

Programming NAND Flash Memories Using Elnec Device Programmers

Application note

April 2025

an_programming_nand_flash_using_elnec_programmers, version 1.4

Disclaimer:

This application note describes how to program NAND flash devices using Elnec 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.

This application note may refer to various products and brand names that may be claimed as property of their respective owners.

Contents

1. Brief Comments On NAND Flash Inwards.....	9
2. Brief Comments On Invalid Blocks.....	12
3. Brief Comments On Bit Errors.....	13
4. Two Factors That Programmer Relies On.....	14
5. Data Organization In Pg4uw Control Software Buffer.....	15
6. Loading Data Into Pg4uw Control Software Buffer.....	16
6.1. Loading Multiple Data Images.....	17
7. Access Method Window.....	18
7.1. Invalid Block Management.....	19
7.1.1. Treat All Blocks.....	19
7.1.2. Skip IB.....	20
7.1.3. Skip IB with map in 0-th block.....	21
7.1.4. Skip IB with excess abandon.....	22
7.1.5. RBA (Reserved Block Area).....	22
7.1.6. Check IB without access.....	25
7.1.7. Check IB with Skip IB.....	25
7.1.8. Discard Invalid block(s) data.....	25
7.1.9. Multiple partitions with Skip IB.....	26
7.1.9.1. Partition definition file.....	27
7.1.9.1.1. Qualcomm multiply partition format (*.mbn).....	27
7.1.9.1.1.1. Procedure for two input files.....	28
7.1.9.1.1.2. Procedure for single input file.....	28
7.1.9.1.2. Comma separated values format (*.csv).....	28
7.1.9.1.3. Group define format (*.def).....	30
7.1.9.1.4. Loading Partition definition file.....	30
7.1.9.1.4.1. Error codes on Partition definition file load.....	33
7.1.9.2. Access Method window options validity in partitioning mode.....	33
7.1.9.3. Safe working procedure.....	34
7.1.10. Linux MTD compatible.....	34
7.1.11. Redirection with HW Look Up Table (LUT).....	34
7.2. Spare Area Usage.....	35

7.2.1. Do not use.....	35
7.2.2. User data.....	35
7.2.3. User data with IB info forced.....	35
7.2.4. ECC – Hamming (by Samsung).....	36
7.2.5. ECC – Hamming (2×256 byte frame) variant 1 and 2.....	37
7.3. Device Internal ECC Controller Options.....	40
7.3.1. Enable device internal ECC controller.....	40
7.4. User Area Options.....	40
7.4.1. User Area – Start Block.....	41
7.4.2. User Area – Number of Blocks.....	41
7.4.3. User Area – Last Block.....	41
7.4.4. User Area – Max. Allowed Number of Invalid Blocks.....	41
7.5. Required Valid Blocks Area Options.....	42
7.5.1. Check Required Valid Blocks Area.....	42
7.5.2. Required Valid Blocks Area – Start Block.....	42
7.5.3. Required Valid Blocks Area – Number of Blocks.....	43
7.6. Max. Allowed Number Of Invalid Blocks In Device Options.....	43
7.6.1. Check Max. Allowed Number of Invalid Blocks in Device.....	43
7.6.2. Max. Allowed Number of Invalid Blocks in Device.....	43
7.7. Behaviour On New Invalid Block Options.....	44
7.7.1. If new invalid blocks is developed.....	44
7.8. Tolerant Verification Options.....	44
7.8.1. Use Tolerant Verify feature.....	45
7.8.2. ECC frame size.....	45
7.8.3. Acceptable number of errors.....	45
7.8.4. Tolerant verify examples.....	45
7.9. Invalid Block Indication Options (Simplified Version).....	46
7.9.1. Invalid Block Indication Byte Value.....	46
7.10. Invalid Block Indication Options (Extended Version).....	47
7.10.1. Use customized invalid blocks indication scheme.....	48
7.10.2. Alternative block validity indication byte value for invalid block.....	48
7.10.3. Alternative block validity indication byte value for good block.....	48
7.10.4. Block validity indication byte offset on a page.....	49
7.10.5. Pages for block validity indication.....	49
7.10.6. Fill invalid block with predefined value.....	49
7.10.7. Invalid block filling value.....	49
7.11. Reserved Block Area Options.....	50
7.11.1. RBA Table – Start Block.....	50
7.11.2. RBA Table – Number of Blocks.....	50
7.11.3. RBA Table should be located.....	50
7.12. Linux MTD Compatible Options.....	51

7.12.1. Write BBT to device.....	51
7.12.2. BBT should be placed.....	52
7.12.3. BBT should be placed starting from.....	52
7.12.4. Number of blocks reserved for BBT.....	52
7.12.5. Page numbers where BBT should be placed.....	52
7.12.6. Page numbers where Mirror BBT should be placed.....	53
7.12.7. BBT should be stored.....	53
7.12.8. Store BBT version counter.....	53
7.12.9. BBT version counter Value.....	53
7.12.10. Number of bits used per block in BBT on device.....	53
7.12.11. Value used for reserved blocks marking.....	54
7.12.12. Use Smart Media bytes order for ECC.....	54
7.12.13. Apply MTD specific ECC on partition data.....	54
7.13. Special Device Features.....	54
8. Device Operation Options Window.....	56
8.1. Insertion Test And / Or ID Check.....	57
8.1.1. Insertion test.....	57
8.1.1.1. Basic test of IC functionality.....	57
8.1.2. Device ID check error terminates the operation.....	57
8.2. Command Execution.....	58
8.2.1. Erase before programming.....	58
8.2.2. Blank check before programming.....	58
8.2.3. Verify after reading.....	58
8.2.4. Verify after programming.....	59
8.3. Special Device Operation Options.....	59
8.3.1. Target device uses.....	59
9. Special NAND Flash Commands.....	60
9.1. Read ONFI Parameter Page.....	60
9.2. Read JEDEC Parameter Page.....	63
9.3. Check Invalid Blocks.....	65
10. Using Multi-Project Feature For NAND Flash.....	66
10.1. Working With Multi-Project Wizard.....	66
10.1.1. Defining the Project files for individual NAND partitions.....	68
10.1.2. Building the Multi-Project file.....	69
10.1.3. Running the multi-chip device operation.....	69
11. Customized NAND Flash Support.....	70
11.1. Partitioning Scheme.....	70
11.2. Invalid Blocks Management Scheme.....	70
11.3. Dynamic Meta-data Scheme.....	71
11.4. Page Arrangement Scheme.....	71

11.5. Error Control And Correction Scheme.....	71
11.6. Unused Blocks Formatting Scheme.....	72
11.7. Input Data File Scheme.....	72
12. Frequently Asked Questions.....	73
12.1. Device / Buffer Conversions.....	73
12.1.1. Conversion of the device offset to the block number.....	73
12.1.2. Conversion of the device offset to the buffer offset.....	73
12.1.3. Conversion of the file size to the blocks count.....	74
12.1.4. Conversion of the block number to buffer offset.....	75
12.2. Copying NAND Flash Memory.....	76
12.3. Problems With Too Many Invalid Blocks.....	77
12.3.1. How does your programmer identify invalid blocks?.....	77
12.3.2. When working with device, programmer reports tens (hundreds, thousands) of invalid blocks. Is it normal?.....	77
12.3.3. I need to do a test with a bad block so I want to make a sample with a bad blocks I make on my own.....	77
12.3.4. I have made a lot of invalid blocks in my device. Can I fix it somehow?.....	78
12.4. Command Execution Dilemmas.....	78
12.4.1. Erase before programming.....	78
12.4.2. Blank check before programming.....	78
12.4.3. Verify after programming.....	78
12.4.4. Pg4uw software recommends me to set more User Area blocks than I have set, saying it is more effective. Is it OK?.....	79
13. Appendix A: Errors In NAND Flash – The Background.....	80
13.1. Memory Wear (Endurance) Errors.....	80
13.2. Read Disturb Errors.....	81
13.3. Program Disturb Errors.....	82
13.4. Over-programming Errors.....	83
13.5. Data Retention Errors.....	84

List Of Figures

Figure 1: NAND flash vs. NOR flash cell comparison.....	9
Figure 2: NAND flash internal structure.....	10
Figure 3: Invalid Block Map building flowchart.....	12
Figure 4: Buffer data layout, if spare area is not used.....	15
Figure 5: Buffer data layout, if spare area is used.....	15
Figure 6: Load file dialog window.....	16
Figure 7: Invalid Block Management options.....	19
Figure 8: Treat All Blocks technique graphic representation.....	20
Figure 9: Skip IB technique graphic representation.....	21
Figure 10: 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.....	23
Figure 11: RBA technique graphic representation.....	23
Figure 12: Multiple partitions with Skip IB technique graphic representation.....	27
Figure 13: Load partition table window.....	31
Figure 14: Example of partition table stored in buffer.....	31
Figure 15: Successful partition definition file load listing example in log window (an example from CSV format description was used).....	32
Figure 16: Spare area usage options.....	35
Figure 17: ECC - Hamming (by Samsung) page segmentation example.....	36
Figure 18: ECC Hamming (by Samsung) spare area layout for small page (512+16 bytes).....	36
Figure 19: ECC Hamming (by Samsung) spare area layout for large page (2 048+64 bytes).....	36
Figure 20: Device internal ECC controller options.....	40
Figure 21: User Area options.....	40
Figure 22: Required valid blocks area options.....	42
Figure 23: Max. allowed number of blocks in device options.....	43
Figure 24: Behaviour on new invalid block options.....	44
Figure 25: Tolerant verification options.....	45
Figure 26: Invalid block indication options (simplified version).....	46
Figure 27: Invalid blocks indication options (extended version).....	48
Figure 28: Reserved blocks area options.....	50
Figure 29: Linux MTD compatible options.....	51
Figure 30: Special device features menu example.....	54
Figure 31: Insertion test and ID check options.....	57

Figure 32: Command execution options.....	58
Figure 33: Special device operation options.....	59
Figure 34: Menu device.....	60
Figure 35: Empty Pg4uw Multi-Project Wizard window.....	66
Figure 36: Multi-Project Wizard window with Multi-Project file loaded.....	69
Figure 37: A NAND flash block architecture.....	80
Figure 38: Wear-out (endurance) errors.....	81
Figure 39: Read disturb errors.....	82
Figure 40: Program disturb errors.....	83
Figure 41: Over-programming errors.....	84
Figure 42: Data retention errors.....	85

1. Brief Comments On NAND Flash Inwards

- NOR flash and NAND flash **cell difference** (see Figure 1):

In **NOR flash**, each cell has one end connected directly to source line (ground), and the other end connected directly to a bit line. This arrangement is called NOR flash because it acts like a NOR gate: when one of the word lines is brought high, the corresponding storage transistor acts to pull the output bit line low.

In **NAND flash** the transistors are connected in a way that resembles a NAND gate: several transistors are connected in series, and the bit line is pulled low only if all the word lines are pulled high. These groups are then connected via some additional transistors to a NOR-style bit line array in the same way that single transistors are linked in NOR flash.

