Programming NAND Flash Memories

on ELNEC

Universal Device Programmers

Application Note

See also Elnec application note: NAND Flash Memories

January 2014
an_elnec_nandflash, version 0.2

Please, read carefully:

This application note describes how to program nand flash devices using ELNEC universal device programmers. Before reading this document, user should be familiarized with nand flash devices. There are plentiful sources available through the web containing detailed informations about nand flash internal organization, errors in nand flash, basic algorithms, etc. Study, please, your device datasheet thoroughly, at least.

This application note is provided by our technical support department to help our customers and is provided "as-is", without warranty of any kind, either expressed or implied. We reserve the rights to make changes to the information available in this application note at any time and assume no liability for applications assistance, customer product design and any damages arising from the use of this application note.

CONTENT

BRIEF COMMENTS ON INVALID BLOCKS

- Invalid block (in various sources may be referred also as “bad block” or “damaged block”) is a block that contains one or more permanently damaged memory cells.
- Invalid block occurrence doesn't affect the function of other blocks in device.
- There may be invalid blocks yet in new (not used before) device. Other invalid blocks may develop over time.
- Invalid block shouldn't be used for programming – data may be lost.
- Invalid block shouldn't be erased – information about its invalidity may be lost.
- There is BI byte somewhere in a block. Its location is specified by device manufacturer. For SLC devices, it is typically in first spare area byte within first and/or second page in a block. For MLC devices, it is typically in first spare area byte within first and/or last page in a block. But other locations are also used.
- Before any operation with device, all blocks must be screened for BI bytes values. This process is so-called *Invalid blocks map building*. Typical flowchart:

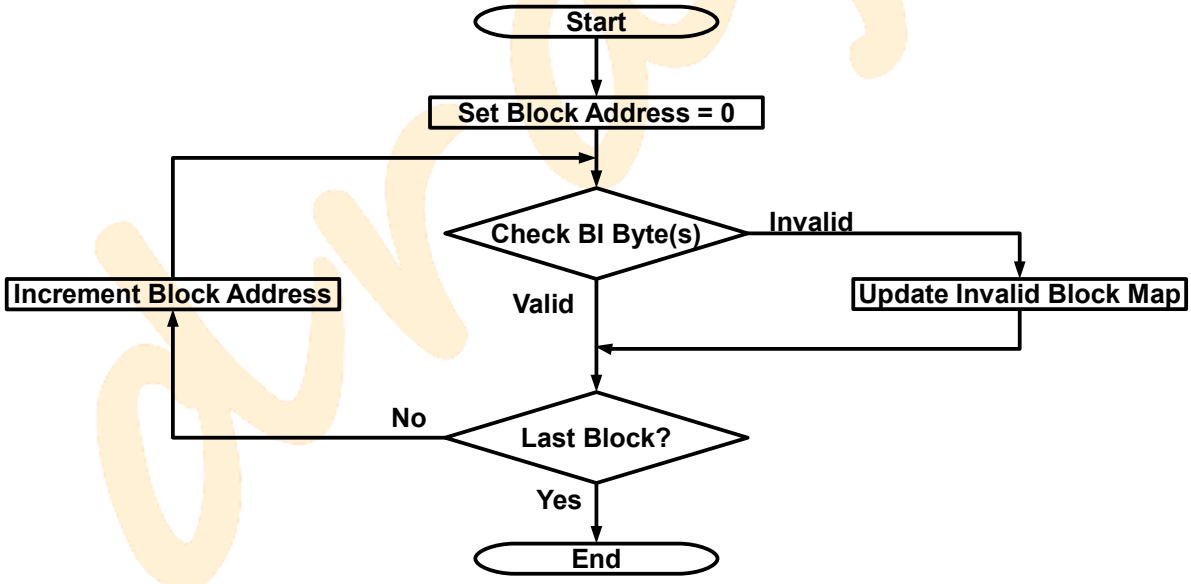


Figure 1: Example of typical invalid blocks map building flowchart.

- There are software techniques generally called *invalid blocks management* used for treatment of existing invalid blocks. These techniques are relevant to know before pre-programming nand flash device.
- There are software techniques generally called *wear management* used for new invalid blocks development prevention. These techniques are used during end-appliance usage and, usually, are not relevant to know before pre-programming nand flash device.

BRIEF COMMENTS ON BIT ERRORS

- Bit errors are temporary errors. Typically, they appear on read only, and disappear after erase. Otherwise, respective block must be considered invalid.
- Bit errors are native to nand flash memories. They can be considered to be a drawback of nand flash technology. Typically, they occur due to an influence between adjacent memory cells. Please, refer to general nand flash materials for more details on this topic.
- Bit errors may be detected and recovered. Various ECC algorithms are used for this purpose. Typical representatives are Hamming algorithm, BCH (Bose – Chaudhuri – Hocquenghem) algorithm, and RS (Reed – Solomon) algorithm.
- Individual ECC algorithms may be distinguished using several basic characteristics: the frame size (a number of bytes/words covered by single application of the algorithm), the strength (a number of bit errors that can be recovered in the frame of specified size) and the number of control bits/bytes (a size of overhead data).
- For each nand flash device, the manufacturer specifies how many bit errors are “normal” for a data frame of some size (e. g. 4 bit errors per 512 bytes). At least, an ECC algorithm capable to recover specified number of bit errors over specified frame size must be used.
- Our programmers can support selected ECC algorithms. In addition, we offer customized implementations that may support any ECC algorithm specified by customer. Also, a generalized solution is also available – on verify, the programmer may accept specified number of bit errors in specified number of bytes and suppose, that these bit errors will be corrected by ECC algorithm in real application – see chapter Tolerant Verify.

TWO FACTORS THAT PROGRAMMER RELIES ON

- The user: Programmer will do only what user has instructed its. Programmer can detect device boundary exceeding, but cannot foretell e. g. a block from where data should start. Please, don't rely on default settings. Those are just some general preferences originating from device parameters and algorithm simplifying rather than from your particular needs.
- NAND device internal controller: The controller communicates with programmer via STATUS register. On erase, the controller checks if all memory cells in a block are in erased state. If controller says that the block is erased properly, programmer will rely on this information – none (significant time consuming) blank check is performed after erase. If controller says that the block is not erased properly, programmer will consider that block invalid – the block is treated regarding to selected invalid block management. On programming, the controller checks if all page locations expected to be in 0 are really in 0. If controller says that the page is programmed properly, programmer will continue with next page. If controller says that the page is not programmed properly, programmer will consider related block invalid – the block is treated regarding to selected invalid block management.

DATA ORGANIZATION IN *PG4UW* CONTROL SOFTWARE BUFFER

Data are stored in buffer as a continuous sequence of pages. Please, be aware of fact, that page spare area is not included in normal device addressing. Control software buffer, however, uses linear addressing. This may lead to hazardous misunderstandings resulting in incorrect data positioning in device. Compare, please, following pictures:

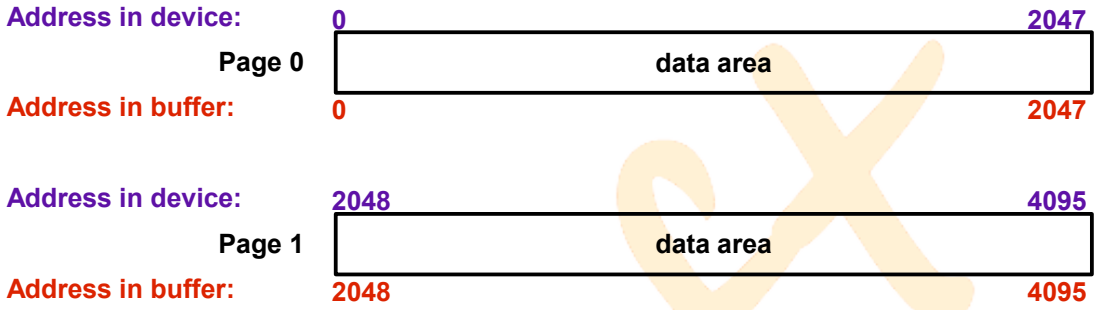


Figure 2: Buffer data layout if spare area is not used.

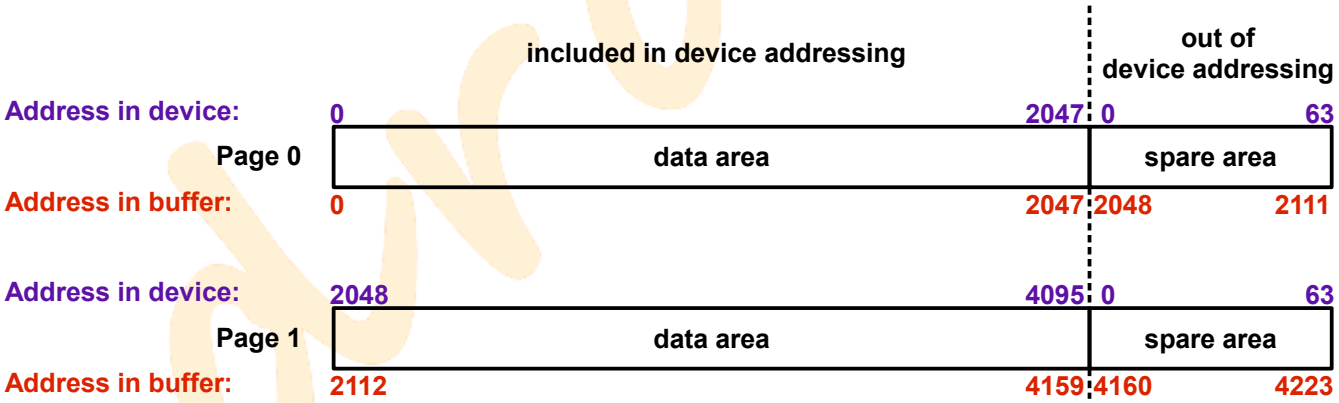


Figure 3: Buffer data layout if spare area is used.

Considering a common nand flash device with 2048+64 bytes in a page and 64 pages in a block, the first byte of second block in device will be addressed using offset 0x20000 in device, but using offset 0x21000 in buffer. It is crucial to keep this in mind, especially if working with partitions.

LOADING DATA INTO *PG4UW* CONTROL SOFTWARE BUFFER

Primarily, command **File >> Load** (short-cut <F3>) should be used for input image loading into buffer. Software can recognize plentiful data formats, however, for devices with capacity of 16 Gbit and more only raw binary mode (*.BIN) may be supported.

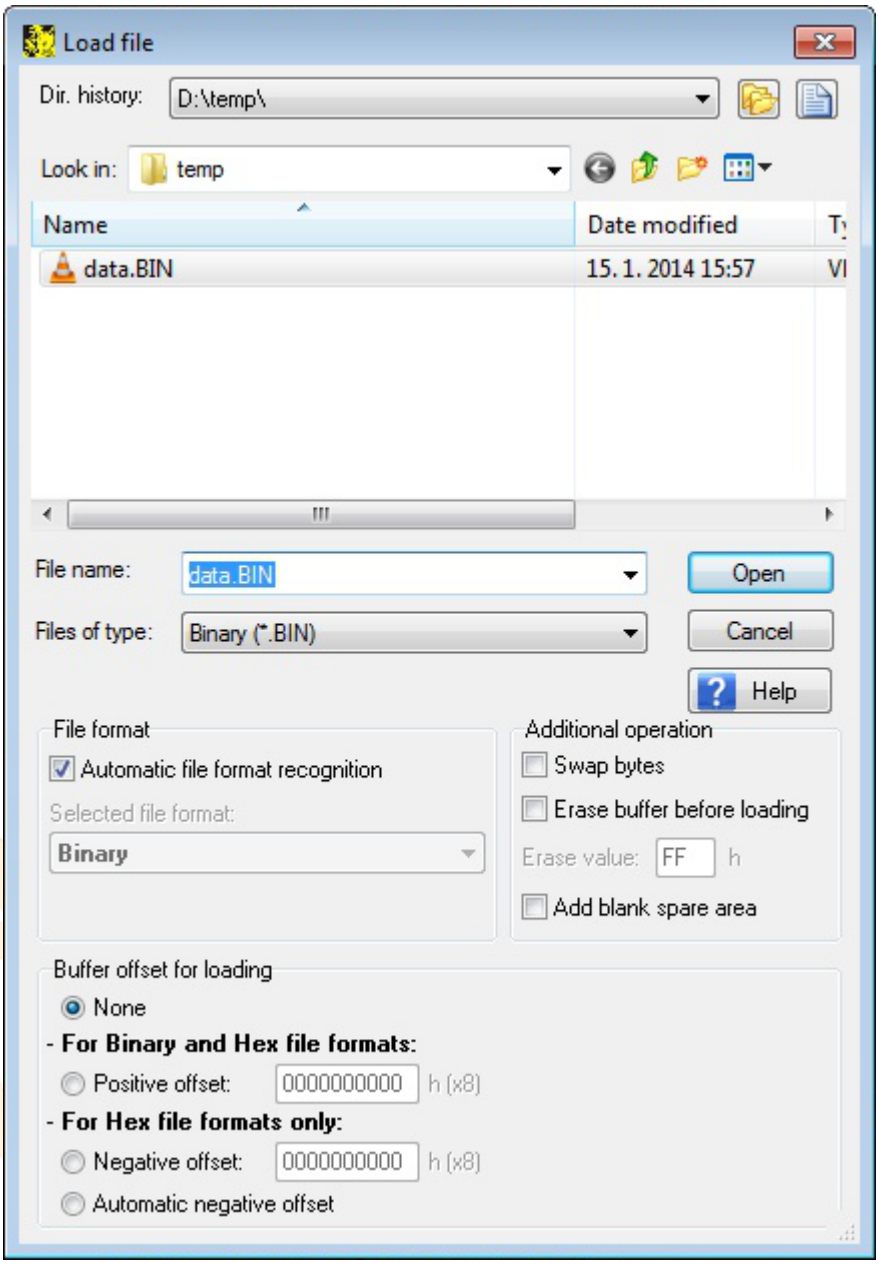


Figure 4: Load file dialog window.

Data image file should correspond with a copy of NAND flash device without any invalid blocks. Depending on other settings, it must or must not contain also spare area data. If selected mode requires spare area data and your image doesn't contain it (relevant mainly for partitioning techniques), you can add blank (all 0xFF) spare area automatically on image load allowing **Add blank spare area** option (see Figure 4, *Additional operation*

panel). It is important to select correct device firstly, since various devices may use different data area and spare area sizes and control software always matches page layout of actually selected device. All other options available in **Load File** window work in their usual way.

