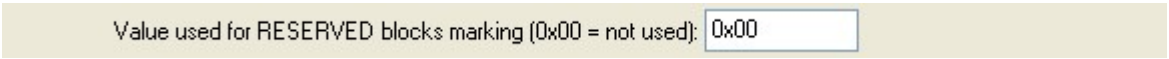
Enables BBT versioning. If Store BBT version counter is enabled, BBT version counter value is stored in spare area of the first page of BBT and/or mirror BBT. By default, the versioning is enabled and the counter is set to 0 (zero). These options correspond to NAND_BBT_VERSION option and parameter nand_bbt_descr→version in driver.

Number of bits used per block in BBT on device

Specifies the number of bits that represent single block in flash-based BBT and/or mirror BBT. Linux MTD uses RAM-based BBT with 2 bits per block, however, it enables to change the representation when storing the BBT and/or mirror BBT to nand flash device. Four options are available, corresponding to driver:

- 1 bit (corresponds to NAND_BBT_1BIT in driver)
- 2 bits - default setting (corresponds to NAND_BBT_2BIT in driver)
- 4 bits (corresponds to NAND_BBT_4BIT in driver)
- 8 bits (corresponds to NAND_BBT_8BIT in driver)

Value used for RESERVED blocks marking



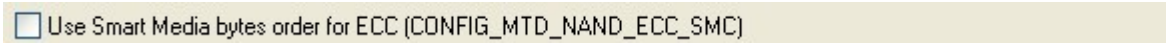
Reserved blocks are the blocks used for BBT and mirror BBT storage. To prevent the system from using them for another purpose, they are marked to be reserved rather than invalid. The value specified here will be used for reserved block marking in nand flash based BBT and/or mirror BBT.

Consider that only few bits may be used depending on number of bits per block setting.

The value of 0x00 (default setting) means that this option is not used and reserved block will be marked in the same way as good block. See driver for more information on values used.

This option corresponds to parameter nand_bbt_descr→reserved_block_code in driver.

Use Smart Media bytes order for ECC



Specifies Smart Media bytes ordering for ECC control sums. Both, Linux MTD and Smart Media cards use the same ECC algorithm, but ECC[0] and ECC[1] bytes are permuted (ordered reciprocally).

This option corresponds to CONFIG_MTD_NAND_ECC_SMC option in driver and is disabled by default. The option is common for both, BBT as well as partitions data (if applied).

Apply MTD specific ECC on partitions data

Enables Linux MTD specific ECC algorithm usage also for user data in partitions. Linux MTD is capable to write its own specific ECC control sums into locations specified in driver. All other spare area locations are left untouched. The option is disabled by default.

USING COMMA SEPARATED VALUES FORMAT

This mode uses two input files. You can simply prepare both of them on your own:

Partition table definition file

Partition table definition file uses widely used comma separated values file format.

The file should contain the number of rows corresponding to the number of partitions. Each row specifies one partition.

Values in row should be separated by separator - comma (,) or semicolon (;) can be used. Space characters (ASCII code 0x20) are ignored and should not be used in place of values separator.

Each row should contain several values (both, decimal and/or hexadecimal values can be used):

- **partition start** (mandatory) - specifies the block in device, where partition should start. Enter the block number here.
- **partition end** (mandatory) - specifies the block in device, where partition should end. Enter the block number here.
- **used partition size** (mandatory) - specifies the number of blocks really occupied by partition data. Typically, there are some reserve blocks added for invalid blocks replacement, therefore partition end - partition start > used partition size. Enter number of blocks here.

– **special options/reserved** (optional/mandatory) - this value enables to specify some special options. If you use it just due to **comment** option, enter the value of 0xFFFFFFFF (4 bytes size) here to ensure future compatibility.

Special options format specification:

```
MSB(bit 31)                    LSB(bit 0)

xxxx.xxxx.xxxx.xxxx.xxxx.xxxx.xxxx.xxxx
```

bits 11:0 - Maximum allowed number of invalid blocks in partition:

0xFFF = option disabled

any other value specifies the number of invalid blocks that can be accepted in partition

bits 15:12 - Invalid blocks management method:

0x0 = Treat All Blocks

0x1 or 0xFF = Skip IB (default)

0x2 = Skip IB with Excess Abandon

0x3 = Check IB without Access

Note: Check IB with Skip IB can be specified using Skip IB (0x1 or 0xFF) method and non 0xFFF value for Max. allowed number of invalid blocks in partition.

bits 23:16 - reserved for future use, consider 0xFF value for future compatibility

bits 31:24 - File system preparation:

0xFF = option not used

0x00 = JFFS2 Clean Markers using MSB byte ordering (big endian)

0x01 = JFFS2 Clean Markers using LSB byte ordering (little endian)

Using values not specified here will cause partition table load error.

– **comment** (optional) - you can enter any text here. Primarily, this item is intended for your notes that will help you to orientate in the file. It may contain e. g. partition name. If you use comments, **reserved** item must be also used.

Partition table definition file listening example:

```
0;100;20;0xffffffff;boot
101;200;50;0xffffffff;exec
201;300;0;0xffffffff;res1
301;400;50;0xffffffff;fsys
401;500;0;0xffffffff;res2
501;1000;50;0xffffffff;data
```

Load this partition table definition file using menu **File >> Load Partition table...**, see chapter Loading partition table definition file.

It is possible to save your partition table definition using this format. To save the partition table data, use menu **File >> Save Partition table...,** see Figure 4. The table is saved using all items in raw. A partition number is used for comment.

Saved partition table definition file listening, using the above example:

```
0;100;20;0xFFFFFFFF;Partition 0
101;200;50;0xFFFFFFFF;Partition 1
201;300;0;0xFFFFFFFF;Partition 2
301;400;50;0xFFFFFFFF;Partition 3
401;500;0;0xFFFFFFFF;Partition 4
501;1000;50;0xFFFFFFFF;Partition 5
```

Data image file

Input data image file should be a binary file (recommended) that meets all requirements specified in chapter Loading data image file. Use standard Load procedure to load this file (see chapter Loading data image file).

HISTORY

Version 1.00 - February 2010

– initial release

Version 1.01 - June 2010

– Special options added for CSV partition definition file (max. allowed number of invalid blocks in partition, invalid blocks management method, file system preparation)

Version 1.02 - September 2010

– added passage on loading multiple input data files

Version 1.03 - February 2011

– some minor changes in values alignment for CSV format description