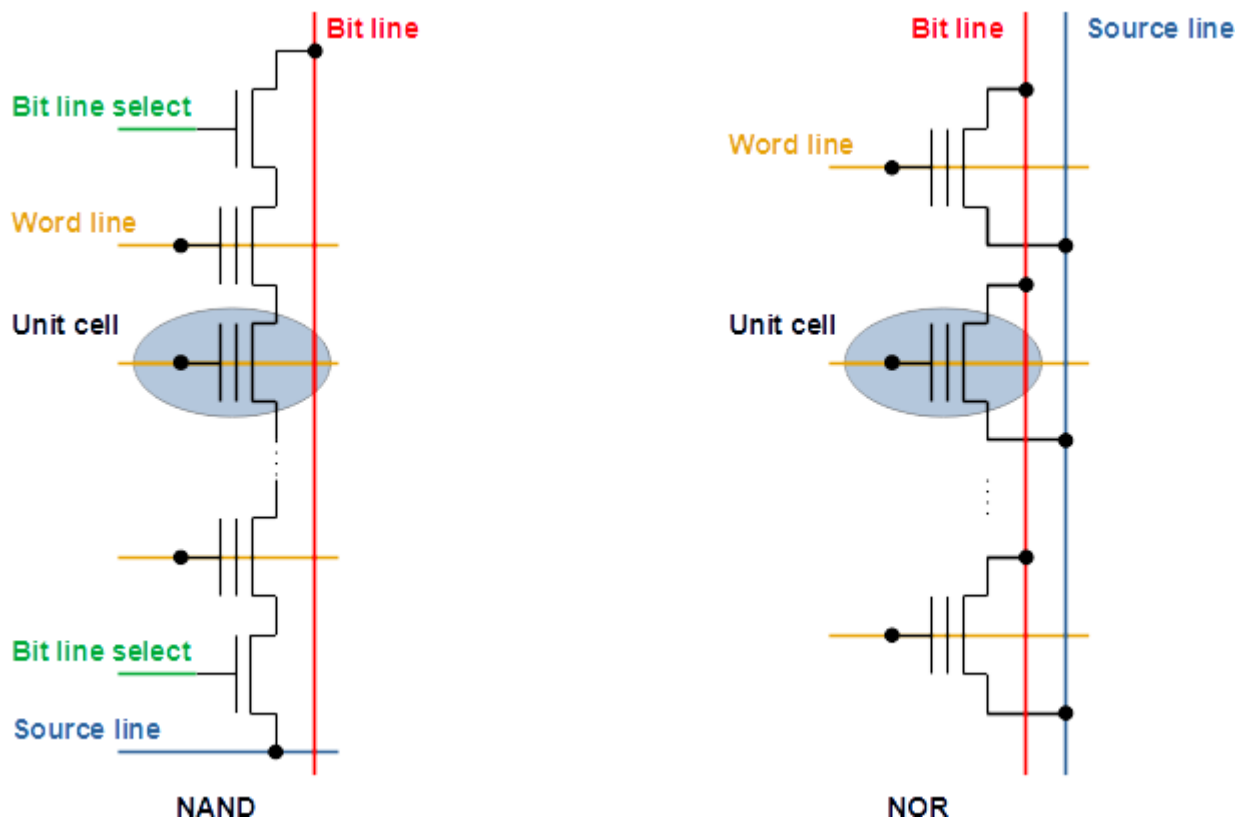


Figure 1: NAND flash vs. NOR flash cell comparison.

- SLC, MLC, TLC, QLC...**: These magic acronyms stand for x-Level Cell technology. Their usage is, however, rather confusing. Single Level Cell (SLC) uses 2 levels of electric charge to store 1 bit of information (0 or 1). Multi Level Cell (MLC) uses 4 levels of electric charge to store 2 bits of information (00, 01, 10, 11). Triple Level Cell (TLC) uses 8 levels of electric charge to store 3 bits of information (000, 001,

010, 011, 100, 101, 110, 111). And finally Quad Level Cell (QLC) uses 16 levels of electric charge to store 4 bits of information. Converting of electric charge level to corresponding logic level is quite a difficult task, so with increasing number of levels, the complexity of the converters increases, as well as the error rate.

- **NAND flash internal structure** (see Figure 2): The hierarchical structure of NAND Flash starts at a cell level which establishes strings, then pages, blocks, planes and ultimately a die. A string is a series of connected NAND cells in which the source of one cell is connected to the drain of the next one. Depending on the NAND technology, a string typically consists of 32 to 128 NAND cells. Strings are organised into pages which are then organised into blocks in which each string is connected to a separate line called a bit line. All cells with the same position in the string are connected through the control gates by a word line. A plane contains a certain number of blocks that are connected through the same bit line. A flash die consists of one or more planes, and the peripheral circuitry that is necessary to perform all the read / write / erase operations.

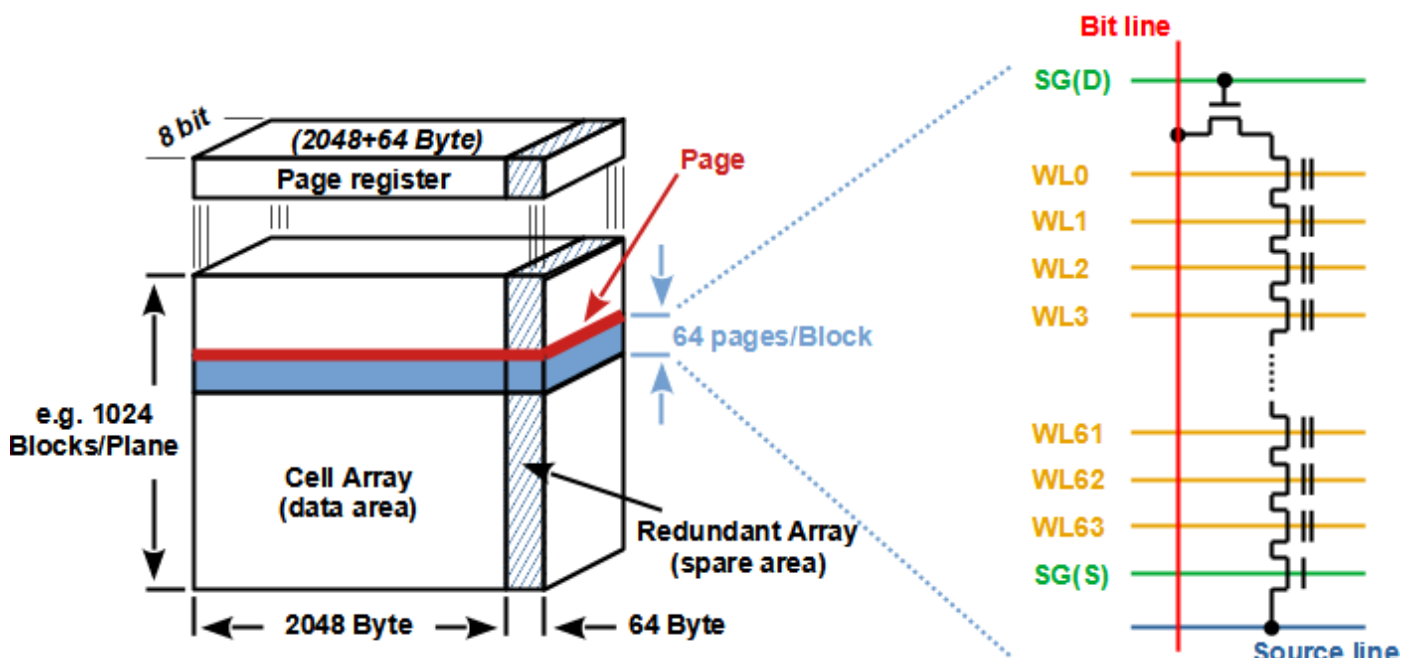


Figure 2: NAND flash internal structure.

- A **NAND flash page** consists of data area and spare area (in various sources may be referred also as “redundant” or “out-of-boundary (OOB)” area). Typically, the data area is used to store a payload data, while the spare area stores NAND flash management related data – ECC checksums, logical block / page numbers, usage counters, etc.
- **Erasing NAND flash:** Flash memory allows only two states – erased and non-erased. Particular bit of data can be written only if the media is in erased state. Once written, the bit is considered not usable for other write operation. Write operation on flash device can be accomplished only on erased units, so a write operation must be preceded by an erase operation. Only erase operation can revert programmed cells to erased state. During the erase, all cells on a bit string (see Figure 2) are erased, forming a huge block of flash – so called erase block. As a result, it is not possible to erase single cell in flash array. During the erasing, the ready / busy signal (R/B# pin or ready / busy flag in STATUS register) is low to indicate that the

device is in busy state. At the end of erase process, all cells in a block are checked if they are in erased state and the result is reported via pass / fail flag in STATUS register.

- **Programming** NAND flash: NAND flash devices are programmed on a page-by-page basis. A page is written into page register and then programmed into memory array. During the programming, the ready / busy signal (R/B# pin or ready / busy bit in STATUS register) is low to indicate that the device is in busy state. At the end of programming process, all bits on a page expected to be programmed to 0 are checked and the result is reported in pass / fail flag in STATUS register.
- **Reading** NAND flash: NAND flash devices are read on a page-by-page basis. Bits in a flash cell array are read by changing the voltage on rows and columns of cells followed by accessing the results. A page is moved from flash cell array into page register. During a page preparation, the ready / busy signal (R/B# pin or ready / busy bit in STATUS register) is low to indicate that the device is in busy state. After the page is ready, data can be lifted out of the device.

2. 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.
- Presence of invalid block(s) does not 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 should not be used for programming – data may be lost.
- Invalid block should not 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 block map building. Typical flowchart:

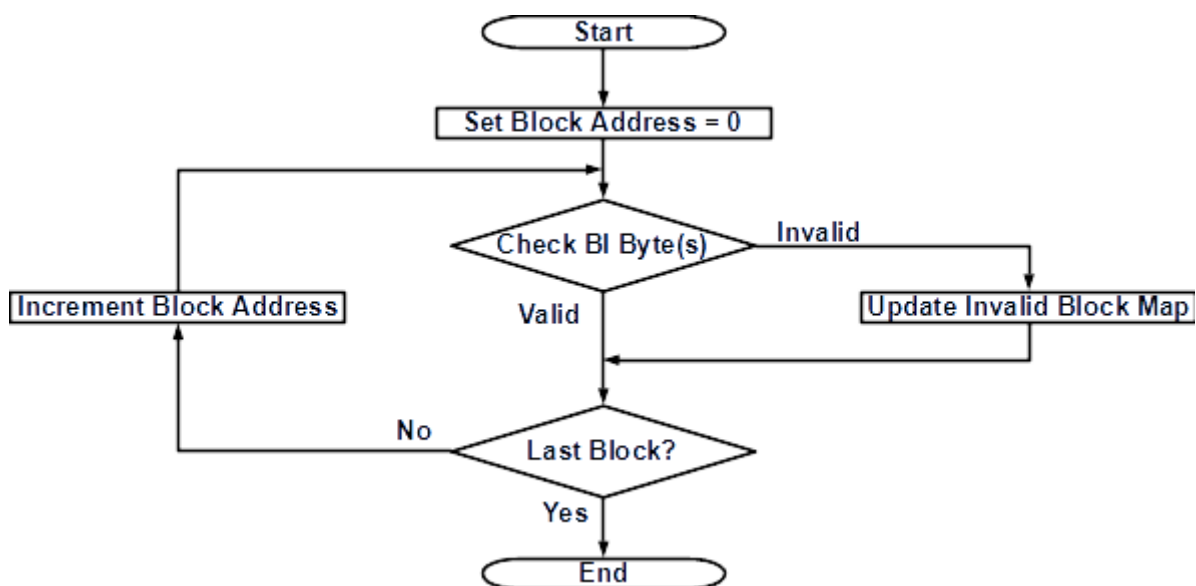


Figure 3: Invalid Block Map building flowchart.