LOADING MULTIPLE DATA IMAGES

If you need to load multiple data image files for single device (relevant mainly for partitioning techniques), you may need to employ **Positive offset** option (see Figure 4, *Buffer offset for loading* panel). You may compute the offset using following formula:

positive_offset = target_block_number_in_buffer x number_of_pages_in_block x page_size

where:

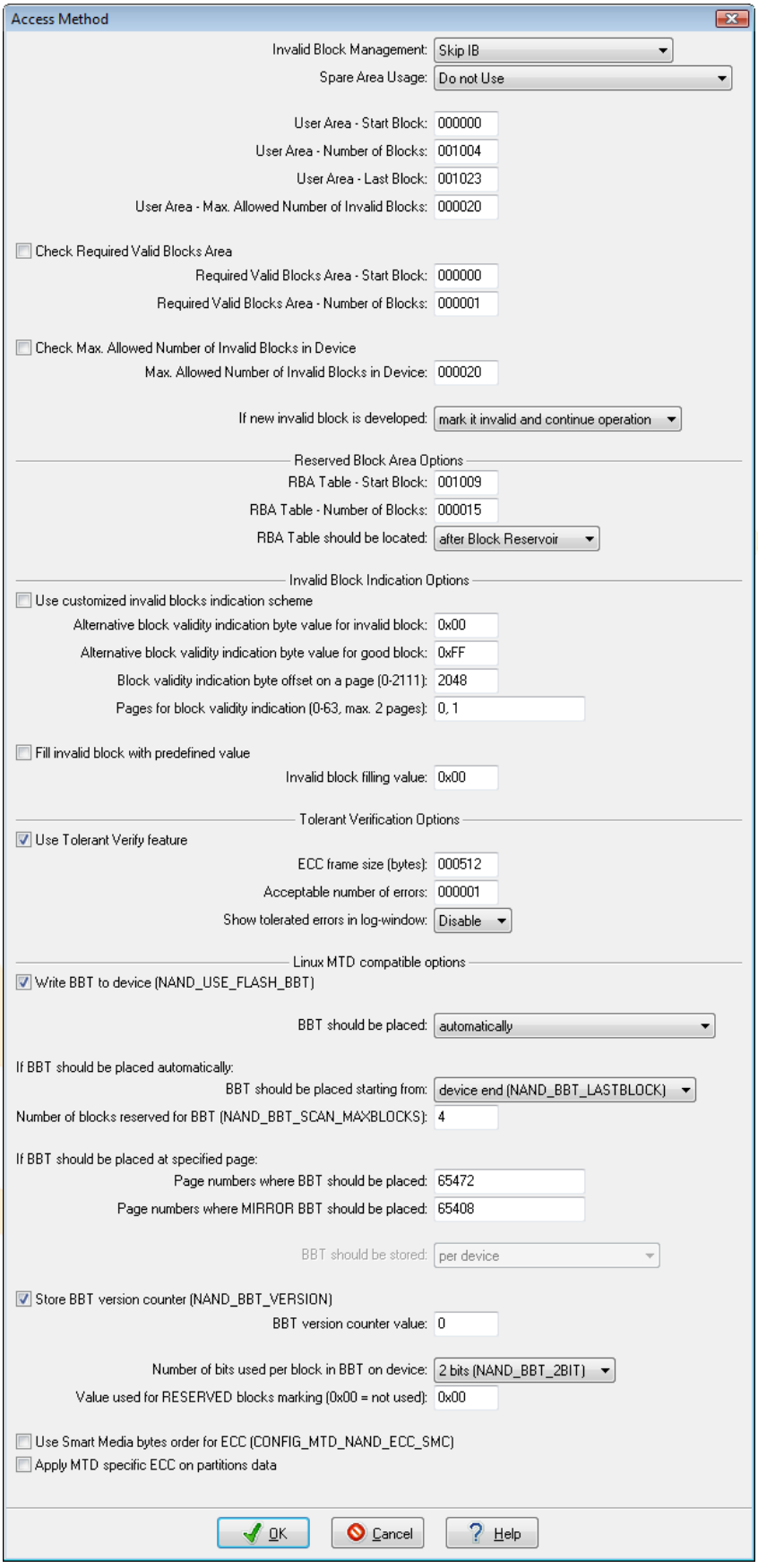
target_block_number_in_buffer is the number of target block as is mapped in buffer. Blocks ordering in buffer may differ from their real ordering in device, see buffer to device mapping in chapter dedicated to respective invalid blocks management technique.

number_of_pages_in_block is the count of pages in one block, as is given in your nand device datasheet.

page_size is the size of a page in bytes or words (for x8 or x16 devices, respectively), as is given in your nand device datasheet. The page size must, or must not include spare area size, depending on other settings.

This way you may load all your data images, file after file, and place them at correct locations in buffer.

ACCESS METHOD WINDOW



INVALID BLOCKS MANAGEMENT

Our programmers support several general invalid blocks management techniques. Not all methods described here are supported on all programmers. Any other invalid blocks management technique can be supported upon user's request.

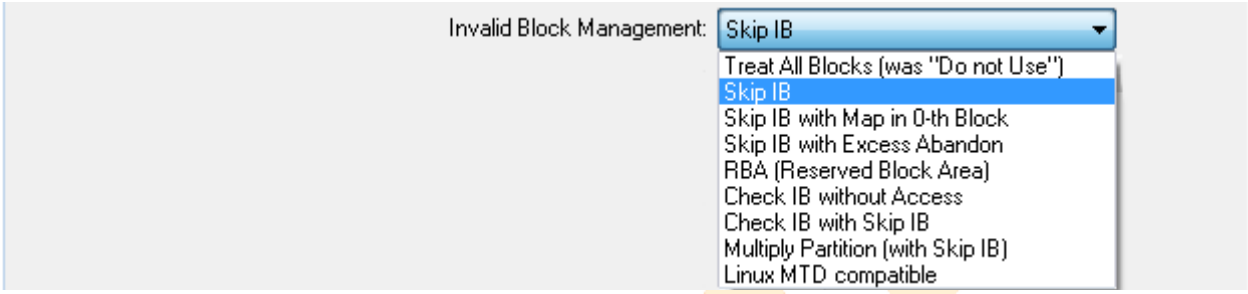


Figure 5: Invalid blocks management options.

TREAT ALL BLOCKS

In past, we called this technique “Do not Use”, simply because none block validity related decision algorithm is used. All blocks in device are processed equally, not regarding their real validity status.

The technique may be very helpful if dumping of unknown data is necessary, e. g. for data recovery from broken USB stick. It allows to create the image comprising all blocks in device for further analysis.

Proceed with caution!

Since this technique doesn't differentiate between valid and invalid blocks, you can suffer a damage!

On programming, programmer will try to write data also to invalid blocks. The operation will fail on verify after programming (if enabled), however, if device is even thought used in end appliance, it may cause its malfunction.

On erase, programmer will try to erase also invalid blocks. This may damage BI bytes in invalid blocks, so information about their invalidity might be lost. Programmer is rather simple device not capable to perform any reliability tests similar to those one on manufacturing line, so it cannot recover this information.

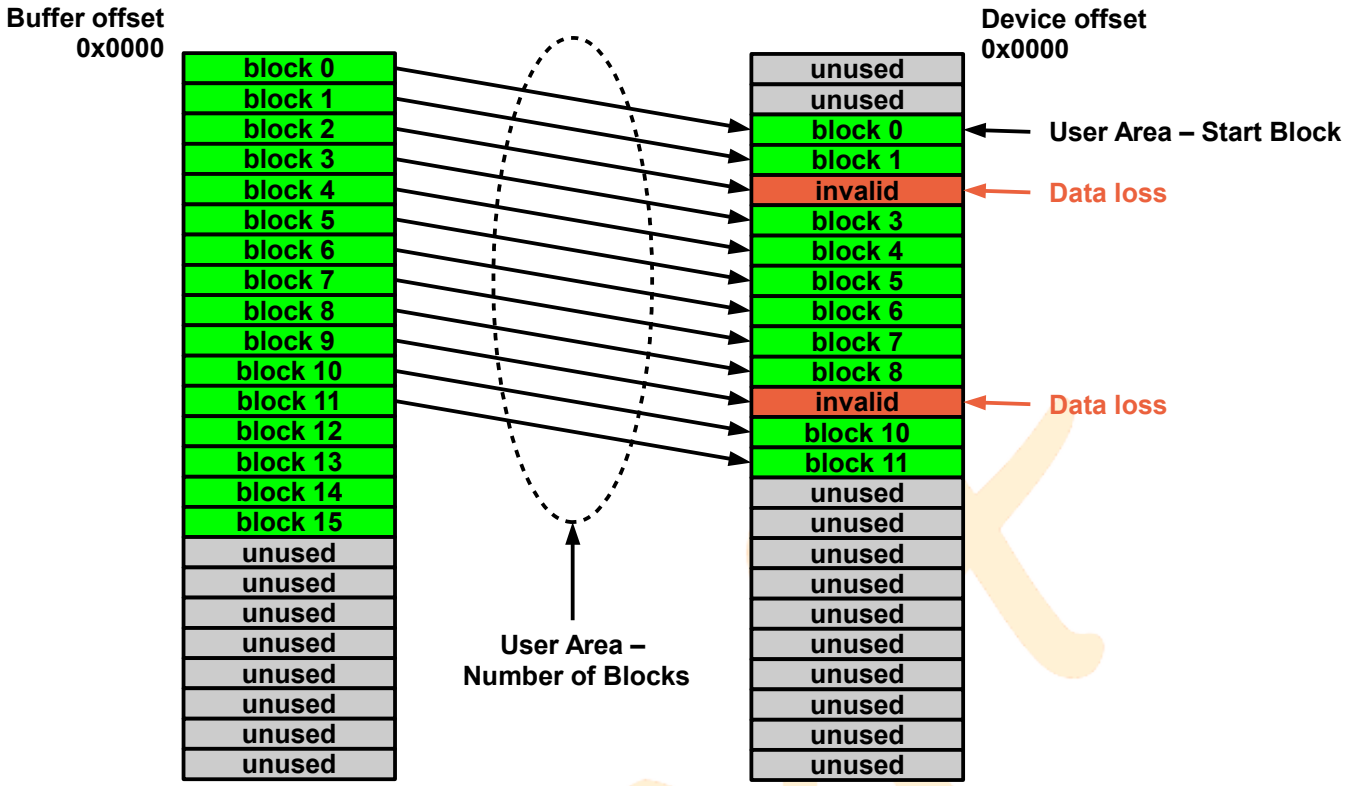


*Figure 6: **Treat all blocks** technique graphic representation.*

Using **Treat All Blocks** technique, a number of blocks specified in option **User Area – Number of Blocks** will be taken counting from buffer start, and programmed into device starting from a block specified in option **User Area – Start Block**. These blocks will be programmed in device, not regarding the blocks validity. If target block is invalid, data will be lost. The number of blocks specified for processing is not necessary equal to the size of data loaded in buffer.

On device read, reciprocally, a number of blocks specified in option **User Area – Number of Blocks** will be read from device starting from a block specified in option **User Area – Start Block**, not taking source blocks validity into account, and stored into buffer counting from buffer start.

SKIP IB

This is the simplest technique used for treatment of invalid blocks. If target block is invalid, it is skipped and next valid block is used instead. The next data are then programmed into (next+1)th block. This will produce a shift in data offset. The shift increases with each skipped invalid block. If there are too many invalid blocks in target device area, not all data might be programmed. The overflow data would be lost, therefore operation is halted at first moment when such condition is recognized (typically on initial Invalid Blocks Map building).

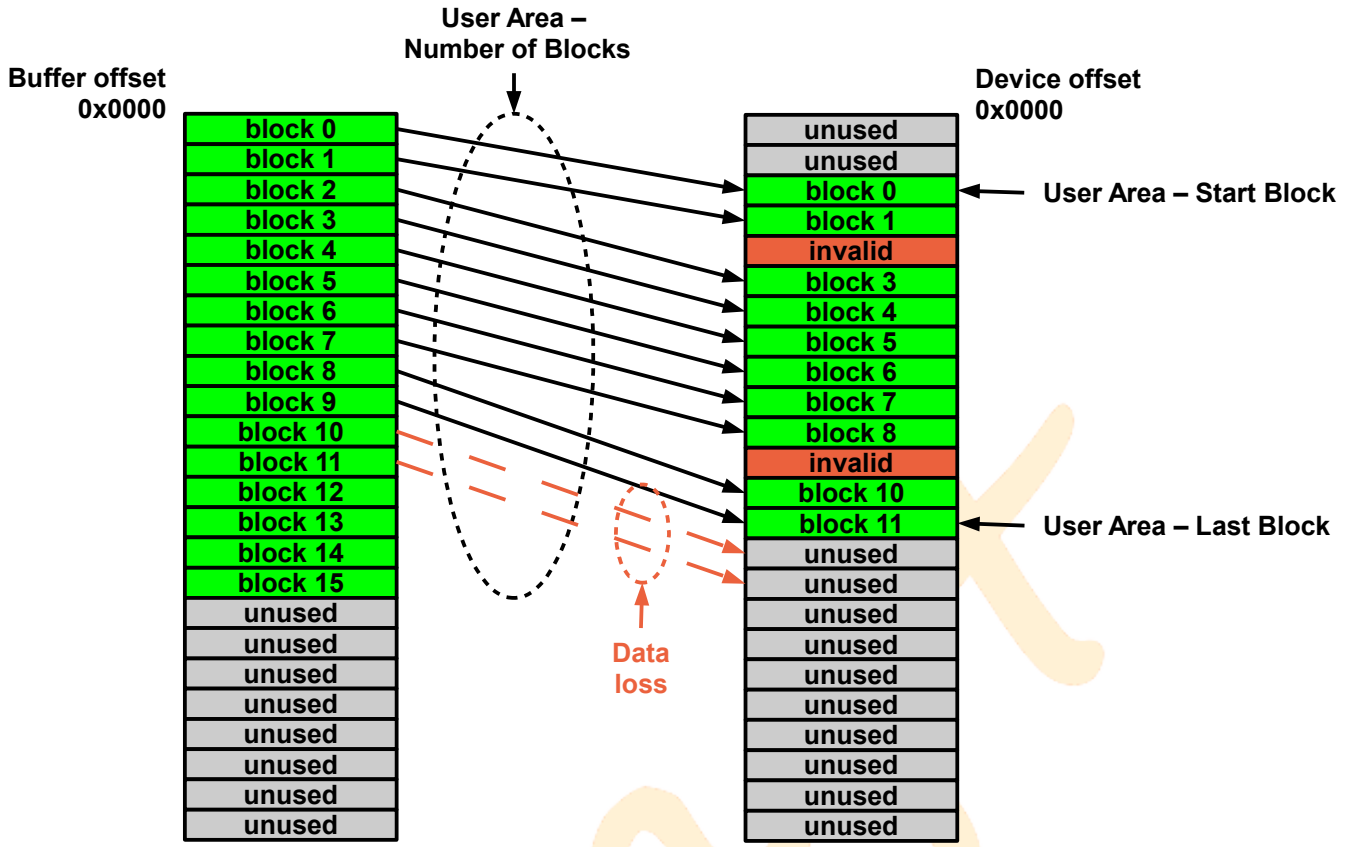


*Figure 7: **Skip IB** technique graphic representation.*

Using **Skip IB** technique, a number of blocks specified in option **User Area – Number of Blocks** will be taken counting from buffer start, and programmed into device starting from a block specified in option **User Area – Start Block**. If target block is invalid, actual data will be programmed into next valid block, thus shifting all next data by offset of one block. The number of blocks specified for processing is not necessary equal to the size of data loaded in buffer. If a block specified in option **User Area – Last Block** is reached and not all specified blocks are programmed, operation is halted with error.

On device read, reciprocally, a number of blocks specified in option **User Area – Number of Blocks** will be read from device starting from a block specified in option **User Area – Start Block**. Read data will be stored into buffer counting from buffer start. If source block is invalid, it will be skipped (not processed) and programmer will continue with next valid block. Data are stored in buffer continually, without gaps from invalid blocks, so the same image will be created in buffer not regarding invalid blocks distribution over the device. If a block specified in option **User Area – Last Block** is reached and not all specified blocks are read, operation will close with warning.

SKIP IB WITH MAP IN 0TH BLOCK

This technique is for backward compatibility with algorithms developed for old nand flash devices.

Very first nand flash memories came without spare area, so it was not possible to store BI byte nor any other validity mark out of payload data. Initial invalid blocks were forced to all zeros state. But after first device programming, it was not possible to distinguish, whether the block is invalid or programmed with all zeros intentionally (e. g. some variables initialization section). One of used solutions consisted in programming Invalid Blocks Map in first device block (block #0000). All other behaviour is the same as for **Skip IB** technique.

The map uses one bit value to store information about one block. Bit 0 of Byte 0 corresponds to block #0000, bit 1 of Byte 0 corresponds to block #0001, …, bit 0 of Byte 1 corresponds to block #0008, and so on until the device end. If bit value = 1 then corresponding block is invalid.

You can display the same Invalid Blocks Map using menu command **Buffer / View/Edit Buffer** (short-cut <F4>) and then clicking on Invalid Blocks Map tab.