- 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 levelling management used for new invalid blocks development prevention (see [Wikipedia](https://en.wikipedia.org/wiki/Wear_levelling) for more information). These techniques are used during end-appliance usage and, usually, are not relevant to know before pre-programming NAND flash device.

3. Brief Comments On Bit Errors

- Typically, bit errors are temporary errors 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.
- Bit errors may be detected and recovered. Various ECC algorithms are used for this purpose. Typical representatives are Hamming, BCH (Bose – Chaudhuri – Hocquenghem) and RS (Reed – Solomon) algorithms (see Wikipedia for more information on [Hamming](#), [BCH](#) and [RS](#)).
- 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 required minimum ECC parameters (e.g. 4 bit errors recovery in 512 bytes frame). 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 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 verification options**.

See chapter **Errors in NAND flash – the background** for detailed information about NAND flash errors origin.

4. Two Factors That Programmer Relies On

- **The user:** Programmer will do only what user has ordered to do. Programmer can detect device boundary exceeding, but cannot foretell e.g. a block from where data should start. Please, do not rely on default settings. Those are just some general preferences originating from device parameters and algorithm simplifying rather than from your particular needs.
- **NAND flash 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.

5. 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:

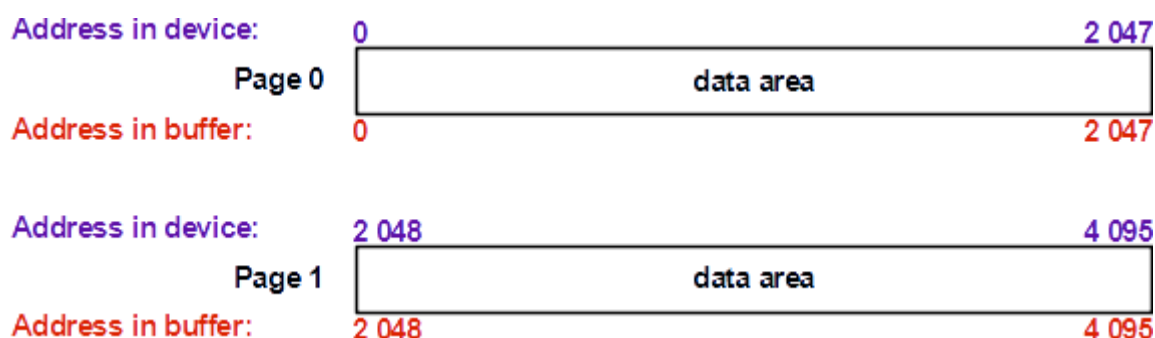


Figure 4: Buffer data layout, if spare area is not used.

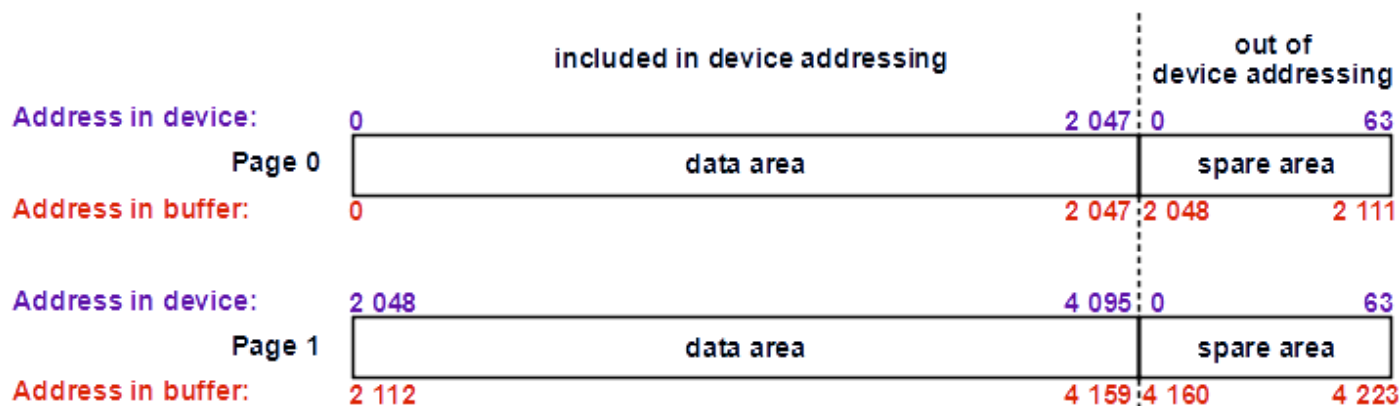


Figure 5: Buffer data layout, if spare area is used.

Considering a common NAND flash device with 2 048 + 64 bytes in a page and 64 pages in a block, the first byte of second block in device will be addressed using offset 2 0000h in device, but using offset 2 1000h in buffer. It is crucial to keep this in mind, especially if working with partitions.

6. 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.

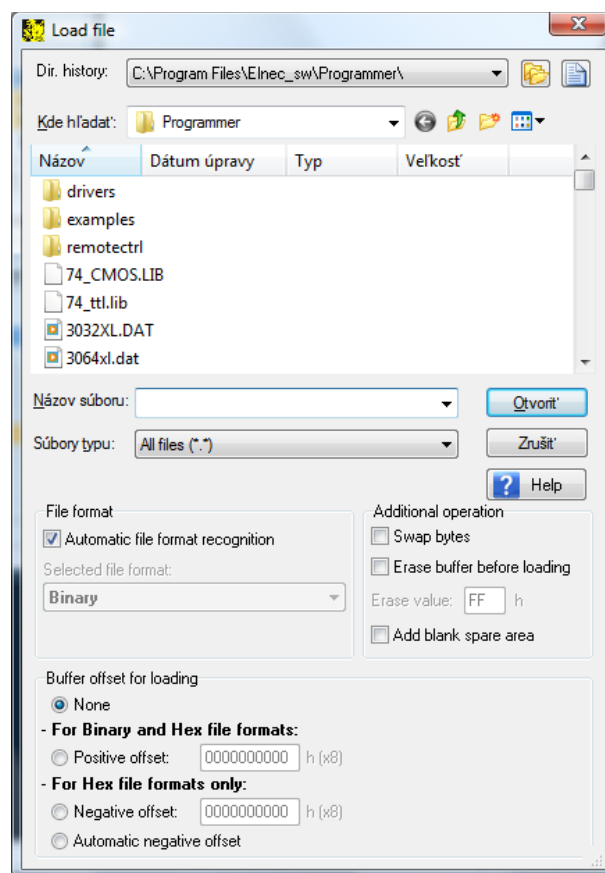


Figure 6: Load file dialog window.

Data image file should correspond with a copy of NAND flash device free of 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 does not contain it (relevant mainly for partitioning techniques), you can add blank (all FFh) spare area automatically on image load allowing **Add blank spare area** option (see Figure 6, *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.

6.1. 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 use **Positive offset** option (see Figure 6, *Buffer offset for loading* panel). You may compute the offset using following formula:

$$\text{positive_offset} = \text{block_number} \times \text{pages_in_block} \times \text{page_size}$$

where:

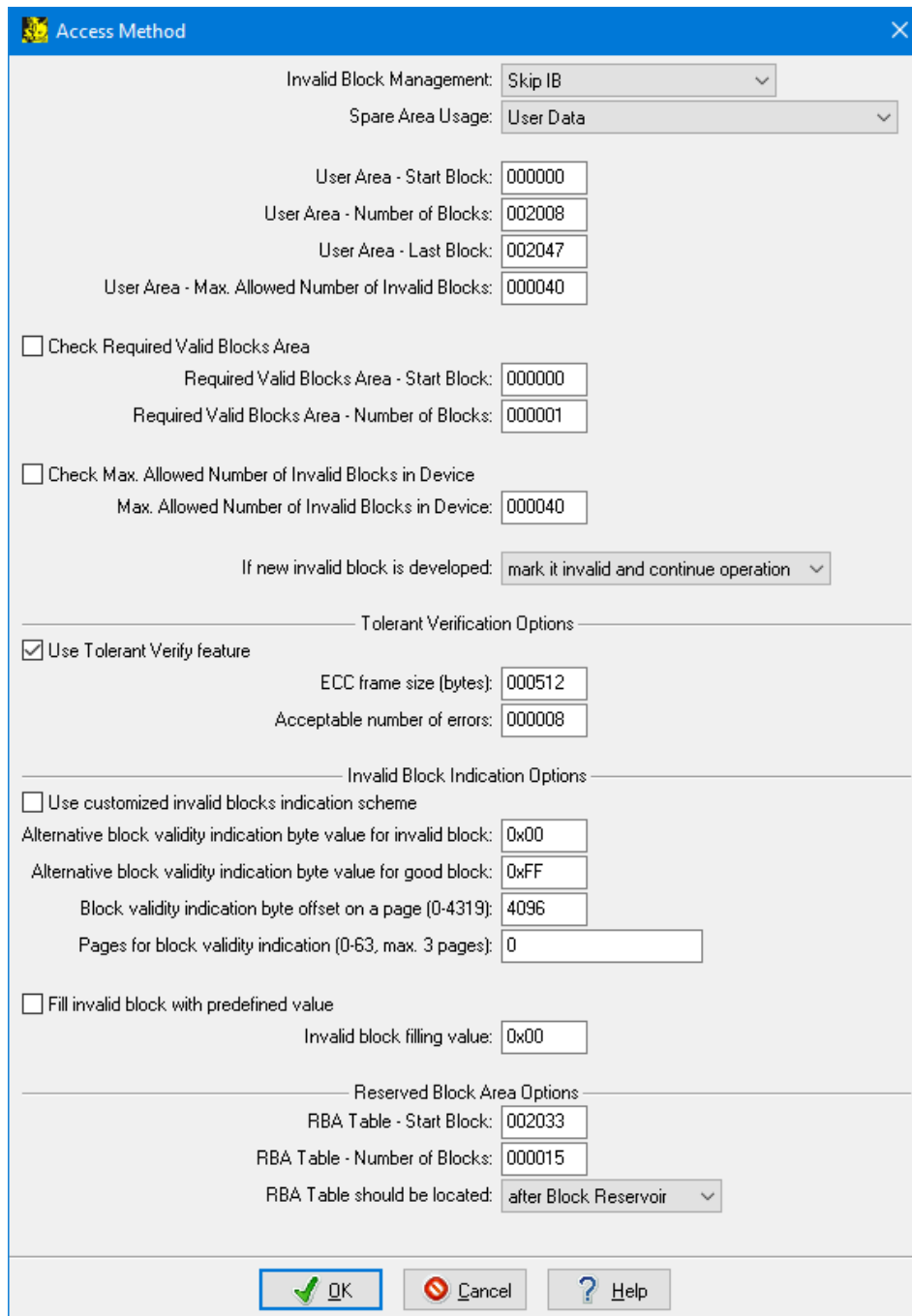
block_number is the number of target block as is mapped in buffer. Blocks ordering in buffer may differ from their real ordering in device, see buffer to device mapping in chapter dedicated to respective invalid blocks management technique.

pages_in_block is the count of pages in one block, as is given in your NAND flash 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 flash 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.

7. Access Method Window



Access Method

Invalid Block Management: Skip IB

Spare Area Usage: User Data

User Area - Start Block: 000000

User Area - Number of Blocks: 002008

User Area - Last Block: 002047

User Area - Max. Allowed Number of Invalid Blocks: 000040

☐ Check Required Valid Blocks Area

Required Valid Blocks Area - Start Block: 000000

Required Valid Blocks Area - Number of Blocks: 000001

☐ Check Max. Allowed Number of Invalid Blocks in Device

Max. Allowed Number of Invalid Blocks in Device: 000040

If new invalid block is developed: mark it invalid and continue operation

Tolerant Verification Options

☒ Use Tolerant Verify feature

ECC frame size (bytes): 000512

Acceptable number of errors: 000008

Invalid Block Indication Options

☐ Use customized invalid blocks indication scheme

Alternative block validity indication byte value for invalid block: 0x00

Alternative block validity indication byte value for good block: 0xFF

Block validity indication byte offset on a page (0-4319): 4096

Pages for block validity indication (0-63, max. 3 pages): 0

☐ Fill invalid block with predefined value

Invalid block filling value: 0x00

Reserved Block Area Options

RBA Table - Start Block: 002033

RBA Table - Number of Blocks: 000015

RBA Table should be located: after Block Reservoir

OK Cancel Help

7.1. Invalid Block Management

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

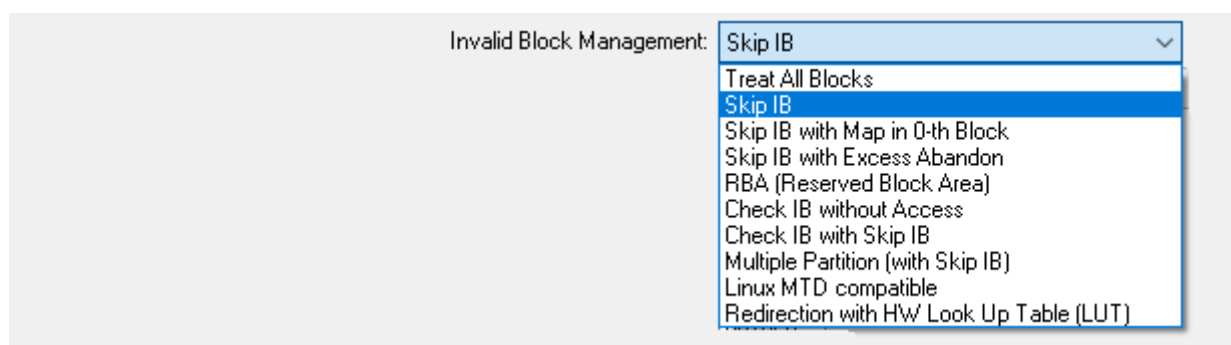


Figure 7: Invalid Block Management options

7.1.1. 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 identically, not regarding their real validity status.

The technique may be very helpful if dumping 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 does not 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 might fail on verify after programming (if enabled), however, if device is still used in end appliance, it might 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.

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**. The blocks will be programmed in device, not regarding their validity. If target block is invalid, data will be lost. The number of blocks specified for processing does not necessarily have to be the same size as 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.

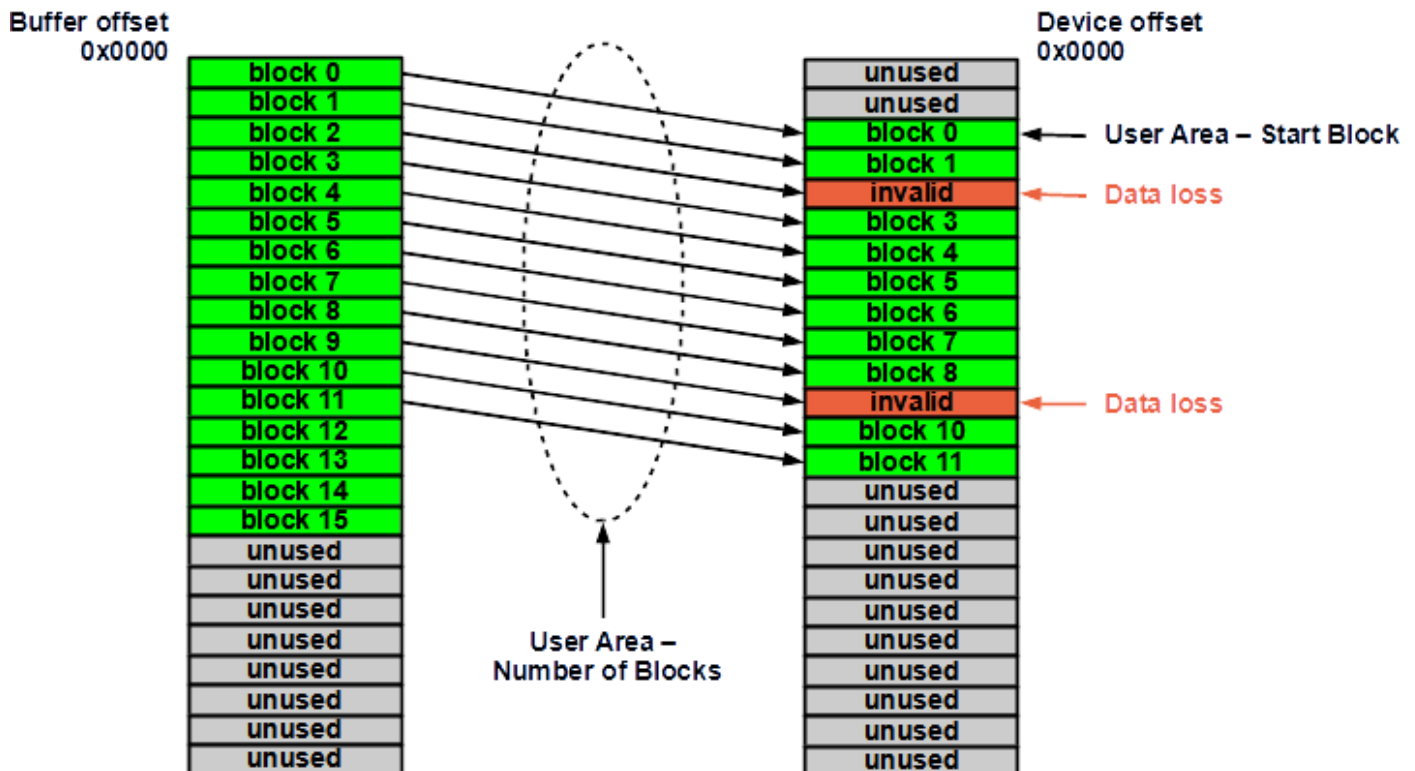


Figure 8: Treat All Blocks technique graphic representation.

7.1.2. 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. Excess data would be lost, therefore operation is halted at the first moment when such a condition is recognized (typically on initial Invalid Blocks Map building).

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 does not necessarily have to be the same size as 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 error.

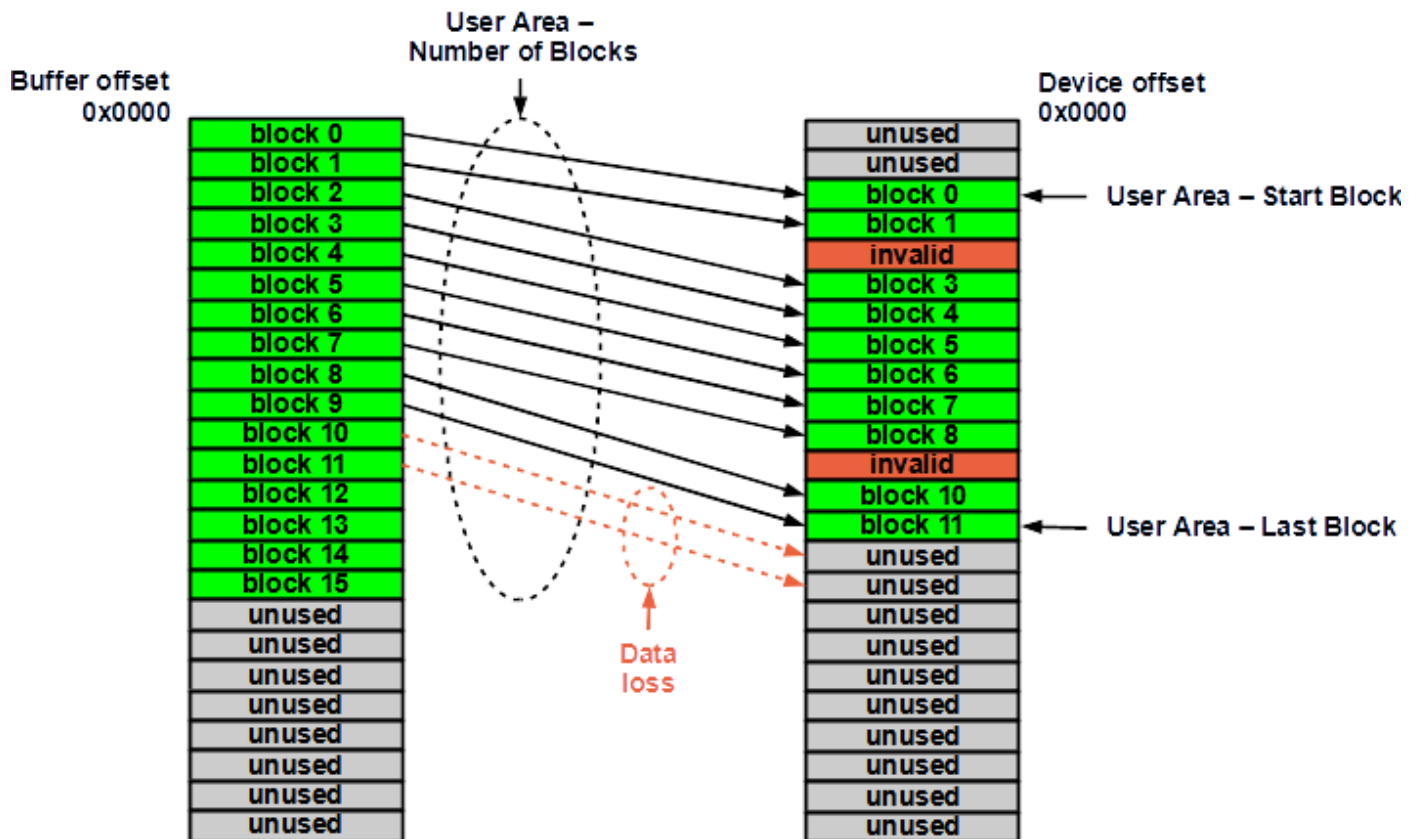


Figure 9: Skip IB technique graphic representation.

7.1.3. Skip IB with map in 0-th 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.

7.1.4. 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 does not generate error if this lossy condition happen. Data that cannot be programmed will be simply abandoned (lost).

This technique may be useful for applications where multiple data copies are used as a mean of error protection. Typical example is a bootloader storage. Another task where this technique may be useful is programming of various file system related headers into unused (padding) blocks. It can process all valid blocks and will not finish with error due to invalid blocks occurrence.

Compared to **Treat All Blocks** technique, **Skip IB with excess abandon** skips invalid blocks, so programmer does not expect any data there and verify operation can still succeed.

7.1.5. 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 10.

Figure 11 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 10.