SKIP IB WITH EXCESS ABANDON

This technique is very close to basic **Skip IB** technique, too. Recall, please, a possible data loss due to excessive invalid blocks count in specified area. **Skip IB with excess abandon** technique doesn't generate error if this lossy condition happen. Data that cannot be programmed will be simply abandoned (lost).

This technique may be very useful for applications where multiple data copies are used as a mean of error protection. Typical example is a bootloader storage. First-stage bootloader stored in processor's internal flash/ROM memory may try to load second-stage bootloader from nand flash memory block #0000. On failure, it continues with block #0001, and so on until some limit. Using e. g. 8 bootloader copies, its availability is guaranteed over long time.

Compared to **Treat all blocks** technique, **Skip IB with excess abandon** skips invalid blocks, so programmer doesn't expect any data there and verify operation can still succeed.

RBA (RESERVED BLOCK AREA)

This is an another kind of invalid blocks management technique, based on replacement of invalid blocks.

Using this approach, the device is subdivided into three regions – user data area, reservoir of blocks for replacement of invalid blocks from user data area, and an area reserved for redirection table (sometimes referred also as table of substitutions). Normally, data are programmed into user data area. If target block is invalid, next free valid block from reservoir is used instead. Redirection table is updated by new invalid-valid pair of blocks. Process then continues with next block data and next block in user data area. After programming all required blocks, redirection table is programmed into the area reserved for this purpose. In addition to information about redirected block pairs, the table may also contain other kinds of data, like some identification header, version numbering, device parameters information, etc.

Reserved block area technique, as is implemented in our programmers, is based on Samsung's algorithm and works as is described in following paragraphs. You can exactly specify two areas of three in use – user area, where data should be stored primarily; and RBA Table area, where redirection table should be stored. Reservoir is created automatically, based on setting of option **RBA Table should be located**, see Figure 8.

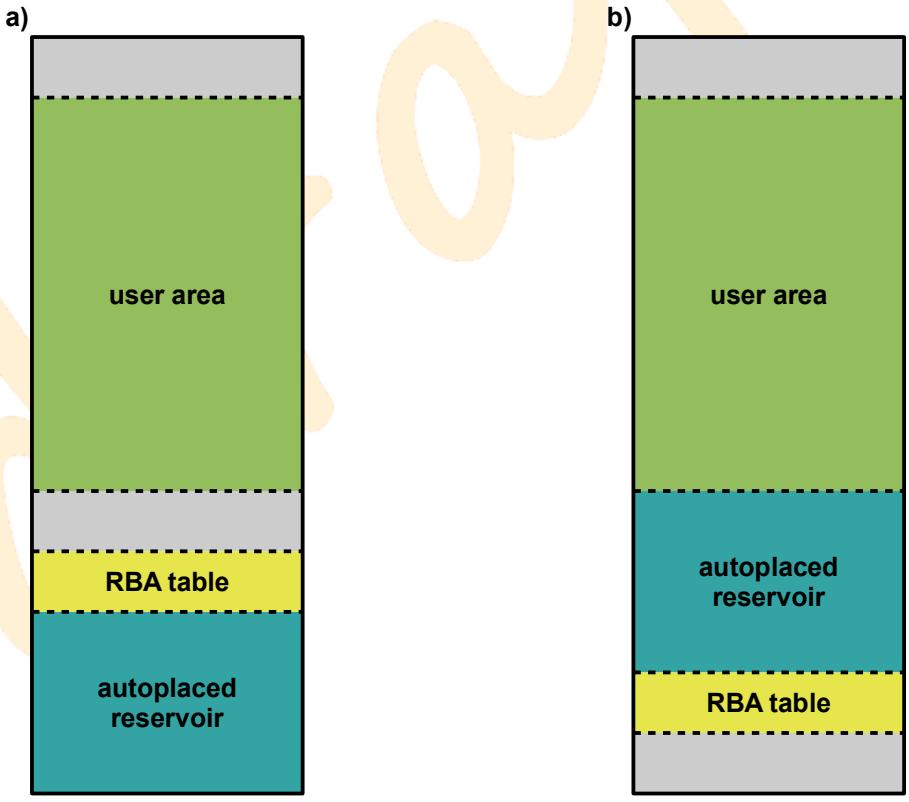


*Figure 8: Device layout depending of **RBA Table should be located** option value: **before Block Reservoir** (a) and **after Block reservoir** (b). There may be unused blocks accepted in grey areas.*

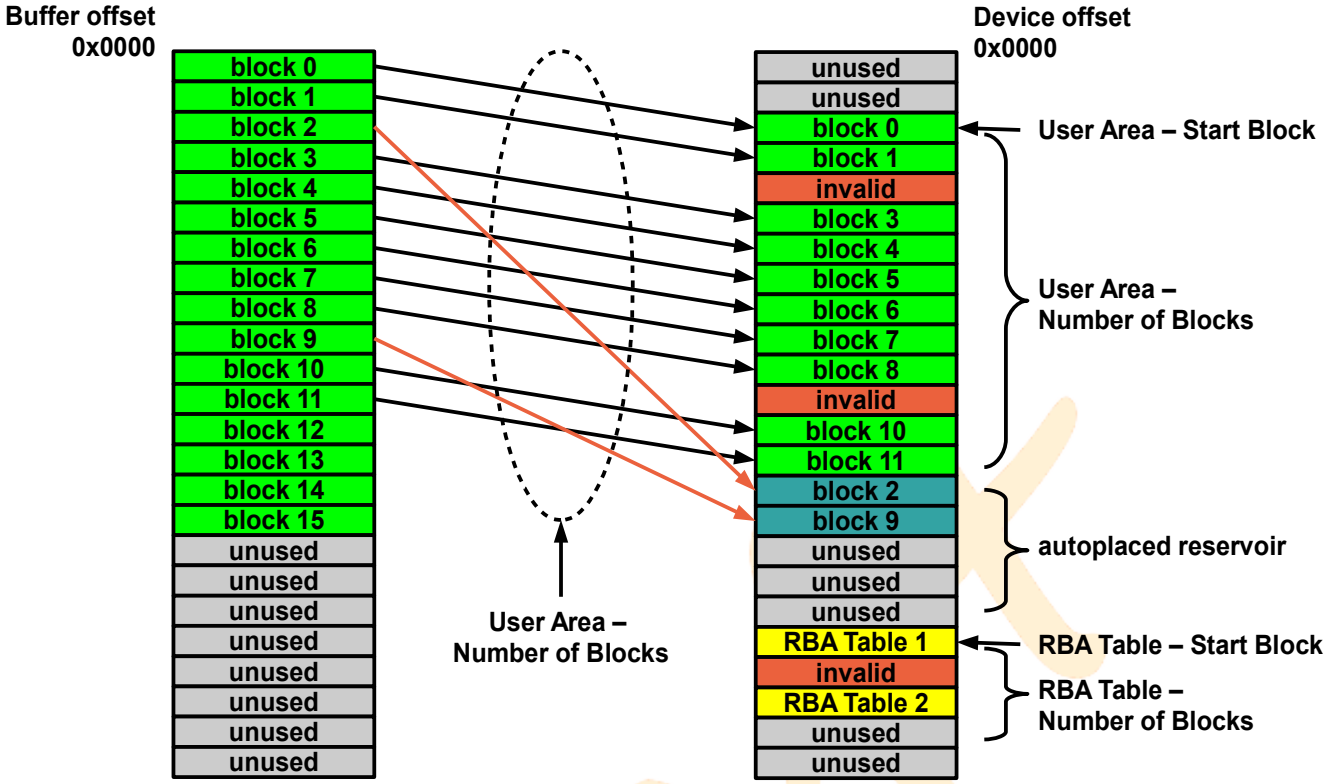


*Figure 9: **RBA** technique graphic representation.*

Figure 9 illustrates buffer data to physical blocks assignment on example where RBA Table should be located after reservoir. In the other case, the principle of blocks substitution will be the same, just areas allocation will differ, see Figure 8.

On programming:

A number of blocks specified in option **User Area – Number of Blocks** will be allocated for user data area, starting from block specified in option **User Area – Start Block**. Another number of blocks specified in option **RBA Table – Number of Blocks** will be allocated for redirection table, starting from block specified in option **RBA Table – Start Block**. If **RBA Table should be located** = **before Block Reservoir**, all free blocks between redirection table area and device end will be allocated for block reservoir. If **RBA Table should be located** = **after Block Reservoir**, all free blocks between user data area and redirection table area will be allocated for block reservoir.

A number of blocks specified in option **User Area – Number of Blocks** will be taken counting from buffer start and programmed into user data area in device, block by block. If target block is invalid, next free valid block from block reservoir will be used instead. Blocks are picked-up from block reservoir in ascending order (from device start towards device end), invalid blocks are not used. Redirection table in programmer memory will be updated. If there are more invalid blocks in user data area than valid blocks in block reservoir, data loss will occur. In such

case, operation will be halted with error.

After programming specified number of data blocks, two copies (original and back-up) of redirection table (RBA Table) will be programmed into redirection table area. Skip IB technique is used. If there are less than two valid blocks in redirection table area, those two copies cannot be programmed and operation will be halted with error.

On read:

A number of blocks specified in option **User Area – Number of Blocks** will be allocated for user data area, starting from block specified in option **User Area – Start Block**. Another number of blocks specified in option **RBA Table – Number of Blocks** will be allocated for redirection table, starting from block specified in option **RBA Table – Start Block**. If **RBA Table should be located** = **before Block Reservoir**, all free blocks between redirection table area and device end will be allocated for block reservoir. If **RBA Table should be located** = **after Block Reservoir**, all free blocks between user data area and redirection table area will be allocated for block reservoir.

Redirection table area will be searched for at least one valid copy of redirection table. If valid redirection table is not found, operation is halted with error.

After successful RBA Table decoding, a number of blocks specified in option **User Area – Number of Blocks** will be read from user data area and stored into buffer counting from buffer start. If source block is listed in redirection table, its substitutive block will be read instead. If it is not possible to read specified number of blocks from user data area + block reservoir, operation will close with warning.

Redirection table format:

RBA Table consists of pages. Each page uses the same data field layout. Each data field is 16 bit wide, stored using little endian format.

The first data field on a page is *header*. The header is always of the same value 0xFDFE.

The second data field on a page is *count field*. Count field stores page sequence number, counting from 1 for first page of first RBA Table copy and incrementing by one for each other page. For second RBA table copy, the counter continues incrementing (if table uses e. g. 4 pages, count field value for first page of second RBA Table copy will be 5).

Further, a page continues with *invalid block – replacement block* data field pairs. These pairs store numbers of invalid blocks from user data area and theirs respective substitution blocks from blocks reservoir used for replacement. Single page can hold information about (page_data_area_size – 4)/4 redirections.

Unused bytes in RBA Table block are set to blank state 0xFF.

CHECK IB WITHOUT ACCESS

Note: This invalid blocks management technique is available on selected programmers and/or devices only.

Check IB without access technique performs checks with regard to set rules, but doesn't execute any other access. For that reason, only programming command is available after confirming this technique. The command is used for running the tests.

This technique may be used for programming simulation, e. g. if you are going to do initial nand flash device programming after the end-appliance assembly. In such case you may be interested in not using devices with too much invalid blocks for assembly (thus minimizing the waste due to memory units of poor quality).

An area starting from block specified in option **User Area – Start Block** up to block specified in option **User Area – Last Block** is scanned for invalid blocks.

If the count of invalid blocks in that area exceeds a number specified in option **User Area – Max. Allowed Number of Invalid Blocks**, an error is reported.

If the count of valid blocks in that area is less than a number specified in option **User Area – Number of Blocks**, an error is reported.

If enabled, **required valid blocks** area is checked.

If enabled, **maximum allowed number of invalid blocks in device** is checked.

CHECK IB WITH SKIP IB

Note: This invalid blocks management technique is available on selected programmers and/or devices only.

After performing all tests in the same manner like if **Check IB without access** technique would be used, **Skip IB** technique will be used for accessing the device.

This technique may be helpful if you need to guarantee some number of unused valid blocks in user data area. E. g. if you need to program 80 blocks into area of 100 blocks, standard **Skip IB** technique will accept 20 invalid blocks in that area. But using **Check IB with Skip IB** technique, you may allow acceptance of only e. g. 10 invalid blocks. Remaining 10 valid blocks may be used for further invalid blocks replacement during end-appliance lifetime.

MULTIPLE PARTITIONS WITH SKIP IB

Note: This invalid blocks management technique is available for selected programmers and/or devices only. It offers wide range of options that were implemented in successive steps. If enabled for discontinued programmers, it is called **Qualcomm Multiple Partition** (historical reason).

In very simple words, this is **Skip IB** technique extended for allowance of multiple user data areas. This comes with variety of new possibilities, but also with more complicated configuration and handling.

User data area is now called **partition**. All user data area settings have respective partition equivalent:

- **User Area – Start Block → Partition start**
- **User Area – Number of Blocks → Used partition size**
- **User Area – Last Block → Partition end**

There are several significant differences from **Skip IB** technique:

- Spare area data are always expected in buffer. If you don't use spare area, you may fill respective areas by blank data on input image file load by enabling **Add blank spare area** feature, see chapter **Loading data into pg4uw control software buffer**.
- Partition start data in buffer are now expected with the same offset as in device, i. e. **Partition start** value specifies the partition beginning in both, device and buffer.
- Instead of specifying necessary options in **Access Method** window, partitions are specified via **Partition definition file**.
- Only some of options available in **Access Method** window will be accepted during operation, see chapter **Access Method window options validity in partitioning mode**.
- Probably, you will need to load several input data images, see chapter **Loading multiple data images**.

Using **Multiple Partition with Skip IB** technique, programmer will process each partition individually, in increasing order. After programming or reading a partition, the same partition is verified (if enabled). Only after then, if succeeded, the programmer will continue with next partition.

A number of blocks specified by value of **Used partition size** will be taken from buffer counting from a block specified by value of **Partition start**. These data will be programmed into device starting from a block specified by value of **Partition start**, too. If target block is invalid, actual data will be programmed into next valid block, thus shifting all next data by offset of one block. If a block specified by value of **Partition end** is reached and not all specified blocks are programmed, operation is halted with error.

On device read, reciprocally, a number of blocks specified by value of **Used partition size** will be read from device starting from a block specified value of **Partition start**. Read data will be stored into buffer counting from a block specified by value of **Partition start**, too. If source block is invalid, it will be skipped (not processed) and programmer will continue with next valid block. Data are stored in buffer continually, without gaps from invalid blocks, so the same image will be created in buffer not regarding invalid blocks distribution over the partition. If a block specified by value of **Partition end** is reached and not all specified blocks are read, operation will close with warning.

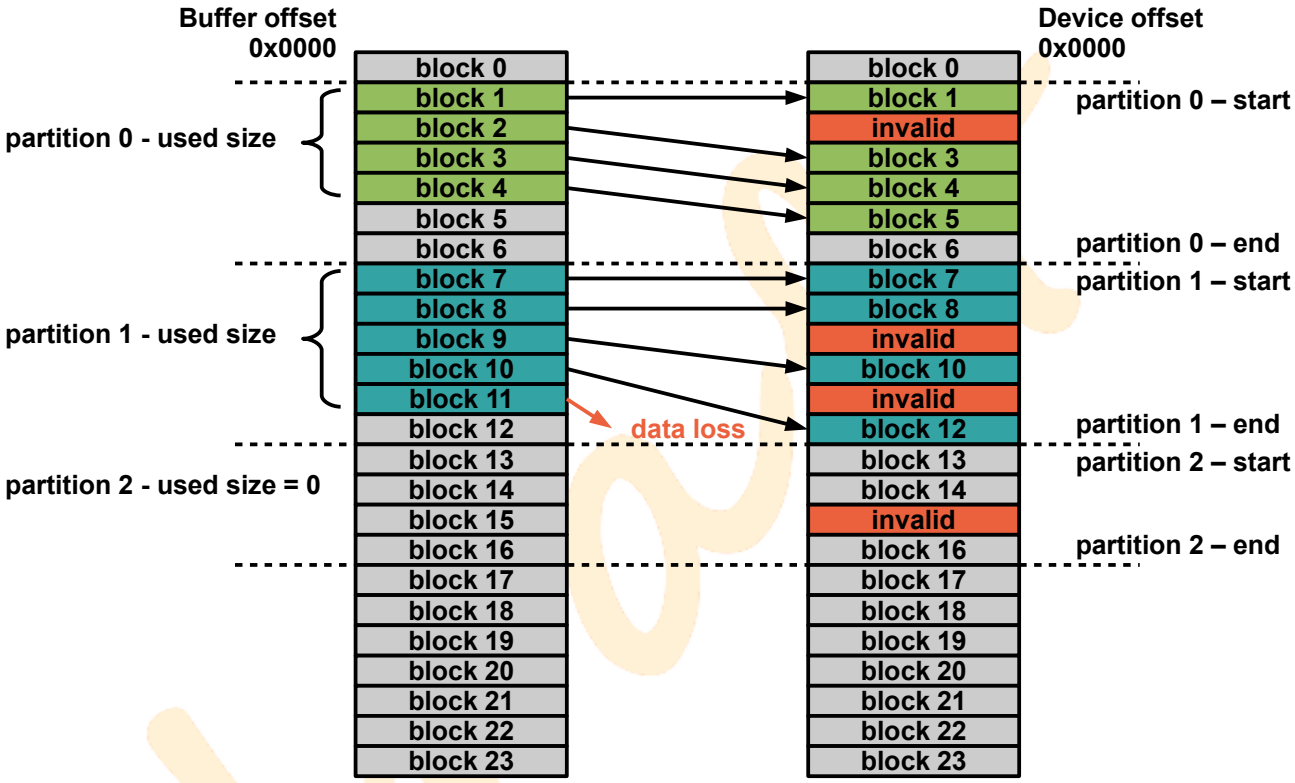


Figure 10: ***Multiple partitions with Skip IB*** *technique graphic representation.*

Figure 10 shows an example device with three partitions.

Partition 0 was programmed successfully. Two unused blocks left at partition end (also referred as padding blocks) are enough for compensation of one invalid block found in device.

Partition 1 couldn't be programmed successfully. There is only one unused block left for invalid blocks compensation, but two invalid blocks were found in device. In consequence, one data block was lost.

Partition 2 is a special kind of unused partition. In fact, it may be used later, by end-appliance itself, but it is not programmed on pre-assembly programming. It may be just specified but not used, simply for your better orientation in more complicated partitioning scheme. Or, there may be other options specified for this partition, providing some level of device quality check (e. g. devices with too many invalid blocks in this partition may be rejected this way from further processing).

PARTITION DEFINITION FILE

Partition definition file is used for instructing the programmer about how to allocate blocks for partitions, and, eventually, what further pre- or post-processing apply on partition. There are several different formats supported. Each partition definition file format will be described in further chapters.

QUALCOMM MULTIPLY PARTITION FORMAT (.MBN)*

Legal note:

Qualcomm Multiply Partition format was invented by Qualcomm Incorporated (USA), not by Elnec. The owner of all potential legal rights is Qualcomm Incorporated. Please, contact Qualcomm CDMA Technologies (http://www.qctconnect.com/) for technical specification.

Generally, our programmers support two versions of Qualcomm Multiply Partition format. They can be simply distinguished by the number of input files.

Procedure for two input files

If you have two input files available, they are generally named FactoryImage.bin and PartitionTable.mbn.

PartitionTable.mbn is rather small (256 bytes) and contains partition table definition. Load this file using menu **File >> Load Partition table**, see chapter **Loading partition table definition file**.

FactoryImage.bin may be rather huge and contains binary data image. Load this file using standard Load procedure, see chapter **Loading data into pg4uw control software buffer**.

It is possible to save data using this format. To save buffer content in binary format, use standard Save procedure (menu **File >> Save**, shortcut **<F2>** or **Save** command from Main toolbar). To save partition table in Qualcomm Multiply Partition compatible format, use menu **File >> Save Partition table**.

Procedure for single input file

If you have single input file available, it is generally named FactoryImage2.mbn. The file is rather huge and contains both, partition table definition and binary data image, plus a header. The file can be simply identified using hex-viewer – you must identify text “Image with header” at file start.

The header specifies also block validity indication byte position. This parameter is also accepted and used for proper reading and/or verifying the device. The value overwrites manual settings in **Invalid blocks indication options** section of **Access Method** window.

Load this file using standard Load procedure, see chapter **Loading data into pg4uw control software buffer**.

It is not possible to save data using this format.

COMMA SEPARATED VALUES (.CSV)*

Partition table definition file uses well-known comma separated values file format.

The file should contain a 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 (;) may be used. Space characters (ASCII code 0x20) are ignored and shouldn't 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 obviously 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 usage, 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 = feature disabled (default)

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

bits 15:12 – Invalid blocks management technique:

0x0 = Treat all blocks

0x1 or 0xF = Skip IB (default)

0x2 = Skip IB with excess abandon

0x3 = Check IB without access

Note: It is possible to specify an equivalent of Check IB with Skip IB technique using Skip IB (0x1 or 0xF) technique and non 0xFFF value for Max. allowed number of invalid blocks in partition.

bits 22:16 – Reserved for future use, consider 0x7F value for future compatibility.

bit 23 – First block in partition must be good:

0 = if first block in respective partition is invalid, device is considered bad and operation is aborted

1 = feature disabled (default)

bits 31:24 – File system preparation:

0xFF = feature disabled (default)

0x00 = JFFS2 Clean Markers are written to unused blocks at respective partition end using MSB byte ordering (big endian)

0x01 = JFFS2 Clean Markers are written to unused blocks at respective partition end using LSB byte ordering (little endian)

Using values other than specified here may 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** option must be also specified.

Partition table definition file example:

```
  0;  100; 20; 0xff7fffff; boot
101;  200; 50; 0xff7fffff; exec
201;  300;  0; 0xff7f3010; res1
301;  400; 50; 0xff7fffff; fsys
401;  500;  0; 0xff7f3010; res2
501; 1000; 50; 0xffffffff; data
```

For loading the table, use 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 partition table data, use menu **File >> Save Partition table**. The table is saved using all values in row, a partition number is used for comment.

GROUP DEFINE FORMAT (.DEF)*

Important note:

The support of this format was implemented based only on fragment of specification available from customer. Therefore it cannot be considered full and reliable. We don't recommend to us it, unless you exactly know what you are doing. If you observe any problems, please, contact our technical support with full Group Define file format specification.

Partition table definition file consists of file header and group records. Each group record specifies one partition.

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 file using this format. To save partition table data, use menu **File >> Save Partition table**.

LOADING PARTITION TABLE DEFINITION FILE

Use menu **File >> Load Partition table** to open **Load Partition table** window. Filter your folder content using partition definition file type mask. Select your file and press button **Open**.

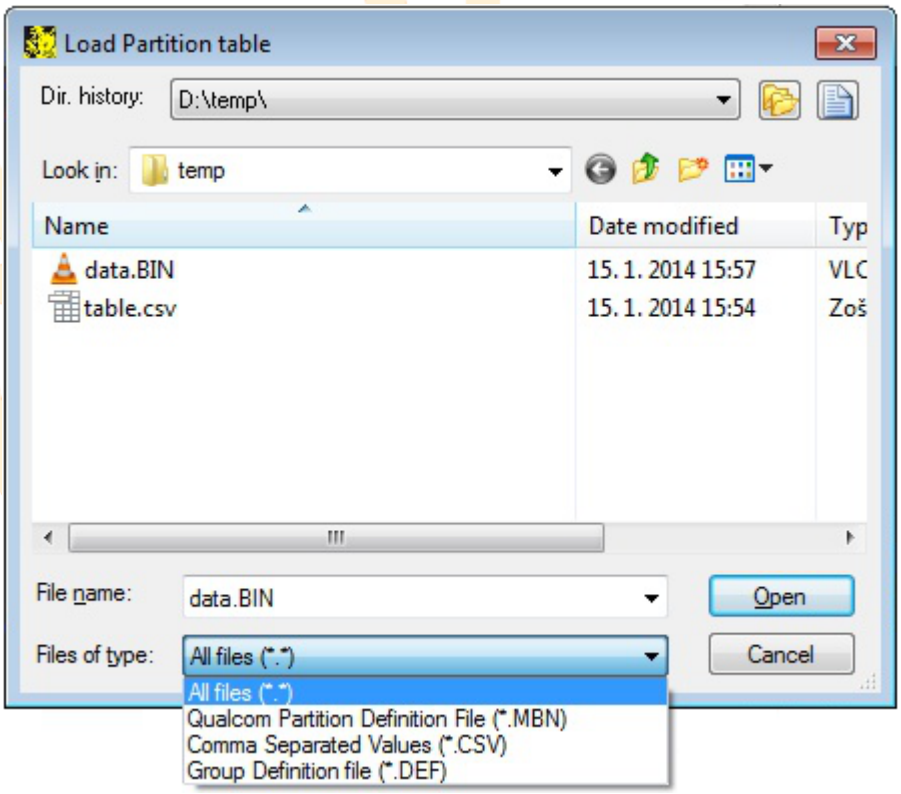


Figure 11: Load Partition table dialog window.

Your partition definition file is then opened, decoded, checked, listed in log window and stored in special buffer (use menu **Buffer >> View/Edit Buffer** to display buffer window, then click on **Partition Table** tab).

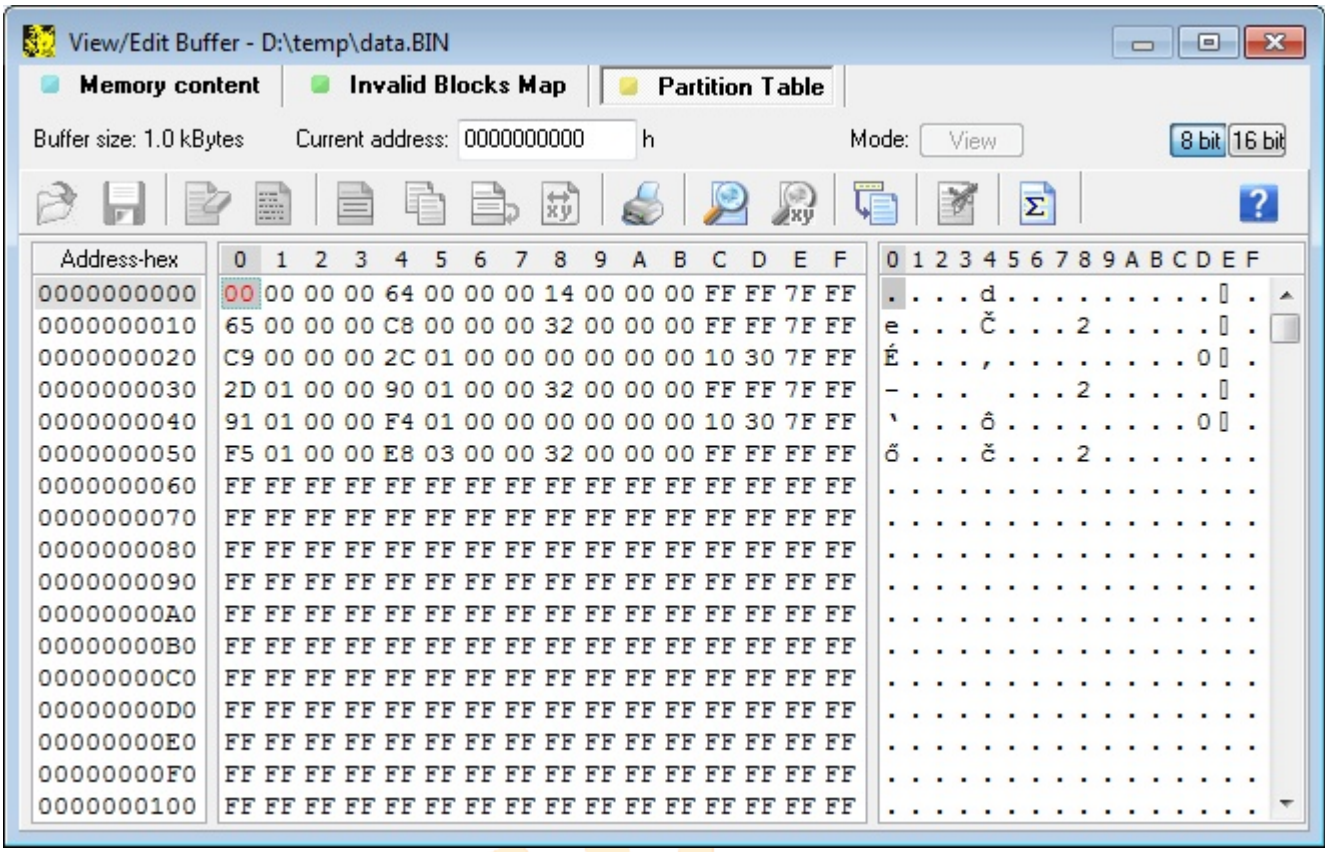


Figure 12: Example of partition table stored in buffer.

Figure 13: Successful partition table definition file load listing example in log window (an example from CSV format description was used).

ACCESS METHOD WINDOW OPTIONS VALIDITY IN PARTITIONING MODE

Only some of options available in **Access Method window** are valid if invalid block management technique based on partitioning is applied. These are:

- Required valid blocks area
- Max. allowed number of invalid blocks in device
- Invalid block indication options
- Tolerant verification options

Please, see respective chapters for detailed informations.

Some devices may involve special hardware features, such as internal ECC controller, some kind of non-volatile blocks locking, etc. If target nand flash device is equipped with such feature, it's enabling and/or other settings are also available in Access Method window. These special hardware features may be used also with partitioning techniques.

SAFE WORKING PROCEDURE

1. Select **Multiple partitions with Skip IB** in **Access Method window**. It is very important to start with this selection, since it triggers programmer and control software internal pre-settings. Only after then it is safe to continue with next steps.
2. Prepare and load **Partition definition file**. In general, it does not matter what is loaded first – partition definition file or input data image(s). But some customized implementations may pre-process input images with respect to specifications in partition definition file, so it is safer to familiarize with operation sequence as is listed here.
3. If necessary, set other options in **Access Method** window, of those accepted in partitioning mode, see chapter **Access Method window options validity in partitioning mode**.
4. Load input data into buffer, see chapter **Loading data into pg4uw control software buffer**.
5. Save your settings into project file and test the operation.

LINUX MTD COMPATIBLE

Note: This invalid blocks management technique is available for selected programmers and/or devices only.

This technique further extends **Multiple partitions with Skip IB** technique with a special feature used by MTD driver in Linux-based operating systems – Bad Blocks Table. All features and procedures mentioned in previous chapters dedicated to **Multiple partitions with Skip IB** technique are valid without any change. The difference is a new options group available in **Access Method window** and accepted only if this technique is in use – **Linux MTD compatible options**. Study, please, respective chapters to get complete information on how to use **Linux MTD compatible** technique.

Limitations:

Only Hamming ECC algorithm is supported by our programmers. The algorithm can recover 1 bit error in 256 byte frame. If manufacturer prescribes more powerful error protection for target nand flash device, Linux MTD compatible technique is not allowed for that device.

If you need to use another ECC algorithm, contact, please, our technical support with your demand.

SPARE AREA USAGE

Our programmers support several modes of spare area usage. Not all modes described here are supported on all programmers. Any other spare area usage mode can be supported upon user's request.