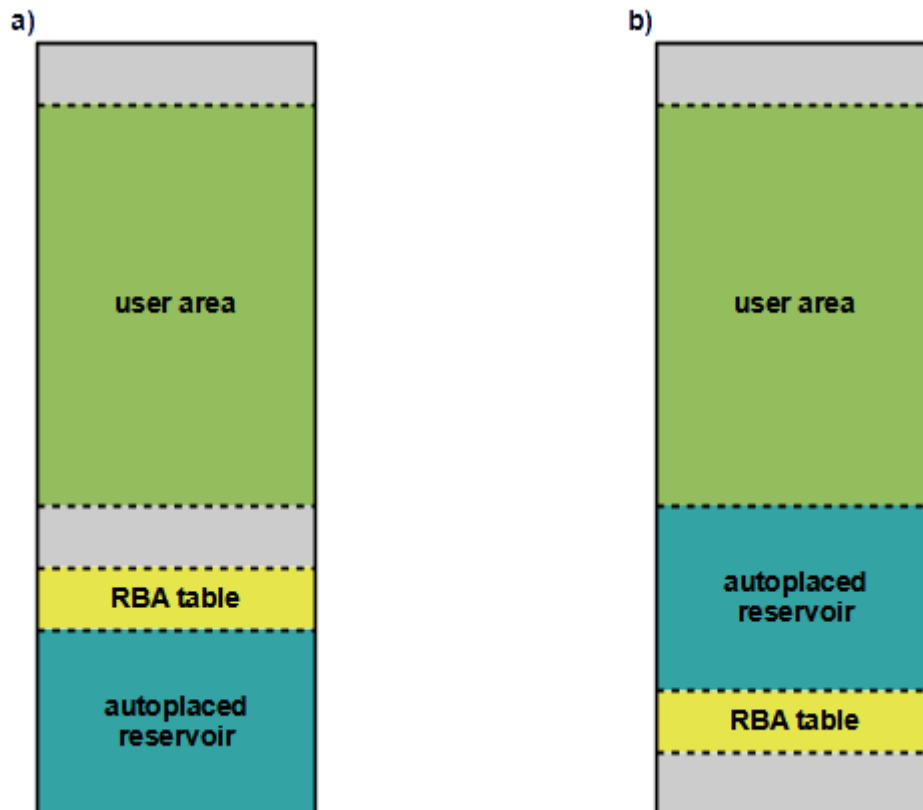


Figure 10: Device layout depending of **RBA Table should be located** option value: **before Block Reservoir** (a) and **after Block reservoir** (b). There may be unused blocks accepted in grey areas.

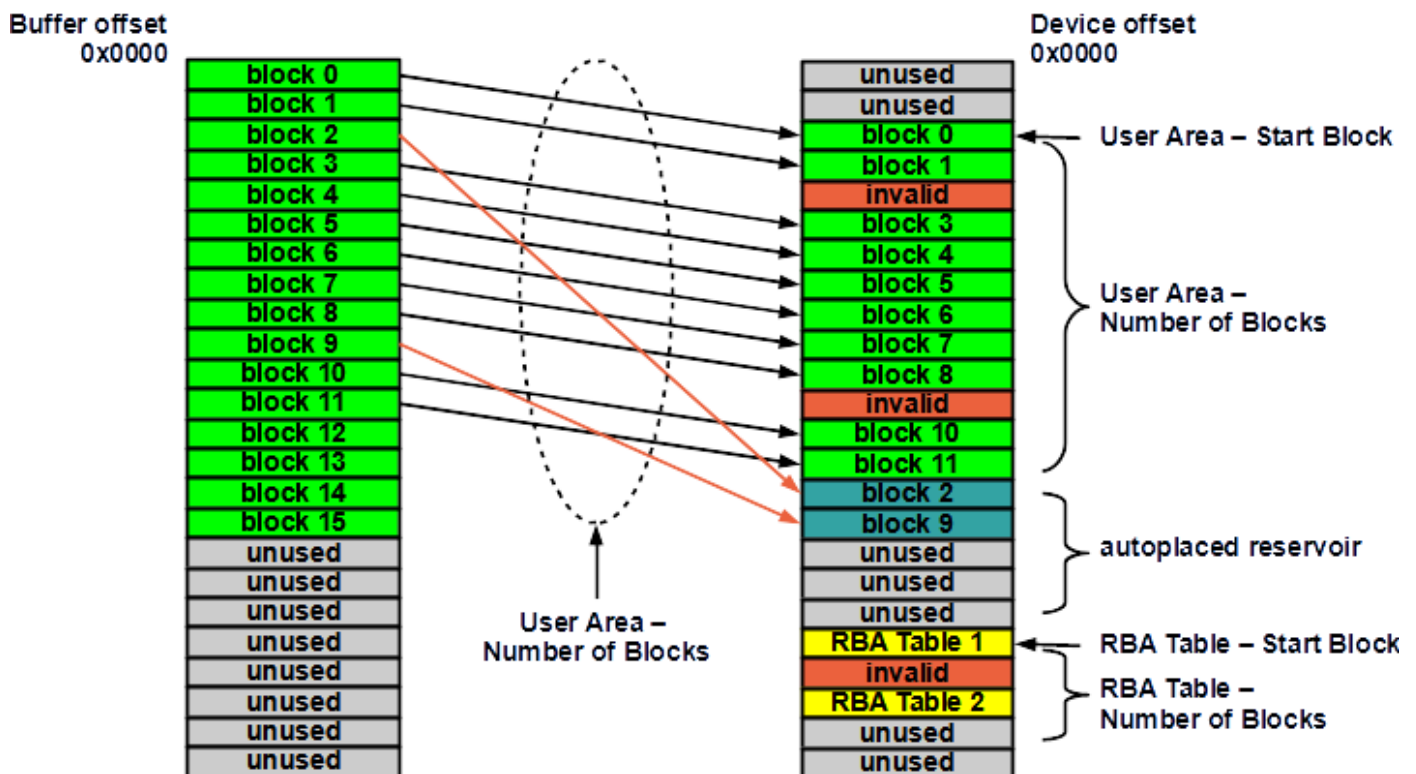


Figure 11: **RBA** technique graphic representation.

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 error.

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 a header. The header is always of the same value FDFEh.

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 their respective substitution blocks from blocks reservoir used for replacement. Single page can hold information about $(\text{page_data_area_size} - 4) / 4$ redirections.

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

7.1.6. Check IB without access

Check IB without access technique performs checks with regard to set rules, but does not execute any other device 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 to simulate the programming, e.g. if you are going to perform initial programming of the NAND flash device after assembling the end-appliance. In such case you may be interested in not using memories 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, see chapter **Required Valid Blocks Area options**.

If enabled, **Max. Allowed Number of Invalid Blocks in Device** is checked.

7.1.7. Check IB with Skip IB

After performing all tests in the same manner as if **Check IB without access** technique has been 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.

7.1.8. Discard Invalid block(s) data

Note: This invalid blocks management technique can be activated only for techniques based on partitioning, and only using *.CSV partition definition file.

Discard invalid block(s) data is a hybrid of **Treat All Blocks** and **Skip IB** techniques. If programmer meets invalid block in device, it simply increases the pointer one block forward in both, device and buffer. Data belonging to invalid block are discarded (ignored).

This technique is intended particularly for programming various bootloaders and data tables, when multiple copies are used as an instrument of error protection.

7.1.9. Multiple partitions with Skip IB

Note: This invalid blocks management technique 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 do not 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 the 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 by 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 the block specified by value of **Partition end** is reached and not all specified blocks are read, operation will close with warning.

Figure 12 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 could not 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).

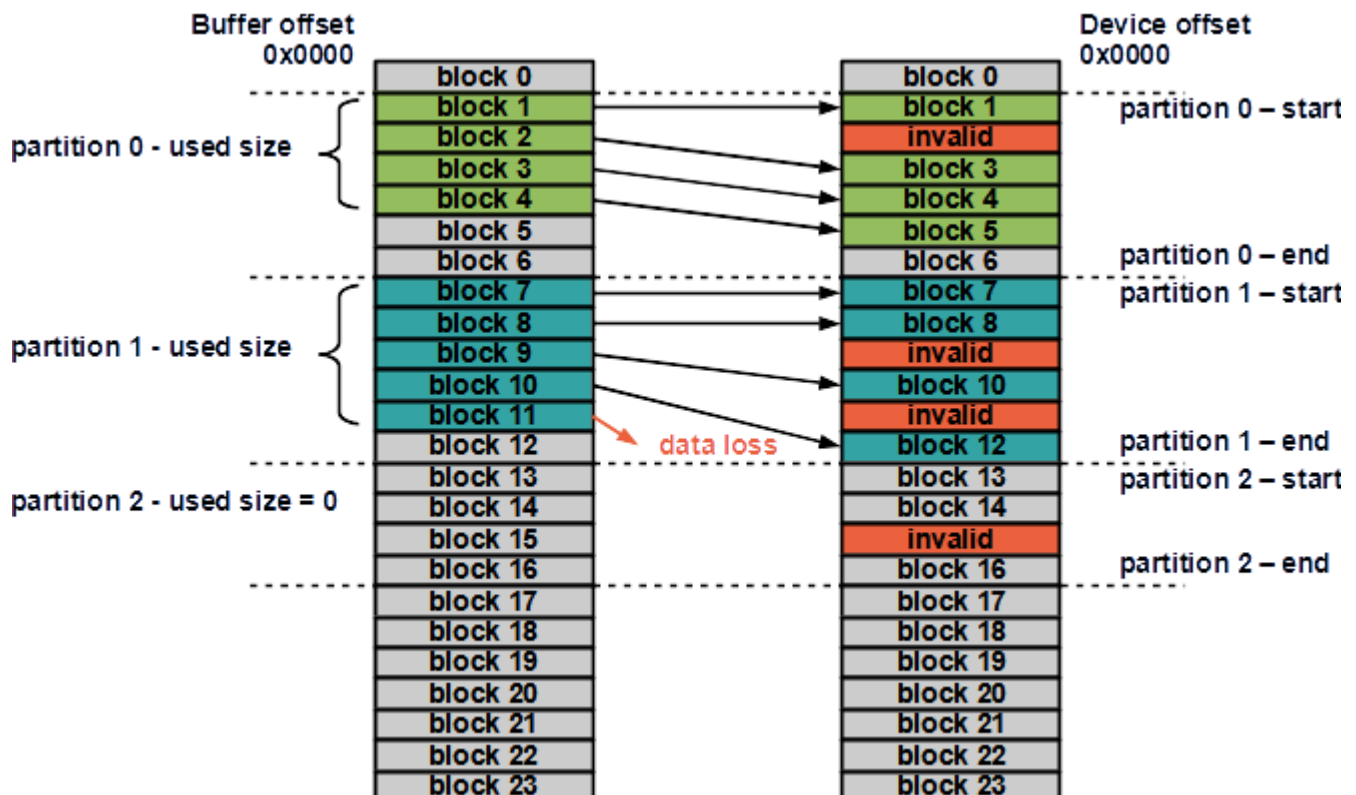


Figure 12: Multiple partitions with Skip IB technique graphic representation.

7.1.9.1. 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. All partition definition file formats are described in following chapters.

7.1.9.1.1. Qualcomm multiply partition format (*.mbn)

Note: The support of this format is implemented based only on fragment of specification available from our customer(s). Therefore it cannot be considered full and reliable. If you observe any problems, please, contact our technical support with full MBN file format specification.

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

7.1.9.1.1.1. 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 (typically 256 bytes) and contains partition table definition. Load this file using menu **File / Load Partition table**, see chapter **Loading Partition definition file**. Actually, the maximum count of supported partitions is 64.

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**.

7.1.9.1.1.2. 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 **Block validity indication byte offset on a page** 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.

7.1.9.1.2. Comma separated values format (*.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 20h) 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 obviously `partition_end – partition_start > used_partition_size`. Enter the count 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 `FFFFFFFFh` here to ensure future compatibility (or `0xFFFFFFFF` whichever form you prefer; keep 4 bytes size).