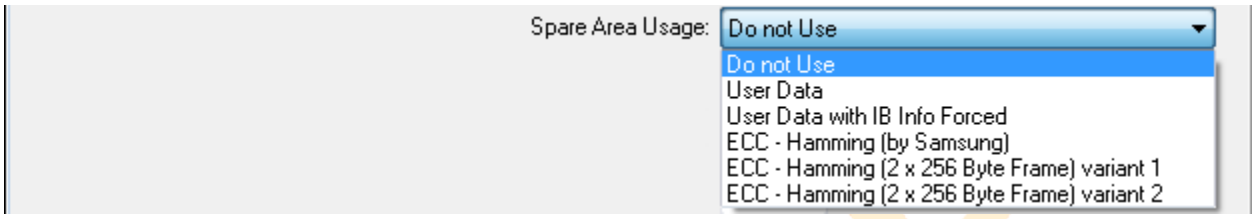


Figure 14: Spare area usage options.

DO NOT USE

Do not use mode is about what its name means – spare area is not used. Data for spare area are neither expected in buffer (see Figure 2 in chapter Data organization in pg4uw control software buffer) nor programmed in or read from target device, respectively.

USER DATA

User data mode treats spare area as is, without any change. Spare area data are both, expected in buffer (see Figure 3 in chapter Data organization in pg4uw control software buffer) and programmed in or read from device, respectively.

This is default spare area usage mode for partitioning techniques (Multiple partitions with Skip IB and Linux MTD compatible).

Important note:

Using this mode may lead to block validity information loss if BI byte is rewritten with any data different from 0xFF (or 0xFFFF for x16 devices). Use with care!

USER DATA WITH IB INFO FORCED

This is an extension of **User data** mode. Data from buffer are modified during programming with aim to keep block validity information – the value at BI byte position is forced to 0xFF (or 0xFFFF for x16 devices).

User can change BI byte position from default using **Invalid blocks indication options (extended)** and related settings.

ECC – HAMMING (BY SAMSUNG)

This spare area usage mode is based on Hamming ECC algorithm, as was proposed by Samsung some time ago. You can access original document also from our archive by clicking this link.

Using **ECC Hamming (by Samsung)** mode, spare area data are not expected in buffer. Programmer will add spare data instead.

A page data area is segmented into 512 byte frames. Spare area is segmented into the same number of frames. E. g. for typical 2048+64 byte page, data area will be segmented into 4 frames of 512 bytes, and spare area into corresponding 4 frames of 16 bytes each, see example on Figure 15.

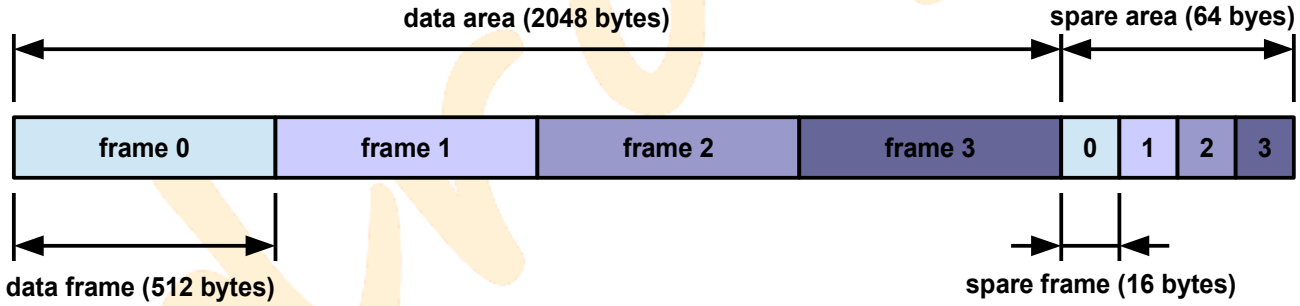


Figure 15: ECC Hamming (by Samsung) page segmentation example.

For each frame in data area, ECC checksum is calculated using Hamming algorithm. This algorithm is capable to detect up to 2 bit errors in a frame, and recover up to 1 bit error in a frame. The calculation produces 3 bytes of checksum. Calculated checksum is inserted into spare area, see Figure 16 and Figure 17 for layouts.

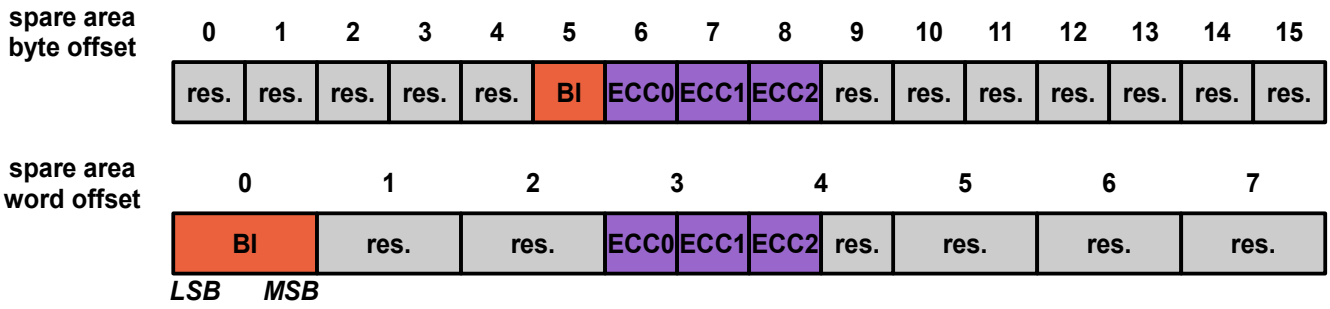


Figure 16: ECC Hamming (by Samsung) spare area layout for small page (512+16 bytes).

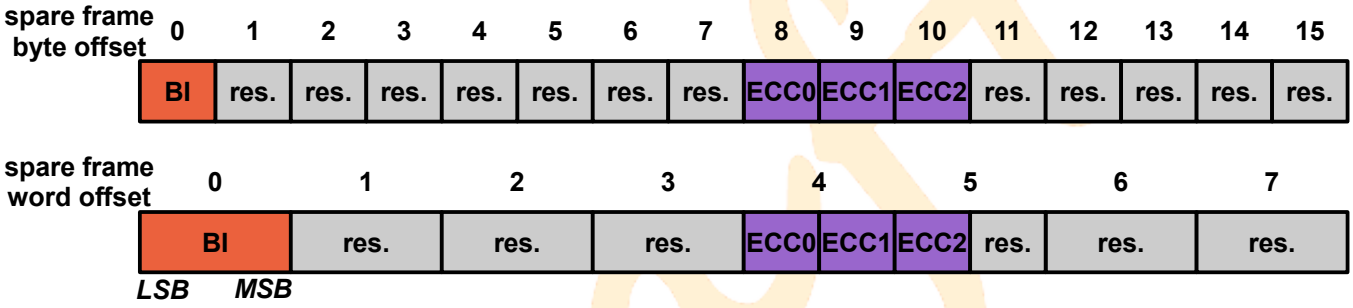


Figure 17: ECC Hamming (by Samsung) spare area layout for large page (2048+64 bytes).

On programming, ECC checksum is calculated and inserted into page buffer. Reserved bytes/words are kept blank. Checksums are programmed into device.

On verifying, ECC checksum is calculated from data in buffer and inserted into compare page buffer. Device page is read as is and its spare area content is compared against calculated content in compare buffer.

On read, ECC checksum is calculated from read data and compared against checksum read from device. Detected errors are repaired before storage in buffer, if possible.

ECC – HAMMING (2 X 256 BYTE FRAME) VARIANT 1 AND 2

Note: This spare area usage mode is available for selected programmers and/or devices only.

This spare area usage mode is based on Hamming ECC algorithm, as is used in Linux MTD subsystem. It is the same spare area usage mode, as can be specified for Linux MTD compatible technique using Apply MTD specific ECC on partition data switch.

Using **ECC Hamming (2 x 256 byte frame)** mode, spare area data are not expected in buffer. Programmer will add spare data instead.

A page data area is segmented into 256 byte frames. Spare area is not segmented (compare to ECC – Hamming (by Samsung) mode).

For each frame in data area, ECC checksum is calculated using Hamming algorithm. This algorithm is capable to detect up to 2 bit errors in a frame, and recover up to 1 bit error in a frame. The calculation produces 3 bytes of checksum. Calculated checksum is inserted into spare area, see Table 1 to Table 3 for layouts.

ECC Hamming (2 x 256 byte frame) variant 1 to **ECC Hamming (2 x 256 byte frame) variant 2** difference is as follows:

In both cases, three bytes of checksum are calculated per data frame – ECC[0], ECC[1], ECC[2]. **Variant 1** stores them in order ECC[0], ECC[1], ECC[2]. This corresponds to default layout used in Linux MTD driver. **Variant 2** stores them in order ECC[1], ECC[0], ECC[2]. This corresponds to SmartMedia layout as can be specified for Linux MTD compatible technique by **Use Smart Media bytes order for ECC** switch (or by CONFIG_MTD_NAND_ECC_SMC switch in Linux MTD driver).

On programming, ECC checksum is calculated and inserted into page buffer. Reserved bytes/words are kept blank. Checksums are programmed into device.

On verifying, ECC checksum is calculated from data in buffer and inserted into compare page buffer. Device page is read as is and its spare area content is compared against calculated content in compare buffer.

On read, ECC checksum is calculated from read data and compared against checksum read from device. Detected errors are repaired before storage in buffer, if possible.

Note: For Linux MTD compatible technique, there are always some data expected for spare area in buffer. These may be user payload data or blank data only. Areas specified as “reserved” in following tables are not affected by **ECC Hamming (2 x 256 byte frame)** spare area usage mode. Existing data in those areas are preserved.

Offset	*Usage*
0	Frame 0 – ECC[0]
1	Frame 0 – ECC[1]
2	Frame 0 – ECC[2]
3	Frame 1 – ECC[0]
4	Reserved
5	Reserved
6	Frame 1 – ECC[1]
7	Frame 1 – ECC[2]
8 ~15	Reserved

Table 1 : ECC - Hamming (2 x 256 byte frame) variant 1 spare area layout for 512 + 16 byte page (variant 2 differs in ECC[0] - ECC[1] ordering).

Offset	*Usage*
0 ~ 39	Reserved
40	Frame 0 – ECC[0]
41	Frame 0 – ECC[1]
42	Frame 0 – ECC[2]
43	Frame 1 – ECC[0]
44	Frame 1 – ECC[1]
45	Frame 1 – ECC[2]
46	Frame 2 – ECC[0]
47	Frame 2 – ECC[1]
48	Frame 2 – ECC[2]
49	Frame 3 – ECC[0]
50	Frame 3 – ECC[1]
51	Frame 3 – ECC[2]
52	Frame 4 – ECC[0]
53	Frame 4 – ECC[1]
54	Frame 4 – ECC[2]
55	Frame 5 – ECC[0]
56	Frame 5 – ECC[1]
57	Frame 5 – ECC[2]
58	Frame 6 – ECC[0]
59	Frame 6 – ECC[1]
60	Frame 6 – ECC[2]
61	Frame 7 – ECC[0]
62	Frame 7 – ECC[1]
63	Frame 7 – ECC[2]

Table 2: ECC - Hamming (2 x 256 byte frame) variant 1 spare area layout for 2048 + 64 byte page (variant 2 differs in ECC[0] - ECC[1] ordering).

Offset	*Usage*
0 ~ 79	Reserved
80	Frame 0 – ECC[0]
81	Frame 0 – ECC[1]
82	Frame 0 – ECC[2]
83	Frame 1 – ECC[0]
84	Frame 1 – ECC[1]
85	Frame 1 – ECC[2]
86	Frame 2 – ECC[0]
87	Frame 2 – ECC[1]
88	Frame 2 – ECC[2]
89	Frame 3 – ECC[0]
90	Frame 3 – ECC[1]
91	Frame 3 – ECC[2]
92	Frame 4 – ECC[0]
93	Frame 4 – ECC[1]
94	Frame 4 – ECC[2]
95	Frame 5 – ECC[0]
96	Frame 5 – ECC[1]
97	Frame 5 – ECC[2]
98	Frame 6 – ECC[0]
99	Frame 6 – ECC[1]
100	Frame 6 – ECC[2]
101	Frame 7 – ECC[0]
102	Frame 7 – ECC[1]
103	Frame 7 – ECC[2]
104	Frame 8 – ECC[0]
105	Frame 8 – ECC[1]
106	Frame 8 – ECC[2]
107	Frame 9 – ECC[0]
108	Frame 9 – ECC[1]
109	Frame 9 – ECC[2]
110	Frame 10 – ECC[0]
111	Frame 10 – ECC[1]
112	Frame 10 – ECC[2]
113	Frame 11 – ECC[0]
114	Frame 11 – ECC[1]
115	Frame 11 – ECC[2]
116	Frame 12 – ECC[0]
117	Frame 12 – ECC[1]
118	Frame 12 – ECC[2]
119	Frame 13 – ECC[0]
120	Frame 13 – ECC[1]
121	Frame 13 – ECC[2]
122	Frame 14 – ECC[0]
123	Frame 14 – ECC[1]
124	Frame 14 – ECC[2]
125	Frame 15 – ECC[0]
126	Frame 15 – ECC[1]
127	Frame 15 – ECC[2]

Table 3 : ECC - Hamming (2 x 256 byte frame) variant 1 spare area layout for 4096 + 128 byte page (variant 2 differs in ECC[0] - ECC[1] ordering).

DEVICE INTERNAL ECC CONTROLLER

Some modern nand flash devices incorporate built-in internal ECC controller. It is a special hardware logic circuit capable to compute ECC checksums for programmed pages, as well as to detect and repair errors for read pages. The option is displayed only for devices equipped with internal ECC controller.

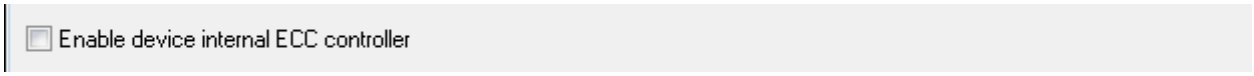


Figure 18: Target device internal ECC controller options.

Important note:

If internal ECC controller usage is enabled on target device, it may come to conflict with spare area data in buffer. In such case, buffer data will be ignored (lost). Always check ECC checksums layout used by internal ECC controller and avoid conflicts while working with target nand flash device equipped with built-in ECC controller.

ENABLE DEVICE INTERNAL ECC CONTROLLER

Confirm the check-box to enable target device internal ECC controller.

Default setting: Disabled.

USER AREA

There are several options available for specification of processed device area. In below form, they can be used for non-partitioning invalid blocks techniques. See chapter Multiple partitions with Skip IB for their partitioning equivalents.

Figure 19: User area options.

USER AREA – START BLOCK

Option **User area – start block** specifies the ordinal number of physical block in target device where user data area should start. If the block specified here is invalid, real user data area start may be shifted or redirected in practice, depending on invalid blocks management technique in use.

Blank check and erase operation don't take this option into account – they always process all or all valid blocks in target device, depending on invalid blocks management technique in use.

Default setting: Block 0 (device start) or block 1 (only for Skip IB with map in 0th block technique).

Note: Decimal (e.g. 123) and hexadecimal (e.g. 0x7B) forms are both accepted for this option.

USER AREA – NUMBER OF BLOCKS

Option **User area – number of blocks** specifies the count of valid physical blocks in target device that should be accessed. If the count specified here cannot be accomplished due to excessive invalid blocks occurrence in device, operation may be aborted with error, depending on invalid blocks management technique in use.

Blank check and erase operation don't take this option into account – they always process all or all valid blocks in target device, depending on invalid blocks management technique in use.

Default setting: 98% of all blocks in target device (typically, manufacturers guarantee less than 2% of invalid blocks, mainly for SLC devices), or minimum valid blocks in device count specified in target device datasheet.

Note: Decimal (e.g. 123) and hexadecimal (e.g. 0x7B) forms are both accepted for this option.

USER AREA – LAST BLOCK

Option **User area – last block** specifies the ordinal number of physical block in target device that operation cannot get beyond. Device area beyond this block should not be accessed. If operation reaches the block specified here and expected count of valid blocks was not processed yet, operation may be aborted with error, depending on invalid blocks management technique in use.

Blank check and erase operation don't take this option into account – they always process all or all valid blocks in target device, depending on invalid blocks management technique in use.

Default setting: last physical block in device.

Note: Decimal (e.g. 123) and hexadecimal (e.g. 0x7B) forms are both accepted for this option.

USER AREA – MAX. ALLOWED NUMBER OF INVALID BLOCKS

Note: This option is available for selected programmers and/or devices only.

Option **User area – max. allowed number of invalid blocks** specifies the maximum count of invalid blocks that are allowed to occur between **User area – start block** and **User area – last block**. If the count specified here is exceeded, operation will be aborted with error.

Blank check and erase operation don't take this option into account – they always process all or all valid blocks in target device, depending on invalid blocks management technique in use.

Default setting: A difference from default User area – number of blocks to all blocks in target device.

Note: Decimal (e.g. 123) and hexadecimal (e.g. 0x7B) forms are both accepted for this option.

REQUIRED VALID BLOCKS AREA

Required valid blocks area options may be used to specify a special area inside of user data area where no one invalid block is allowed. A typical usage of reserved valid blocks area is to preserve uninterrupted bootloader programming into target device first blocks.

Before an operation on target device starts, specified area is checked for invalid blocks presence. If there is any invalid block found there, operation is aborted with error.

These options are accepted by all invalid blocks management techniques except for Treat all blocks.

Blank check and erase operation don't take required valid block area into account – they always process all or all valid blocks in target device, depending on invalid blocks management technique in use.

Figure 20: Required valid blocks area options.

CHECK REQUIRED VALID BLOCKS AREA

Confirm the check-box to enable required valid blocks area feature. If the check-box is not confirmed, the settings of other related options are irrelevant.

Default setting: Disabled.

REQUIRED VALID BLOCKS AREA – START BLOCK

Option **Required valid blocks area – start block** specifies the ordinal number of physical block in target device where required valid blocks area should start. Blocks before the one specified here will be not considered on check.

Default setting: block 0

Note: Decimal (e.g. 123) and hexadecimal (e.g. 0x7B) forms are both accepted for this option.

REQUIRED VALID BLOCKS AREA – NUMBER OF BLOCKS

Option **Required valid blocks area – number of blocks** specifies the count of physical blocks in target device that must be valid, counting from **Required valid blocks area – start block**.

Default setting: 1 block

Note: Decimal (e.g. 123) and hexadecimal (e.g. 0x7B) forms are both accepted for this option.

MAX. ALLOWED NUMBER OF INVALID BLOCKS IN DEVICE

Note: This feature is available for selected programmers and/or devices only.

The feature is useful, if you need to reject devices with too many invalid blocks from usage. If enabled, the count of invalid blocks found in whole target device is evaluated and if preset threshold is exceeded, the device is rejected from further usage. After that, other target device quality features are applied. For example, the programmer may be instructed to reject device if (ordered by evaluation sequence):

- there are more than 20 invalid blocks in device globally – see Max. allowed number of of invalid blocks in device – evaluated once, before operation start;
- there are more than 5 invalid blocks in used area – see User Area – Max. Allowed Number of Invalid Blocks – evaluated continually, as new invalid blocks may be developed during programming and/or erasing;
- there is any invalid block in first 10 used blocks – see Required valid blocks area – evaluated continually, as new invalid blocks may be developed during programming and/or erasing.

Figure 21: Max. allowed number of blocks in device options.

CHECK MAX. ALLOWED NUMBER OF BLOCKS IN DEVICE

Confirm the check-box to enable the feature. If the check-box is not confirmed, the settings of other related options are irrelevant.

Default setting: Disabled.

MAX. ALLOWED NUMBER OF BLOCKS IN DEVICE

Option **Max. allowed blocks in device** specifies the count of physical blocks in target device that are allowed to be invalid.

Default setting: 2 % of total blocks count in device (or total blocks count in device – (minus) minimum valid blocks count in device, if specified in datasheet this way).

Note: Decimal (e.g. 123) and hexadecimal (e.g. 0x7B) forms are both accepted for this option.

BEHAVIOUR ON NEW INVALID BLOCK

Note: This feature is available for selected programmers and/or devices only.

The feature specifies the programmer behaviour in case if new invalid block is developed during operation.

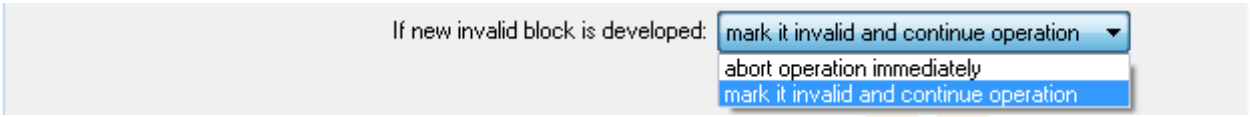


Figure 22: Behaviour on new invalid block options.

IF NEW INVALID BLOCK IS DEVELOPED

Select from drop-down menu:

- **abort operation immediately** – if new invalid block will develop during operation, the programmer will halt immediately with error;
- **mark it invalid and continue operation** – if new invalid block will develop during operation, it will be marked invalid and operation will continue by applying invalid blocks management technique rules.

Default setting: mark it invalid and continue operation

Note: Technically, only program and erase operations are capable to invoke new invalid blocks formation as they apply high voltage across the memory cells array. Read, verify and blank check operations may produce only reversible errors that will disappear after erase. This is a matter of internal nand flash device operation, not of programmer's action. Programmer may just react on events inside of target device.

RESERVED BLOCK AREA OPTIONS

Note: See chapter RBA (Reserved Block Area) for detailed information about related invalid blocks management technique.

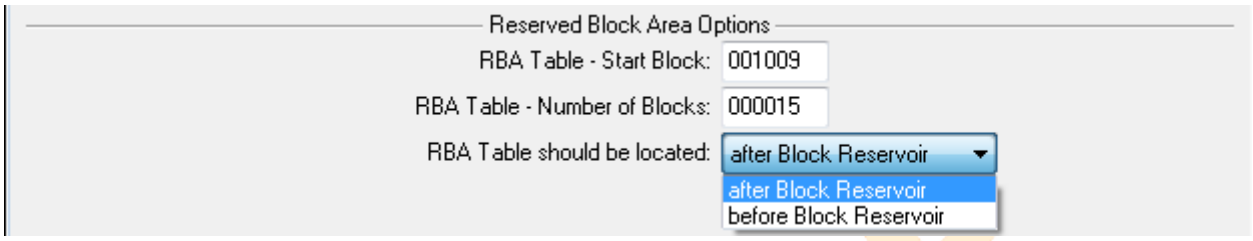


Figure 23: Reserved blocks area options.

RBA TABLE – START BLOCK

Specifies the number of first physical block in device reserved for redirection table programming.

Default setting: last physical block in device – (minus) 15 blocks

Note: Decimal (e.g. 123) and hexadecimal (e.g. 0x7B) forms are both accepted for this option.

RBA TABLE – NUMBER OF BLOCKS

Specifies the count of physical blocks in device reserved for redirection table programming.

Default setting: 15 blocks (we considered this a safe value)

Note: Decimal (e.g. 123) and hexadecimal (e.g. 0x7B) forms are both accepted for this option.

RBA TABLE SHOULD BE LOCATED

Specifies the areas layout in device:

- **after Block Reservoir** – redirection table should be located after the reservoir, i. e. the layout is

as follows: user data area, block reservoir, redirection table area;

- **before Block Reservoir** – redirection table should be placed before the reservoir, i. e. the layout is as follows: user data area, redirection table area, block reservoir.

Default setting: after Block Reservoir

INVALID BLOCKS INDICATION OPTIONS (SIMPLIFIED)

Note: This feature is available on older programmers only.

Invalid blocks indication options allow to customize the way of invalid blocks marking in device.

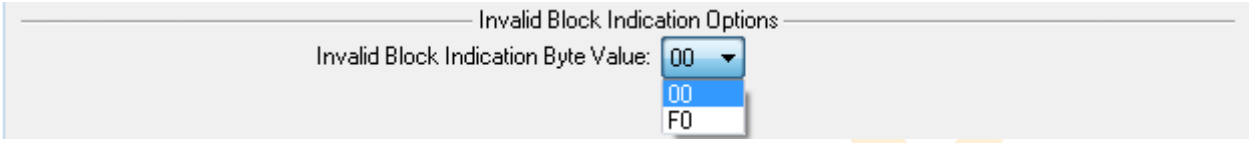


Figure 24: Invalid block indication options (simplified version).

Important note:

Please, keep in mind, that initial invalid blocks often cannot be reprogrammed, so this is only an alternative way. In consequence, there may exist two kinds of invalid blocks in programmed device – one using manufacturer original marking specified in target device datasheet (initial invalid blocks), and other using the marking specified here (acquired invalid blocks).

Also please, keep in mind, that invalid block is invalid because it failed on program or erase operation. It might be not possible to write required mark due to that failure.

INVALID BLOCK INDICATION BYTE VALUE

Specifies the value of BI byte used by programmer for marking invalid blocks developed during program and/or erase operations:

- **0x00** (0x0000 for x16 devices) – default value used by our programmers (datasheets typically specify a non-FF value)
- **0xF0** (0xF0F0 for x16 devices) – the value used in SmartMedia for acquired invalid blocks marking

Default setting: 0x00 (0x0000 for x16 devices)

INVALID BLOCKS INDICATION OPTIONS (EXTENDED)

Note: This feature is available for selected programmers and/or devices only.

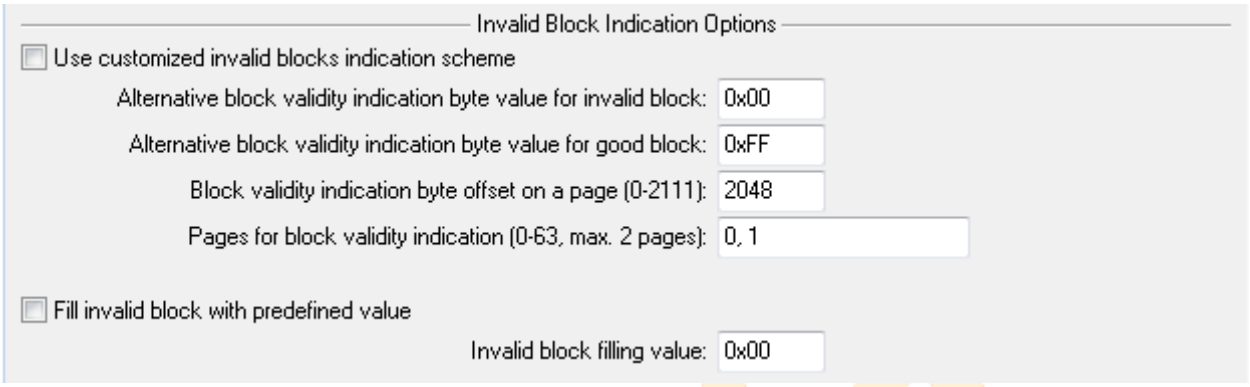


Figure 25: Invalid blocks indication options (extended version).

Invalid blocks indication options allow to customize the way of invalid blocks marking in device. This may be very useful e. g. if an application uses data layout different from device physical page layout. For example, application may work with a page of 512+16+512+16+512+16+512+16 bytes on target device with page of 2048+64 bytes. In such case, device original BI byte will belong to last data frame and some another byte may be used for block validity marking (e. g. byte with page offset 517).

Before the operation start, target device may be scanned for invalid blocks in two ways:

- before program and blank test operation: manufacturer original indication scheme is expected;
- before read, verify and erase: customized indication scheme is expected.

However, the real scheme in device should be indicated using **Target device uses** option in **Device Operation options window.**

On programming:

- initial invalid blocks are left as they are (they might be not rewritable nevertheless);
- acquired initial invalid blocks (if any) are marked using preset scheme;
- valid blocks are marked automatically by user data content or by programmer (if User data with IB info forced is used).

On erasing blank device:

- initial invalid blocks are left as they are (they might be not rewritable nevertheless);
- acquired initial invalid blocks (if any) are marked using preset scheme.

On erasing programmed device:

- programmer will try to rewrite invalid blocks to some generally recognisable format, i. e. it will fill all locations in invalid block to 0x00.

Important note:

Please, keep in mind, that initial invalid blocks often cannot be reprogrammed, so this is only an alternative way. In consequence, there may exist two kinds of invalid blocks in programmed device – one using manufacturer original marking specified in target device datasheet (initial invalid blocks), and other using the marking specified here (acquired invalid blocks).

Also please, keep in mind, that invalid block is invalid because it failed on program or erase operation. It might be not possible to write required mark due to that failure.

USE CUSTOMIZED INVALID BLOCKS INDICATION SCHEME

Confirm the check-box to enable customized invalid blocks indication scheme usage. If enabled, please, bear in mind also target device state specification in **Device Operation options window** <Alt+O> menu, see **Target device uses**.

Default setting: disabled

ALTERNATIVE BLOCK VALIDITY INDICATION BYTE VALUE FOR INVALID BLOCK

Specifies the value of BI byte (BI word for x16 devices) used by programmer for marking invalid blocks developed during program and/or erase operations.

Default setting: 0x00 (0x0000 for x16 devices)

Note: Decimal (e.g. 123) and hexadecimal (e.g. 0x7B) forms are both accepted for this option.

ALTERNATIVE BLOCK VALIDITY INDICATION BYTE VALUE FOR GOOD BLOCK

Specifies the value of BI byte (BI word for x16 devices) used by programmer for marking valid blocks. BI byte will be rewritten during programming:

- automatically by programmer, if User data with IB info forced spare area usage mode is in use;
- by user data, if User data spare area usage mode is in use.

Default setting: 0xFF (0xFFFF for x16 devices)

Note: Decimal (e.g. 123) and hexadecimal (e.g. 0x7B) forms are both accepted for this option.

BLOCK VALIDITY INDICATION BYTE OFFSET ON A PAGE

Specifies BI byte (BI word for x16 devices) offset on a page, in a term of bytes (words for x16 devices), counting from page start = offset 0.

Default setting: first spare area byte (word for x16 devices)

Note: Decimal (e.g. 123) and hexadecimal (e.g. 0x7B) forms are both accepted for this option.

PAGES FOR BLOCK VALIDITY INDICATION

Specifies one or two pages in a block used for BI byte (BI word for x16 devices) recognition.

Default setting: datasheet default (device dependent)

Note: Only decimal numbers are accepted for this setting. Maximum two pages may be specified.

FILL INVALID BLOCK WITH PREDEFINED VALUE

Confirm the check-box to enable filling all positions in invalid blocks by predefined value (see Invalid block filling value below).

Default setting: disabled

Note: If both, **Fill invalid block with predefined value** and Use customized invalid blocks indication scheme are enabled, invalid block will be filled with predefined value on programming, and customized scheme will be used for invalid blocks recognition before operation start (depending on Target device uses setting).

INVALID BLOCK FILLING VALUE

Specifies the value used for filling invalid blocks.

Default setting: 0x00 (0x0000 for x16 devices)

Note: Decimal (e.g. 123) and hexadecimal (e.g. 0x7B) forms are both accepted for this option.

TOLERANT VERIFICATION OPTIONS

Tolerant verification is intended as a substitution of unknown ECC algorithm.

In practice, end-appliance may use any ECC algorithm that meets requirement prescribed by target device manufacturer. Bit flip-flops during read will be detected and recovered by this algorithm.