Special options 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:

- FFFh = 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:

- 0 = Treat All Blocks
- 1 or Fh = Skip IB (default)
- 2 = Skip IB with excess abandon
- 3 = Check IB without access
- 4 = Discard Invalid block(s) data

Note: It is possible to specify an equivalent of Check IB with Skip IB technique using Skip IB (1h or Fh) technique and non FFFh value for Maximum allowed number of invalid blocks in partition.

bits 22:16 – Reserved for future use, consider 7Fh 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 27:24 – File system preparation:

- Fh = feature disabled (default)
- 0 = JFFS2 Clean Markers are written to unused blocks at respective partition end using MSB byte ordering (big endian)
- 1 = JFFS2 Clean Markers are written to unused blocks at respective partition end using LSB byte ordering (little endian)

bits 31:28 – Reserved for future use, consider Fh value for future compatibility.

Note: other values not specified here may be accepted for customized implementations, depending on algorithm specification.

- **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; 0xff7ffffff; boot
101; 200; 50; 0xff7ffffff; exec
201; 300; 0; 0xff7f3010; res1
301; 400; 50; 0xff7ffffff; 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 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.

7.1.9.1.3. Group define format (*.def)

Note: The support of this format is implemented based only on fragment of specification available from customer(s). Therefore it cannot be considered full and reliable. If you observe any problems, please, contact our technical support with full DEF 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 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**.

7.1.9.1.4. Loading Partition 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**.

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).

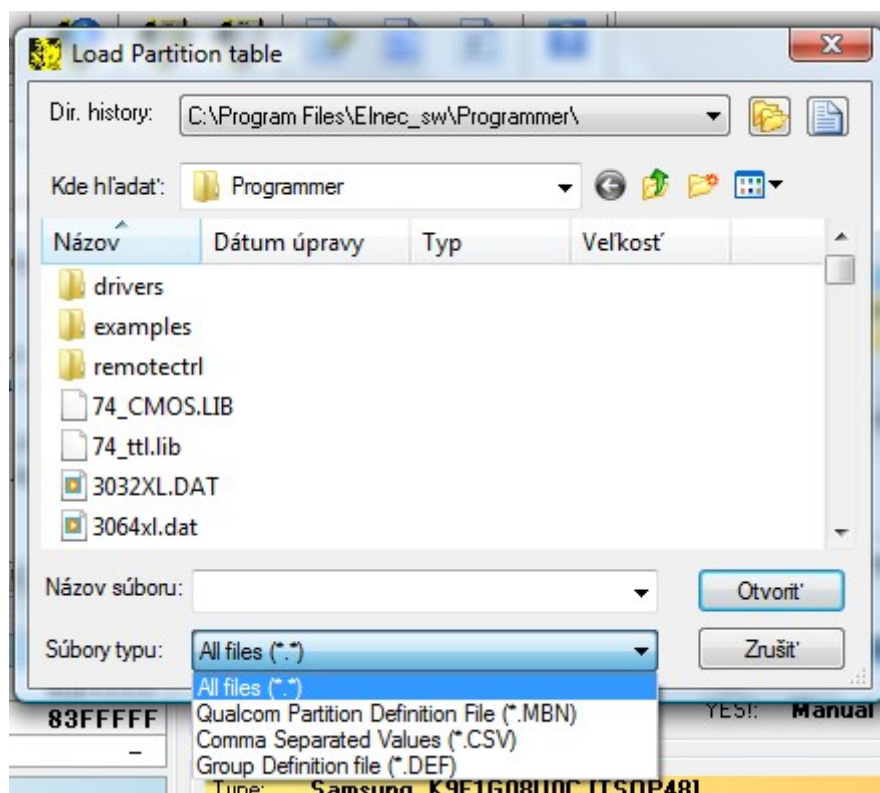


Figure 13: Load partition table window.

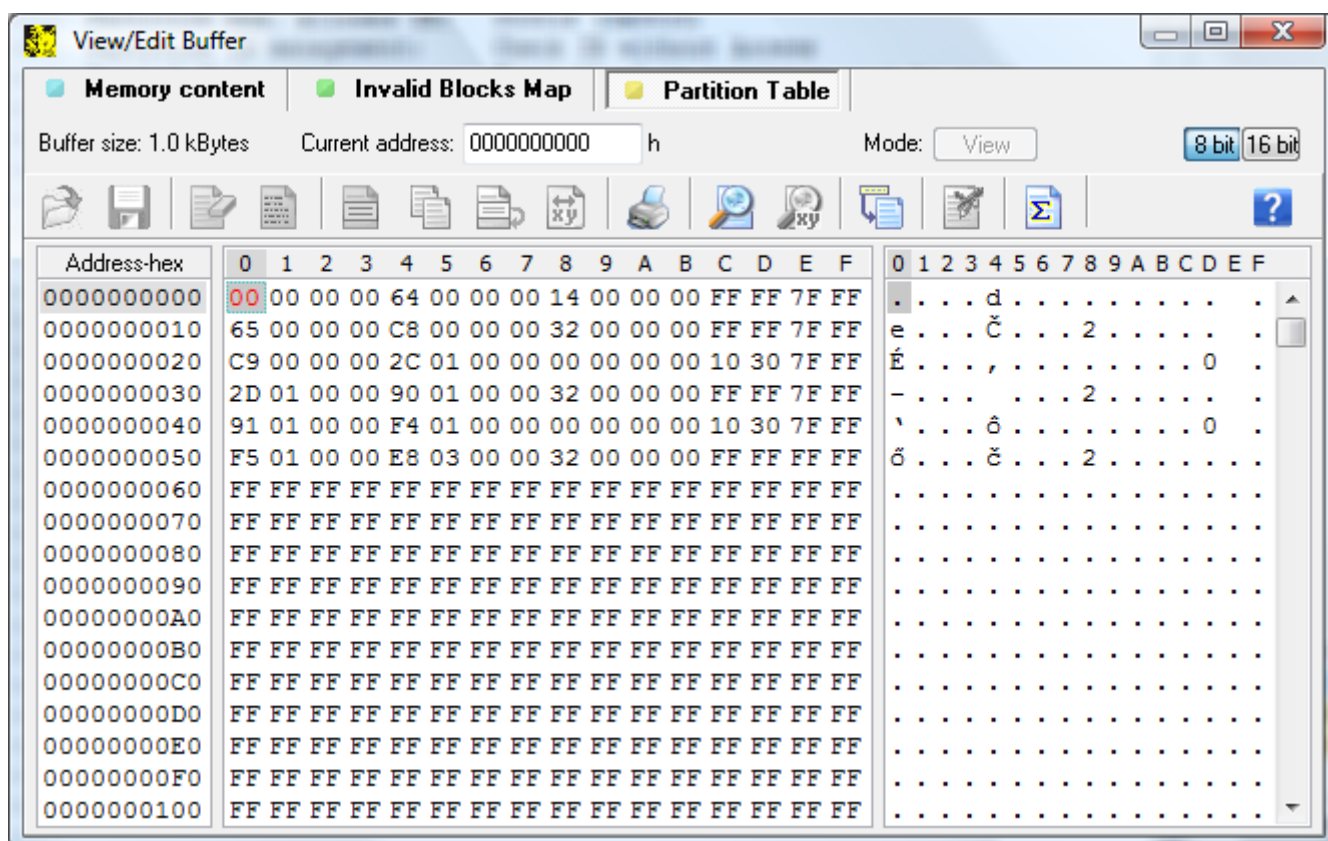


Figure 14: Example of partition table stored in buffer.

```

Programmer activity log
L0175:
L0176: >> 17.09.2013, 19:49:28
L0177: Loading file: C:\Program Files\Elneec_sw\Programmer\parttable.csv
L0178: File format: Comma Separated Values
L0179: Loading Partition Table...
L0180: Checking Partition Table...
L0181: Checking Partition Table - O.K.
L0182: Partition 0
L0183: Partition start (in blocks): 000000 (0x0000)
L0184: Partition end (in blocks): 000100 (0x0064)
L0185: Partition size (in blocks): 000020 (0x0014)
L0186: Partition IB management: Skip IB
L0187: Special features: First block in partition must be good
L0188: Comment: boot
L0189: Partition 1
L0190: Partition start (in blocks): 000101 (0x0065)
L0191: Partition end (in blocks): 000200 (0x00C8)
L0192: Partition size (in blocks): 000050 (0x0032)
L0193: Partition IB management: Skip IB
L0194: Special features: First block in partition must be good
L0195: Comment: exec
L0196: Partition 2
L0197: Partition start (in blocks): 000201 (0x00C9)
L0198: Partition end (in blocks): 000300 (0x012C)
L0199: Partition size (in blocks): 000000 (0x0000)
L0200: Partition max. allowed IB: 000016 (0x0010)
L0201: Partition IB management: Check IB without Access
L0202: Special features: First block in partition must be good
L0203: Comment: res1
L0204: Partition 3
L0205: Partition start (in blocks): 000301 (0x012D)
L0206: Partition end (in blocks): 000400 (0x0190)
L0207: Partition size (in blocks): 000050 (0x0032)
L0208: Partition IB management: Skip IB
L0209: Special features: First block in partition must be good
L0210: Comment: fsys
L0211: Partition 4
L0212: Partition start (in blocks): 000401 (0x0191)
L0213: Partition end (in blocks): 000500 (0x01F4)
L0214: Partition size (in blocks): 000000 (0x0000)
L0215: Partition max. allowed IB: 000016 (0x0010)
L0216: Partition IB management: Check IB without Access
L0217: Special features: First block in partition must be good
L0218: Comment: res2
L0219: Partition 5
L0220: Partition start (in blocks): 000501 (0x01F5)
L0221: Partition end (in blocks): 001000 (0x03E8)
L0222: Partition size (in blocks): 000050 (0x0032)
L0223: Partition IB management: Skip IB
L0224: Special features: None in use
L0225: Comment: data
L0226: File loading successful.

```

Figure 15: Successful partition definition file load listing example in log window (an example from CSV format description was used).

7.1.9.1.4.1. Error codes on Partition definition file load

During the partition definition file loading, several errors can occur. Errors are always displayed in pg4uw log-window. Error message consists of error code and error description in the following form:

File loading problem!

Error code: #xyyy – error description

where xx stands for file format:

- 00 – binary file
- 01 – *.mbn file containing just partition table
- 02 – *.mbn file containing both, partition table as well as partitions data
- 03 – *.csv file containing partition table
- 04 – reserved for future use
- 05 – *.def file containing partition table

and yy stands for error type:

- 10 – disk i/o error (disk i/o error, file access error, ...)
- 11 – maximum buffer limit exceeded (file size greater than max. supported buffer size)
- 12 – unable to re-allocate buffer (file size is greater than buffer size and there is some problem when reallocating the buffer (e.g. not enough disk space))
- 13 – unknown separator (for *.csv files, nor comma (,) nor semicolon ;) were detected as separator)
- 14 – file does not specify any partition (none partition definition read)
- 15 – too many partitions specified in file (max. 16 partitions are supported for Qualcomm Multiply Partition, max. 64 partitions generally)
- 16 – incorrect numeric values format (typo in text-oriented files (*.csv), e.g. @34 instead of 234)
- 17 – version not supported (not supported version of algorithm specification detected)
- 18 – invalid file header (damaged header, some mandatory item missing, ...)

7.1.9.2. 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 options**
- **Max. Allowed Number of Invalid Blocks in Device options**
- **Invalid Block Indication options (Extended version)**
- **Tolerant verification options**
- **Special device features** (if supported)

Please, see respective chapters for detailed informations.

7.1.9.3. 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 **Access method window**.
5. Save your settings and data into project file and test the operation.