On the other side, the programmer need not necessarily be aware of used ECC algorithm. User may prepare data image including correct ECC sums in spare area and write it into target device using User data mode. In such a case, it is possible to simulate ECC algorithm behaviour by enabling Tolerant verify feature. The programmer will connive at as many bit errors in a frame as used ECC algorithm is capable to recover.

On programming and read operations, the feature has no effect on programmer behaviour.

On verify and blank check operation, the programmer will tolerate preset count of bit errors in a frame of specified size. If the condition is violated, verify error (or blank check error, respectively) is generated.

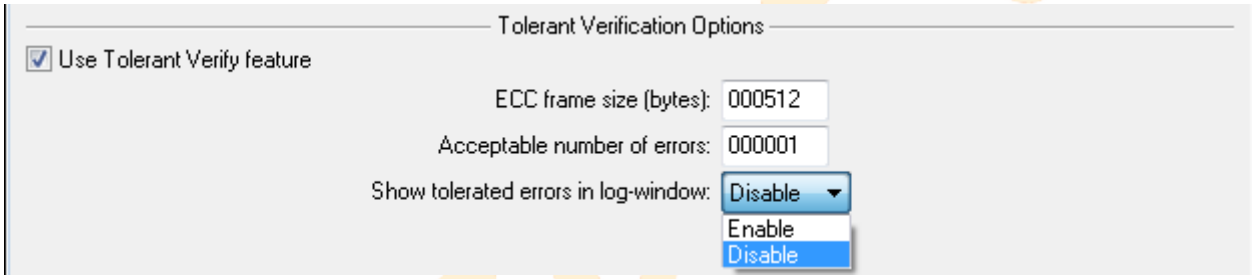


Figure 26: Tolerant verify options.

USE TOLERANT VERIFY FEATURE

Confirm the check-box to enable tolerant verification.

Default setting: enabled

ECC FRAME SIZE (BYTES)

Specifies the frame size in bytes, as is used by ECC algorithm. Frame size should be specified in terms of bytes for both x8 and x16 devices.

Default setting: datasheet default (device dependent)

Note: Decimal (e.g. 123) and hexadecimal (e.g. 0x7B) forms are both accepted for this option.

ACCEPTABLE NUMBER OF ERRORS

Specifies the count of bit-errors in a frame that ECC algorithm is capable to recover.

Default setting: datasheet default (device dependent)

Note: Decimal (e.g. 123) and hexadecimal (e.g. 0x7B) forms are both accepted for this option.

SHOW TOLERATED ERRORS IN LOG-WINDOW

Enables/disables listing of accepted errors in log window.

Default setting: disabled

BLOCK PROTECTION SETTINGS

Note: This feature is available for some devices only.

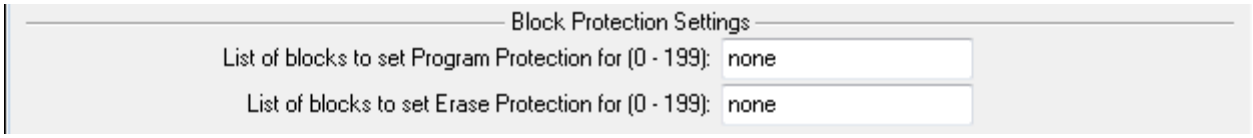


Figure 27: Block protection settings.

Multichip NAND + RAM devices from Samsung based on 512 Mbit C-Die or 1 Gbit (small block) A-Die NAND offer Block protection feature. The feature allows program and/or erase protection on first 200 blocks in device.

LIST OF BLOCKS TO SET PROGRAM PROTECTION FOR

Specifies the blocks that should be protected from further programming. The protection can be disabled by block erase.

Default setting: none

Note: See *missing chapter* for detailed info on how to enter list of blocks.

LIST OF BLOCKS TO SET ERASE PROTECTION FOR

Specifies the blocks that should be protected from further erasing. The protection is permanent.

Default setting: none

Note: See *missing chapter* for detailed info on how to enter list of blocks.

ONE TIME PROTECT AREA

Note: This feature is available for some devices only.

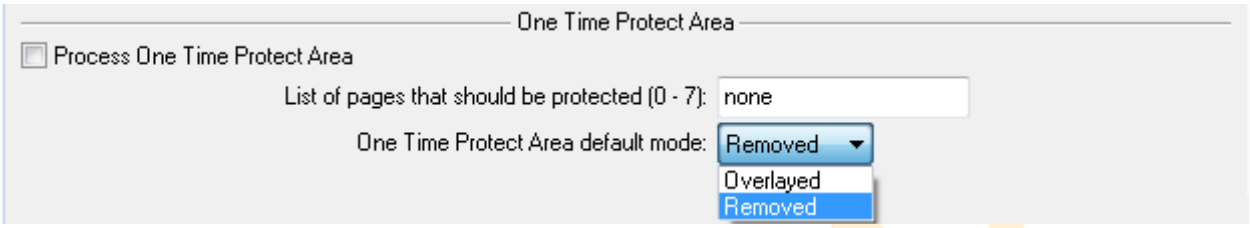


Figure 28: One Time Protect area settings.

S30MS-R device family from Spansion offers a special kind of block – One Time Protect block. The block consists of 8 pages that can be protected from further re-programming. The protection can be set on per-page basis and is permanent. In addition, the block can be set in overlayed mode, thus replacing regular block #0000.

PROCESS ONE TIME PROTECT AREA

Confirm the check-box to enable One Time Protect block processing.

Default setting: disabled

LIST OF PAGES THAT SHOULD BE PROTECTED

Specifies the pages that should be protected from further programming. The protection is permanent.

Default setting: none

Note: See *missing chapter* for detailed info on how to enter list of blocks.

ONE TIME PROTECT AREA DEFAULT MODE

Specifies One Time Protect block mode after device initialization:

- **Overlayed** – One Time Protect block is read in place of the first block in device;
- **Removed** – block #0000 is read in place of the first block in device.

Default setting: Removed.

Note: In default removed mode, One Time Protect block must be temporarily set to overlayed mode and then it can be processed as if it had been block #0000.

LINUX MTD COMPATIBLE OPTIONS

Note: This feature is available for selected programmers only. In addition, target device can not require ECC with more than 1 bit error correction in 256 byte frame (only Hamming's ECC algorithm is supported).

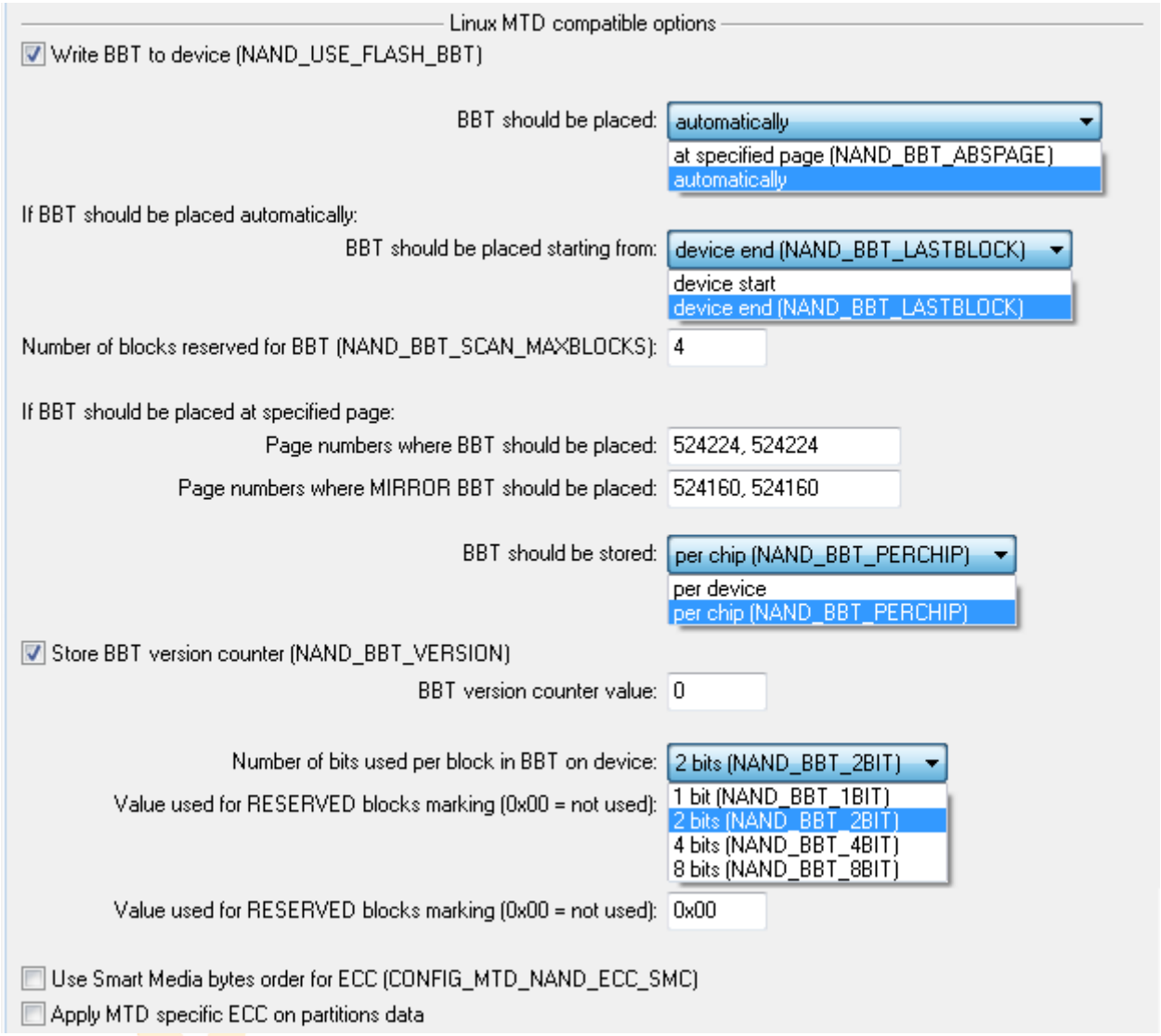


Figure 29: Linux MTD compatible options.

These options allow BBT customization. If there is any symbol name in capitals used in parenthesis (e. g. NAND_USE_FLASH_BBT), it corresponds with the same symbol defined in MTD driver.

WRITE BBT TO DEVICE

Enables/disables BBT write. Confirm the check-box to enable BBT storage in device.

Default setting: enabled

Note: There are always two BBT copies used (BBT and Mirror BBT), as is specified in MTD driver. If it is not possible to write both copies (e. g. due to too many invalid blocks in BBT area), operation is halted with error.

BBT SHOULD BE PLACED

Specifies, whether BBT should be placed by programmer or by user:

- **at specified page** – programmer will write BBT into exactly specified pages;
- **automatically** – programmer will try to place BBT into suitable block within specified area automatically.

Default setting: automatically

BBT SHOULD BE PLACED STARTING FROM

Specifies BBT area location in BBT auto-placement mode:

- **device start** – BBT should be placed in first device blocks;
- **device end** – BBT should be placed in last device blocks.

Default setting: device end

NUMBER OF BLOCKS RESERVED FOR BBT

Specifies the size of BBT area in BBT auto-placement mode in terms of device physical blocks.

Default setting: 4 (MTD driver default value)

Note: Decimal (e.g. 123) and hexadecimal (e.g. 0x7B) forms are both accepted for this option.

Example: For device with 1024 blocks and default settings (BBT at device end, 4 blocks) last 4 blocks in device will be reserved for BBT storage, i. e. blocks #1020, #1021, #1022 and #1023.

PAGE NUMBERS WHERE BBT SHOULD BE PLACED

Specifies the page (or list of pages for multi target devices) where BBT (original version) should be stored in manual-placement mode. Enter page ordinal number here, counting from page 0.

Default setting: first page of last target block

Note: See *missing chapter* for detailed info on how to enter list of blocks.

PAGE NUMBERS WHERE MIRROR BBT SHOULD BE PLACED

Specifies the page (or list of pages for multi target devices) where Mirror BBT (a copy) should be stored in manual-placement mode. Enter page ordinal number here, counting from page 0.

Default setting: first page in last but one target block

Note: See *missing chapter* for detailed info on how to enter list of blocks.

BBT SHOULD BE STORED

Specifies device area covered by single BBT:

- **per device** – one common BBT should be used for all chips in device;
- **per chip** – one BBT should be used for each chip in device.

Default setting: for single chip devices: per device (irrelevant); for multi chip devices: per chip

STORE BBT VERSION COUNTER

Enables/disables version counter storage. Confirm the check-box to enable version numbering.

Default setting: 0

Note: Version counter is incremented by one on each BBT update. In case of power failure during BBT update process, one table copy might be not actualized. On next system boot, BBT or Mirror BBT will be used (if both can be read successfully) based of higher version counter value. This way, the most actual system state will be preserved.

BBT VERSION COUNTER VALUE

Specifies initial BBT version counter value.

Default setting: 0 (MTD driver default value)

Note: Decimal (e.g. 123) and hexadecimal (e.g. 0x7B) forms are both accepted for this option.

NUMBER OF BITS USED PER BLOCK IN BBT ON DEVICE

Specifies the count of bits used to store the status of single block in BBT.

- **1 bit** – 0b = invalid block, 1b = good block
- **2 bits** – 00b = invalid block, 01b = reserved block, 10b = worn-out block, 11b = good block
- **4 bits** – 0000b = invalid block, 0011b = reserved block, 1100b = worn-out block, 1111b = good block
- **8 bits** – 00000000b = invalid block, 00001111b = reserved block, 11110000 = worn-out block, 11111111b = good block

Default setting: 2 bits

VALUE USED FOR RESERVED BLOCKS MARKING

Typically, reserved blocks are for system internal use only and are highlighted in RAM version of BBT, but not in its copy stored in flash device. They appear as normal good blocks in device based BBT. This setting specifies the value used for reserved blocks.

Default setting: 0x00 (reserved block = good block)

USE SMART MEDIA BYTES ORDER FOR ECC

Confirm the check-box to enable Smart Media ECC control sums formatting. See chapter ECC – Hamming (2 x 256 byte frame) variant 1 and 2 for more detailed information.

Default setting: disabled

APPLY MTD SPECIFIC ECC ON PARTITION DATA

Enables/disables on-the-fly ECC application during the action. Though spare area is build automatically if enabled, respective (blank) data are still expected in buffer. See chapter ECC – Hamming (2 x 256 byte frame) variant 1 and 2 for more detailed information.

Default setting: disabled

OTP AREA OPTIONS

Note: The feature is available for selected devices only.

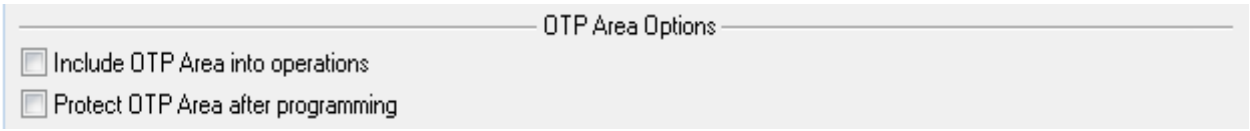


Figure 30: OTP Area options.

INCLUDE OTP AREA INTO OPERATIONS

Enables/disables OTP area processing. Confirm the check-box to enable the feature.