7.1.10. Linux MTD compatible

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 (BBT). 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, see chapter **ECC – Hamming (2×256 byte frame) variant 1 and 2**. 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 such device.

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

See chapter **Linux MTD compatible options** for more information about available options.

7.1.11. Redirection with HW Look Up Table (LUT)

This is a kind of redirection technique based on device internal hardware infrastructure. Once the block redirection link is created, it is permanent. New valid block is accessible while still addressing original invalid block.

The technique is available only for devices providing necessary hardware support. See your device datasheet for more information on this topic.

At time of this revision of the app-note release, we do not support HW-LUT for stacked (mutli-chip) devices.

7.2. Spare Area Usage

Our programmers support several modes of spare area usage. Any other spare area usage mode can be supported upon user's request.

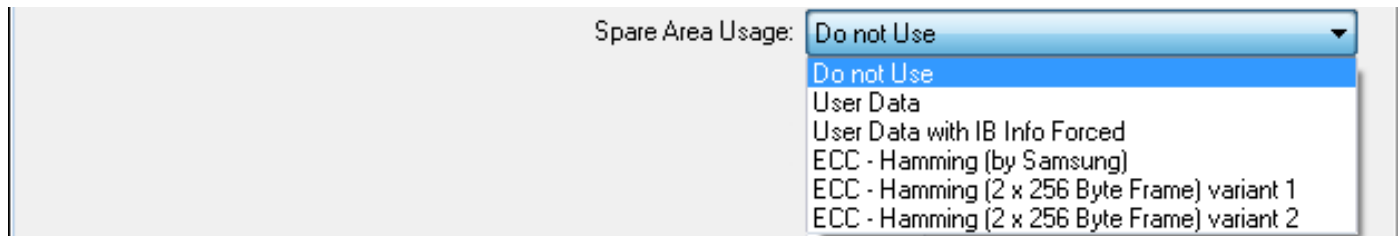


Figure 16: Spare area usage options.

7.2.1. 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 4 in chapter **Loading data into pg4uw control software buffer**) nor programmed in or read from target device, respectively.

7.2.2. User data

User data mode treats spare area as is, without any change. Spare area data are both, expected in buffer (see Figure 5 in chapter **Loading data into 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 FFh (or FFFFh for x16 devices).

7.2.3. 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 FFh (or FFFFh for x16 devices).

User can change BI byte position from default using **Invalid Block Indication options (Extended version)** and related settings.

7.2.4. 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 documents also from our archive: [256 byte frame](#), [512 byte frame](#)

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 2 048 + 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 17.

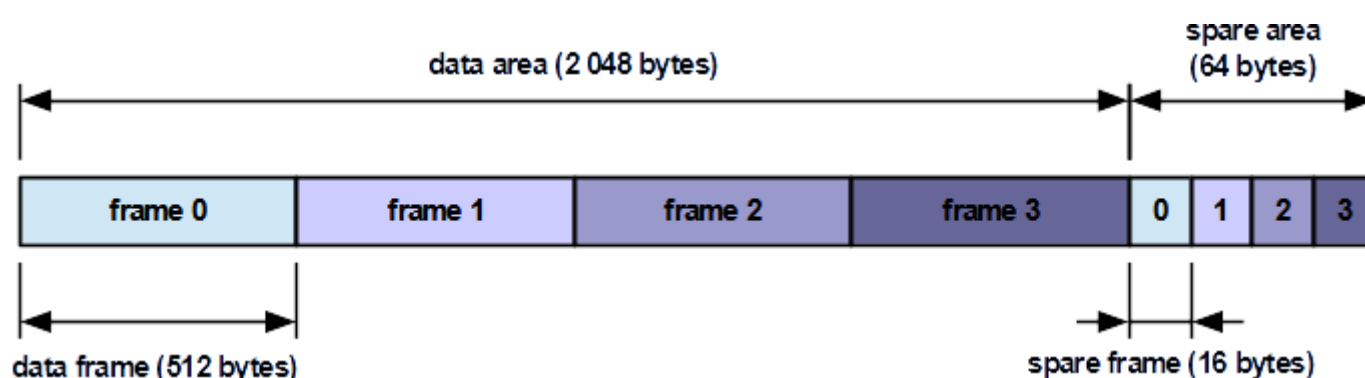


Figure 17: ECC - Hamming (by Samsung) page segmentation example.

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 18 and Figure 19 for layouts. Reserved bytes are not used and are left blank.

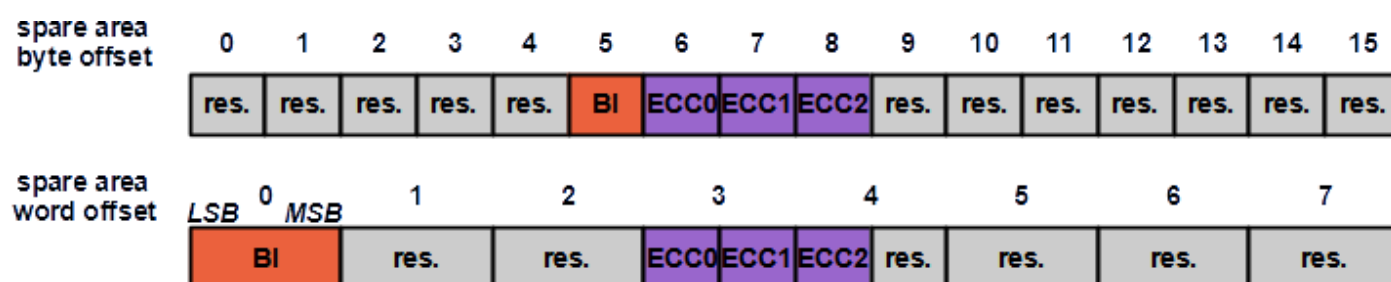


Figure 18: ECC Hamming (by Samsung) spare area layout for small page (512+16 bytes).

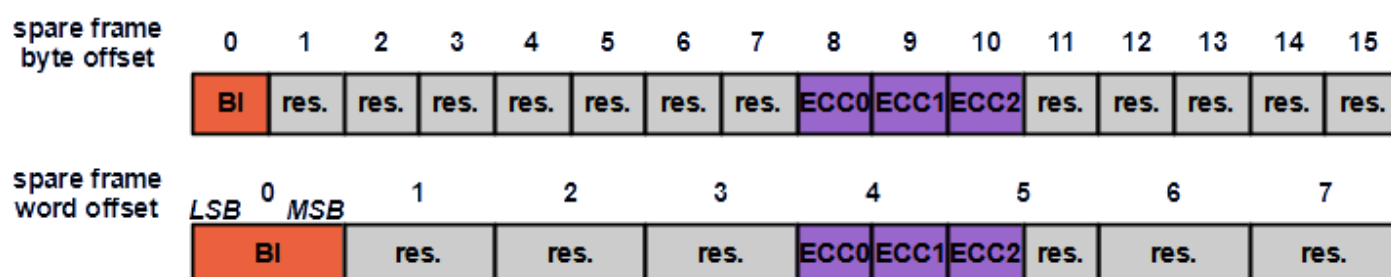


Figure 19: ECC Hamming (by Samsung) spare area layout for large page (2 048+64 bytes).

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 temporary internal 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.

7.2.5. ECC – Hamming (2×256 byte frame) variant 1 and 2

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 (2×256 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 (2×256 byte frame) **variant 1** to ECC Hamming (2×256 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 temporary internal 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.

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 (2x256 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 (2x256 byte frame) variant 1 spare area layout for 2 048 + 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 (2x256 byte frame) variant 1 spare area layout for 4 096 + 128 byte page (variant 2 differs in ECC[0] - ECC[1] ordering).

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 foregoing tables are not affected by ECC Hamming (2x256 byte frame) spare area usage mode. Existing data in those areas are preserved.

7.3. Device Internal ECC Controller Options

Some modern NAND flash devices incorporate built-in internal ECC controller. It is a special hardware logic device 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.

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

Figure 20: Device internal ECC controller options.

7.3.1. Enable device internal ECC controller

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

Default value: Disabled

7.4. User Area Options

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.

User Area - Start Block:	<input type="text" value="000000"/>
User Area - Number of Blocks:	<input type="text" value="001004"/>
User Area - Last Block:	<input type="text" value="001023"/>
User Area - Max. Allowed Number of Invalid Blocks:	<input type="text" value="000020"/>

Figure 21: User Area options.

7.4.1. 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 operations do not 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 value: 0 (device block #0000)

Note: Both, decimal and hexadecimal forms are accepted for this option (e.g. 1023, 3FFh, 0x3FF – you can use the form whichever you prefer). For hexadecimal, suffix “h” or prefix “0x” must be used, as is shown in example.

7.4.2. 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 operations do not 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 value: 98% of all blocks in target device (typically, manufacturers guarantee less than 2% of invalid blocks, mainly for SLC devices), or minimum valid blocks count in device if specified in target device datasheet

Note: Both, decimal and hexadecimal forms are accepted for this option (e.g. 1023, 3FFh, 0x3FF – you can use the form whichever you prefer). For hexadecimal, suffix “h” or prefix “0x” must be used, as is shown in example.

7.4.3. User Area – Last Block

Option **User area – last block** specifies the ordinal number of physical block in target device that operation must not exceed. Device area beyond this block must 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 operations do not 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 value: last physical block in device

Note: Both, decimal and hexadecimal forms are accepted for this option (e.g. 1023, 3FFh, 0x3FF – you can use the form whichever you prefer). For hexadecimal, suffix “h” or prefix “0x” must be used, as is shown in example.

7.4.4. User Area – Max. Allowed Number of Invalid Blocks

Option **User area – max. allowed number of invalid blocks** specifies the maximum count of invalid blocks 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 operations do not take this option into account – respective check is not performed.

Default value: A difference from default **User Area – Number of Blocks** and all blocks in target device.

Note: Both, decimal and hexadecimal forms are accepted for this option (e.g. 1023, 3FFh, 0x3FF – you can use the form whichever you prefer). For hexadecimals, suffix “h” or prefix “0x” must be used, as is shown in example.

7.5. Required Valid Blocks Area Options

Required valid blocks area options may be used to specify a special 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 do not 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.

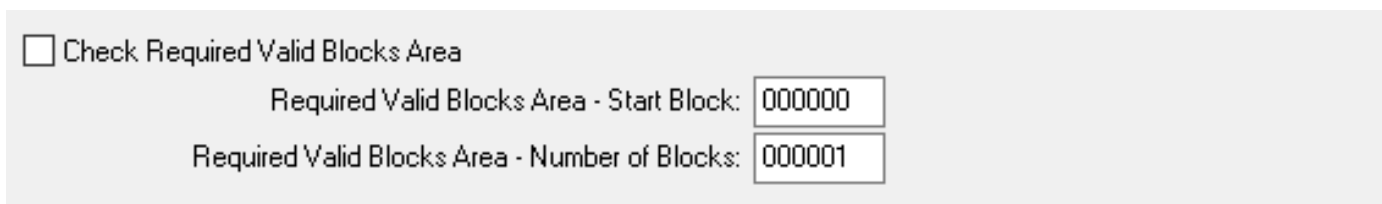


Figure 22: Required valid blocks area options.

7.5.1. Check Required Valid Blocks Area

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

Default value: Disabled

7.5.2. 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 value: 0

Note: Both, decimal and hexadecimal forms are accepted for this option (e.g. 1023, 3FFh, 0x3FF – you can use the form whichever you prefer). For hexadecimals, suffix “h” or prefix “0x” must be used, as is shown in example.