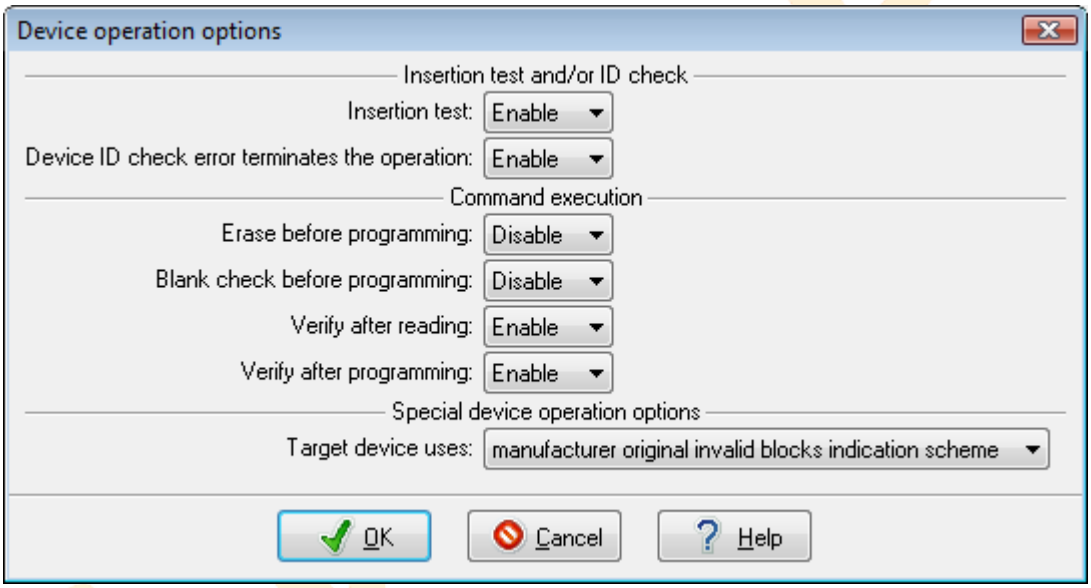
Default setting: disabled

PROTECT OTP AREA AFTER PROGRAMMING

Enables/disables programmed OTP area protection from further rewrite (OTP area cannot be erased, but e. g. 0xFE value still can be rewritten to 0xFC, then to 0xF8 and so on...). Confirm the check-box to enable the protection.

Default setting: disabled

DEVICE OPERATION OPTIONS WINDOW



INSERTION TEST AND/OR ID CHECK

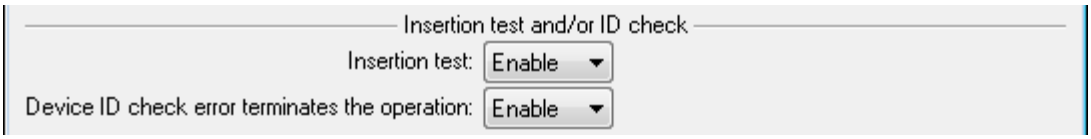


Figure 31: Insertion test and ID check options.

INSERTION TEST

Enables/disables signal continuity check.

Default setting: enable

Note: Please, be aware of static nature of this test. It is capable to detect device misplacing and/or severe integrity fault between programmer's ZIF and device. But it might not detect soft polluted and/or oxidized contact as to which may fully manifest only at high-speed operation. Therefore it is a rule of thumb to check and clean all contacts on adapter(s) and device in case of verify after programming problems. The other important rule is to not overuse programming adapters – respect, please, stated lifetime of adapter's ZIF socket.

DEVICE ID CHECK ERROR TERMINATES THE OPERATION

Enables/disable operation halting on ID check error.

Default setting: enable

Note: ID check error may point to error in device selection and/or insertion. Ignoring it may lead to inserted device and/or programmer damage. Disable this option only if you are sure of what you are doing.

COMMAND EXECUTION

Figure 32: Command execution options.

ERASE BEFORE PROGRAMMING

Enables/disables device erasing before programming.

Default setting: disable

BLANK CHECK BEFORE PROGRAMMING

Enables/disables device erase check before programming.

Default setting: disable

Note: NAND flash devices incorporate internal controller that manages program and erase operations. The operation state is indicated in STATUS register. If block erase command is finished with PASS status, it means, that all memory cells in respective block are in erased state, i. e. blank. Therefore, **Blank check before programming** operation is skipped if it is enabled together with **Erase before programming** operation. See also chapter **Two factors that programmer relies on**.

VERIFY AFTER READING

Enables/disables read data verification.

Default setting: enable

VERIFY AFTER PROGRAMMING

Enables/disables verification of programmed data against data in buffer.

Default setting: enabled

Note: If programmer detects a blank page, it will skip this page during both, **programming** and **verify after programming** operations. But all specified pages are checked during **Verify** operation.

SPECIAL DEVICE OPERATION OPTIONS

Note: This feature is available on selected programmers and/or devices only.

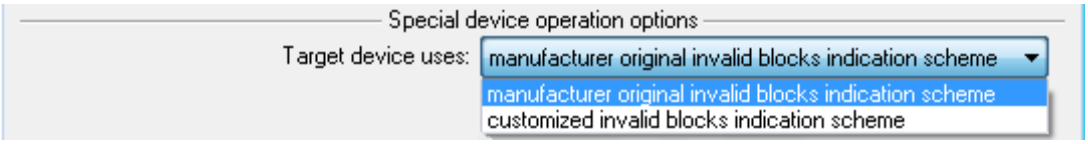


Figure 33: Special device operation options.

TARGET DEVICE USES

Specifies the way of invalid blocks indication used in target device.

- **manufacturer original invalid blocks indication scheme** – data in device respect BI bytes as they are specified in device datasheet
- **customized invalid blocks indication scheme** – data in device use a scheme specified by customer, see also chapter **Invalid blocks indication options (extended)**.

SPECIAL NAND FLASH COMMANDS

Note: Following commands can be accessed through menu **Device**. Commands are available on selected programmers only.

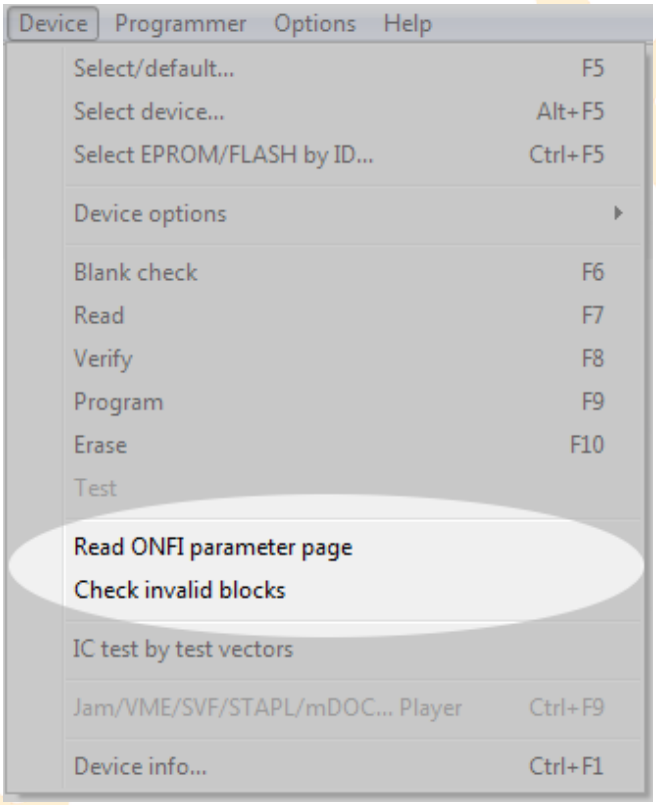


Figure 34: Special nand flash related commands in Device menu.

READ ONFI PARAMETER PAGE

ONFI standards (see http://www.onfi.org for more information about Open Nand Flash Interface working group) involve a special memory page containing detailed data about device identification, memory array arrangement, timing parameters, special features supported, etc. This page can be read using ***Read ONFI parameter page*** command. After parameter page is successfully read, it is decoded and outputted in comprehensible form to a text file stored on your desktop.

ONFI parameter page report example:

```
ELNEC ONFI Decoder
28.11.2013 16:56:14

ID read from device from address 0x00:
 2C F1 80 95 04 00 00 00
ID read from device from address 0x20:
 4F 4E 46 49

Revision information and features block
 Parameter page signature: 'ONFI'
 Revision number: 0x0002
    ONFI version 1.0
 Features supported: 0x0010
    supports odd to even page Copyback
 Optional commands supported: 0x003F
    supports Page Cache Program command
    supports Read Cache commands
    supports Get Features and Set Features
    supports Read Status Enhanced
    supports Copyback
    supports Read Unique ID

Manufacturer information block
 Device manufacturer: 'MICRON      '
 Device model: 'MT29F1G08ABAEAWP    '
 JEDEC manufacturer ID: 0x2C
 Date code: Y:0 W:0

Memory organization block
 Number of data bytes per page: 2048
 Number of spare bytes per page: 64
 Number of data bytes per partial page: 512
 Number of spare bytes per partial page: 16
 Number of pages per block: 64
 Number of blocks per logical unit (LUN): 1024
 Number of logical units (LUNs): 1
 Number of address cycles: 0x22
    Row address cycles: 2
    Column address cycles: 2
 Number of bits per cell: 1
 Bad blocks maximum per LUN: 20
 Block endurance: 1 x 10^5
 Guaranteed valid blocks at beginning of target: 1
 Block endurance for guaranteed valid blocks: 0
 Number of programs per page: 4
 Partial programming attributes: 0x00
    none
 Number of bits ECC correctability: 4
 Number of plane address bits: 0
 Multi-plane operation attributes: 0x00
    none
```

```
Electrical parameters block
 I/O pin capacitance, maximum: 10
 Asynchronous timing mode support: 0x003F
    supports timing mode 0, shall be 1
    supports timing mode 1
    supports timing mode 2
    supports timing mode 3
    supports timing mode 4
    supports timing mode 5
 Asynchronous program cache timing mode support: 0x003F
    supports timing mode 0
    supports timing mode 1
    supports timing mode 2
    supports timing mode 3
    supports timing mode 4
    supports timing mode 5
 tPROG Maximum page program time (us): 600
 tBERS Maximum block erase time (us): 3000
 tR Maximum page read time (us): 25
 tCCS Minimum change column setup time (ns): 100

Vendor block
 Vendor specific Revision number: 0x0001
 Vendor specific (in Hex form):
 01 00 00 02 04 80 01 81
 04 01 02 01 0A 00 00 00
 00 00 00 00 00 00 00 00
 00 00 00 00 00 00 00 00
 00 00 00 00 00 00 00 00
 00 00 00 00 00 00 00 00
 00 00 00 00 00 00 00 00
 00 00 00 00 00 00 00 00
 00 00 00 00 00 00 00 00
 00 00 00 00 00 00 00 00
 00 00 00 00 00 00 00 00

Integrity CRC: 0xAAC2

Parameter page dump (in Hex form):
 4F 4E 46 49 02 00 10 00 3F 00 00 00 00 00 00 00
 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 4D 49 43 52 4F 4E 20 20 20 20 20 20 4D 54 32 39
 46 31 47 30 38 41 42 41 45 41 57 50 20 20 20 20
 2C 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 00 08 00 00 40 00 00 02 00 00 10 00 40 00 00 00
 00 04 00 00 01 22 01 14 00 01 05 01 00 00 04 00
 04 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 0A 3F 00 3F 00 58 02 B8 0B 19 00 64 00 00 00 00
 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 00 00 00 00 01 00 01 00 00 02 04 80 01 81 04 01
 02 01 0A 00 00 00 00 00 00 00 00 00 00 00 00 00
 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 00 00 00 00 00 00 00 00 00 00 00 00 00 00 C2 AA
```

CHECK INVALID BLOCKS

This special nand flash command screens all blocks in device for their validity status. Collected statuses are outputted to a file stored on your desktop, together with basic device quality statistics.

Block numbers are displayed with regard to device as a whole as well as with regard to respective chip. Similarly, overall and chip statistics are displayed.

Check invalid blocks report example:

```
Check invalid blocks - report:

>> 28.11.2013, 17:03:29
Checking invalid blocks: Toshiba TH58NVG5H0ETA20 [TSOP48].

Device contains 2 chips with 8192 blocks each.

Invalid blocks listing:
Total block no. | Chip no. | Block no.
----------------+----------+----------
           5368 |        0 |     5368
           9641 |        1 |     1449
          10133 |        1 |     1941
----------------+----------+----------

Invalid blocks count:
Chip no. 0:        1
Chip no. 1:        2
--------------------
Device    :        3

Invalid blocks percentage:
Chip no. 0:     0,01 % (of chip blocks count)
Chip no. 1:     0,02 % (of chip blocks count)
-------------------------------------------
Device    :     0,02 % (of device blocks count)

Invalid blocks distribution ratio:
Chip no. 0:    33,33 % (of total invalid blocks count)
Chip no. 1:    66,67 % (of total invalid blocks count)
```

GLOSSARY

Acquired invalid block

An invalid block developed during a lifetime.

Address

Consists of row address and column address. The row address identifies the page, block, and LUN to be accessed. The column address identifies the byte or word within a page to access.

BBT (Bad Blocks Table)

A table containing the list of invalid blocks in particular device or chip. Used by certain invalid blocks management methods.

BI byte/word

A column used for invalid blocks marking.

Bit error

A single bit error caused primarily by defective material. May be of temporary nature (disappear after erase) or of permanent nature.

Block

Consists of multiple pages and is the smallest addressable unit for erase operations.

Column

The byte (for x8 devices) or word (for x16 devices) location within the page register.

Data area

A part of a page used for storage of payload data.

ECC (Error-Correcting Code)

A mathematical algorithm used for temporary bit errors detection and correction. Also referred as EDC (Error Detection and Correction).

Extra large page

A page with data area size of 8192 bytes / 4096 words.

Initial invalid block

An invalid block detected already on manufacturing line.

Invalid block

A block that contains one or more damaged memory cells (permanent bit errors). Such block may not be used for data storage. Also referred as bad block or damaged block. An opposite is referred as valid or good block.

Invalid block management

A set of algorithms used for treatment of invalid blocks.

Large block

A block consisting of large pages. Generally, this term is obviously used for blocks containing pages larger than small page.

Large page

A page with data area size of 2048 bytes / 1024 words.

Linux MTD

Linux MTD is a set of drivers used in Linux-based operating systems for treatment of flash-based memory devices. (MTD stands for Memory Technology Device.)

LUN (Logical Unit Number)

The minimum unit that can independently execute commands. There may be one or more LUNs per NAND chip.

MLC (Multi Level Cell)

A nand flash technology that uses memory cells capable to carry more than 1 bit of information. At this time, MLC is typically used for 2 bits-per-cell technologies rather than for its general meaning. 2 bits-per-cell cell type works with four voltage levels (00, 01, 10, 11).

MCP (Multi Chip Package)

A device containing several memory units, typically of various types (e.g. RAM, NOR, NAND, eMMC,...).

NAND chip

A set of LUNs that share one nCE signal within NAND package.

NAND device

Packaged NAND unit. A device consists of one or more NAND chips.

Page

The smallest addressable unit for read and program operations.

Page register

For read operation – this register is used for reading data transferred from flash array. For program operation – data is placed here before transferring to flash array.

Redirection table

A table containing the list of pairs of invalid blocks and valid blocks used for their substitution. Also referred as substitution table.

Reserve block method

A method of invalid blocks management. If invalid block is found, a next valid block is taken from blocks reservoir

and used instead.

Row

Refers to block and page to be accessed.

Skip block method

A method of invalid blocks management. If invalid block is found, it is skipped and next valid block is used instead.

SLC (Single Level Cell)

A nand flash technology that uses memory cells capable to carry 1 bit of information. This cell type works with two voltage levels (0 and 1).

Small block

A block consisting of small pages.

Small page

A page with data area size of 512 bytes / 256 words.

Spare area

A part of a page, typically used for storage of meta-data (ECC, file system information...). Also referred as OOB (Out-Of-Boundary) area.

TLC (Triple level cell)

A nand flash technology that uses memory cells capable to carry 3 bits of information in single cell. This cell type works with eight voltage levels (000, 001, …, 111).

Very large page

A page with data area size of 4096 bytes / 2048 words.

Wear levelling

A set of algorithms used for ensuring balanced blocks usage.

HISTORY

Rev.	Date	Comment
0.1	2013, December 04	Initial draft
0.2	2014, January 30	Images quality enhanced Added link to Elnec application note: NAND flash memories