7.5.3. 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 value: 1

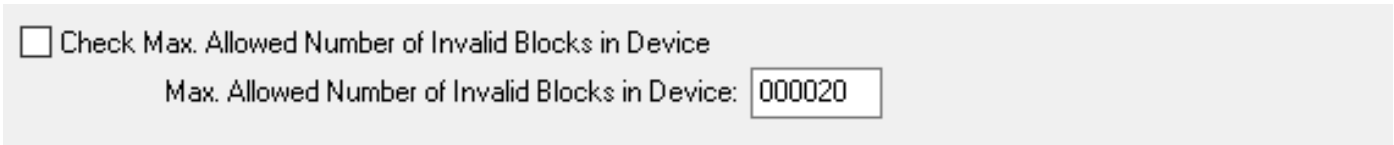
Note: Both, decimal and hexadecimal forms are accepted for this option (e.g. 1023, 3FFh, 0x3FF – you can use the form whichever you prefer). For hexadecimal, suffix “h” or prefix “0x” must be used, as is shown in example.

7.6. Max. Allowed Number Of Invalid Blocks In Device Options

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 entire 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 chapter **Required Valid Blocks Area options** – evaluated continually, as new invalid blocks may be developed during programming and / or erasing.



☐ Check Max. Allowed Number of Invalid Blocks in Device

Max. Allowed Number of Invalid Blocks in Device:

Figure 23: Max. allowed number of blocks in device options.

7.6.1. Check Max. Allowed Number of Invalid 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 value: Disabled

7.6.2. Max. Allowed Number of Invalid Blocks in Device

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

Default value: 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: Both, decimal and hexadecimal forms are accepted for this option (e.g. 1023, 3FFh, 0x3FF – you can use the form whichever you prefer). For hexadecimals, suffix “h” or prefix “0x” must be used, as is shown in example.

7.7. Behaviour On New Invalid Block Options

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

Important note:

Technically, only program and erase operations are capable to invoke new invalid blocks formation as they apply the highest voltage across the memory cells array. Read, verify and blank check operations may produce only reversible errors. 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.

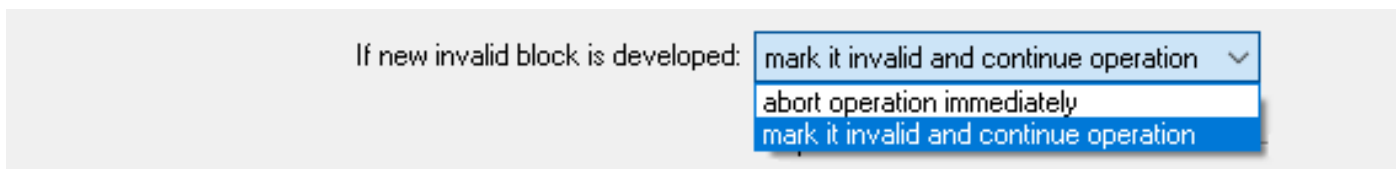


Figure 24: Behaviour on new invalid block options.

7.7.1. If new invalid blocks is developed

Select from drop-down menu:

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

Default value: mark it invalid and continue operation

7.8. 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 requirements 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 close his eyes to as many errors as the ECC algorithm can recover.

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

On verify and blank check operations, 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.

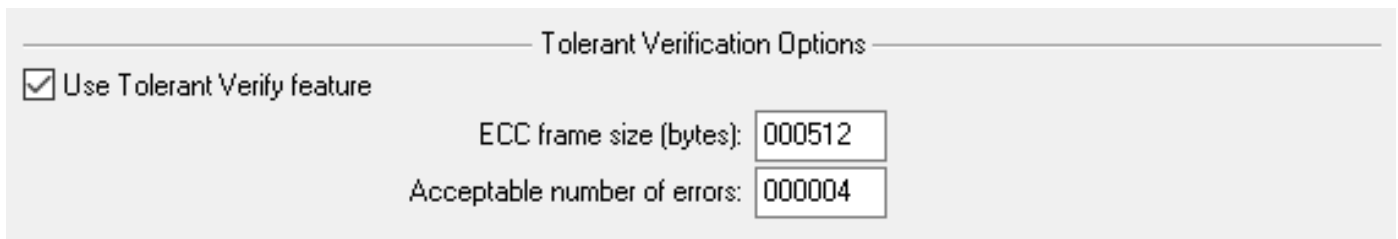


Figure 25: Tolerant verification options.

7.8.1. Use Tolerant Verify feature

Confirm the check-box to enable tolerant verification.

Default value: enabled

7.8.2. ECC frame size

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. Only those frame sizes will be accepted leading to integer frame count in page data area.

Default value: datasheet default (device dependent, typically 512 bytes)

Note: Both, decimal and hexadecimal forms are accepted for this option (e.g. 1023, 3FFh, 0x3FF – you can use the form whichever you prefer). For hexadecimal, suffix “h” or prefix “0x” must be used, as is shown in example.

7.8.3. Acceptable number of errors

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

Default value: datasheet default (device dependent, typically 1, 4 or 8 bits)

Note: Both, decimal and hexadecimal forms are accepted for this option (e.g. 1023, 3FFh, 0x3FF – you can use the form whichever you prefer). For hexadecimal, suffix “h” or prefix “0x” must be used, as is shown in example.

7.8.4. Tolerant verify examples

Example 1:

Let have a page of 2 048 data bytes + 128 spare bytes.

Tolerant verify parameters are 512 byte ECC frame size and 8 accepted errors.

Programmer will consider entire page as 4 full frames and 1 quarter frame (in spare area). In full frames, up to 8 erroneous bits will be accepted. On 9-th bit error found the error will be reported. For the last frame of quarter size, also quarter number of errors will be tolerated, i.e. up to 2 erroneous bits in this example.

Note: non-integer number of accepted errors for partial frame is always rounded up to next higher integer.

Example 2:

Let have a page of 2 048 data bytes + 128 spare bytes.

Tolerant verify parameters are 64 byte ECC frame size and 1 accepted error.

Programmer will consider entire page as 34 full frames – 32 frames in data area and other 2 frames in spare area. For each frame, max. 1 erroneous bit will be accepted.

7.9. Invalid Block Indication Options (Simplified Version)

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

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 marking schemes 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.

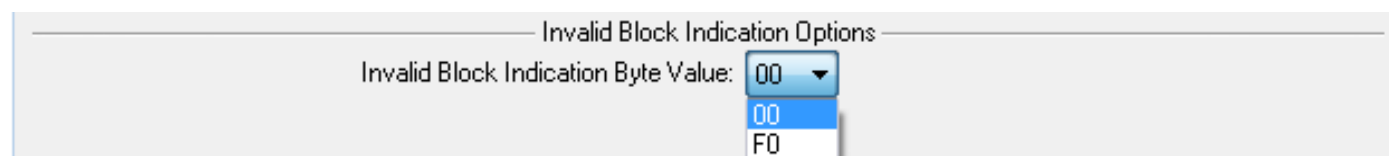


Figure 26: Invalid block indication options (simplified version).

7.9.1. 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:

- **00h** (0000h for x16 devices) – default value used by our programmers (datasheets typically specify a non-FFh value);
- **F0h** (F0F0h for x16 devices) – the value used in SmartMedia for acquired invalid blocks marking.

Default value: 00h (0000h for x16 devices)

7.10. Invalid Block 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 2 048+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.

Anyhow, the real scheme in device should be indicated using **Target device uses** option in **Device operation options window** 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 first two pages of an invalid block to 00h.

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 Options

☐ Use customized invalid blocks indication scheme

Alternative block validity indication byte value for invalid block:

Alternative block validity indication byte value for good block:

Block validity indication byte offset on a page (0-2111):

Pages for block validity indication (0-63, max. 3 pages):

☐ Fill invalid block with predefined value

Invalid block filling value:

Figure 27: Invalid blocks indication options (extended version).

7.10.1. Use customized invalid blocks indication scheme

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

Default value: Disabled

7.10.2. 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 value: 00h (0000h for x16 devices)

Note: Both, decimal and hexadecimal forms are accepted for this option (e.g. 1023, 3FFh, 0x3FF – you can use the form whichever you prefer). For hexadecimals, suffix “h” or prefix “0x” must be used, as is shown in example.

7.10.3. 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 value: FFh (FFFFh for x16 devices)

Note: Both, decimal and hexadecimal forms are accepted for this option (e.g. 1023, 3FFh, 0x3FF – you can use the form whichever you prefer). For hexadecimals, suffix “h” or prefix “0x” must be used, as is shown in example.

7.10.4. 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 value: first spare area byte (word for x16 devices)

Note: Both, decimal and hexadecimal forms are accepted for this option (e.g. 1023, 3FFh, 0x3FF – you can use the form whichever you prefer). For hexadecimal, suffix “h” or prefix “0x” must be used, as is shown in example.

7.10.5. Pages for block validity indication

Specifies up to three pages in a block used for BI byte (BI word for x16 devices) recognition.

Default value: datasheet default (device dependent)

Note: Both, decimal and hexadecimal forms are accepted for this option (e.g. 1023, 3FFh, 0x3FF – you can use the form whichever you prefer). For hexadecimal, suffix “h” or prefix “0x” must be used, as is shown in example.

7.10.6. 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 value: 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).

7.10.7. Invalid block filling value

Specifies the value used for filling invalid blocks.

Default value: 00h (0000h for x16 devices)

Note: Both, decimal and hexadecimal forms are accepted for this option (e.g. 1023, 3FFh, 0x3FF – you can use the form whichever you prefer). For hexadecimal, suffix “h” or prefix “0x” must be used, as is shown in example.

7.11. Reserved Block Area Options

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

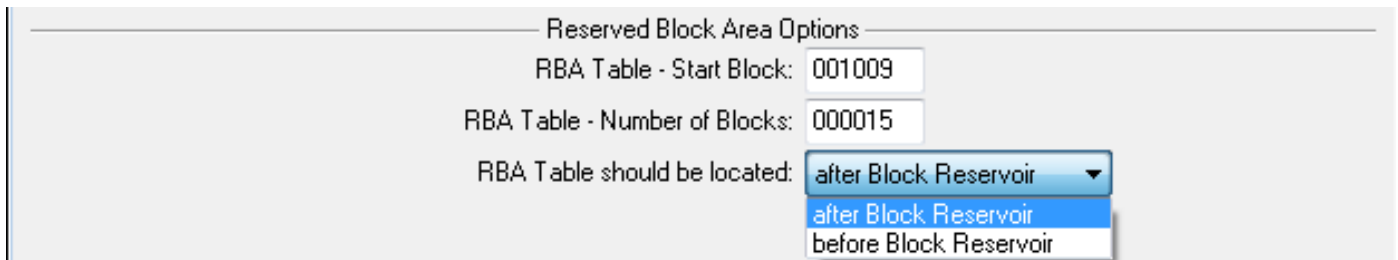


Figure 28: Reserved blocks area options.

7.11.1. RBA Table – Start Block

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

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

Note: Both, decimal and hexadecimal forms are accepted for this option (e.g. 1023, 3FFh, 0x3FF – you can use the form whichever you prefer). For hexadecimal, suffix “h” or prefix “0x” must be used, as is shown in example.

7.11.2. RBA Table – Number of Blocks

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

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

Note: Both, decimal and hexadecimal forms are accepted for this option (e.g. 1023, 3FFh, 0x3FF – you can use the form whichever you prefer). For hexadecimal, suffix “h” or prefix “0x” must be used, as is shown in example.

7.11.3. RBA Table should be located

Specifies the areas layout in device (see Figure 10):

- **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.

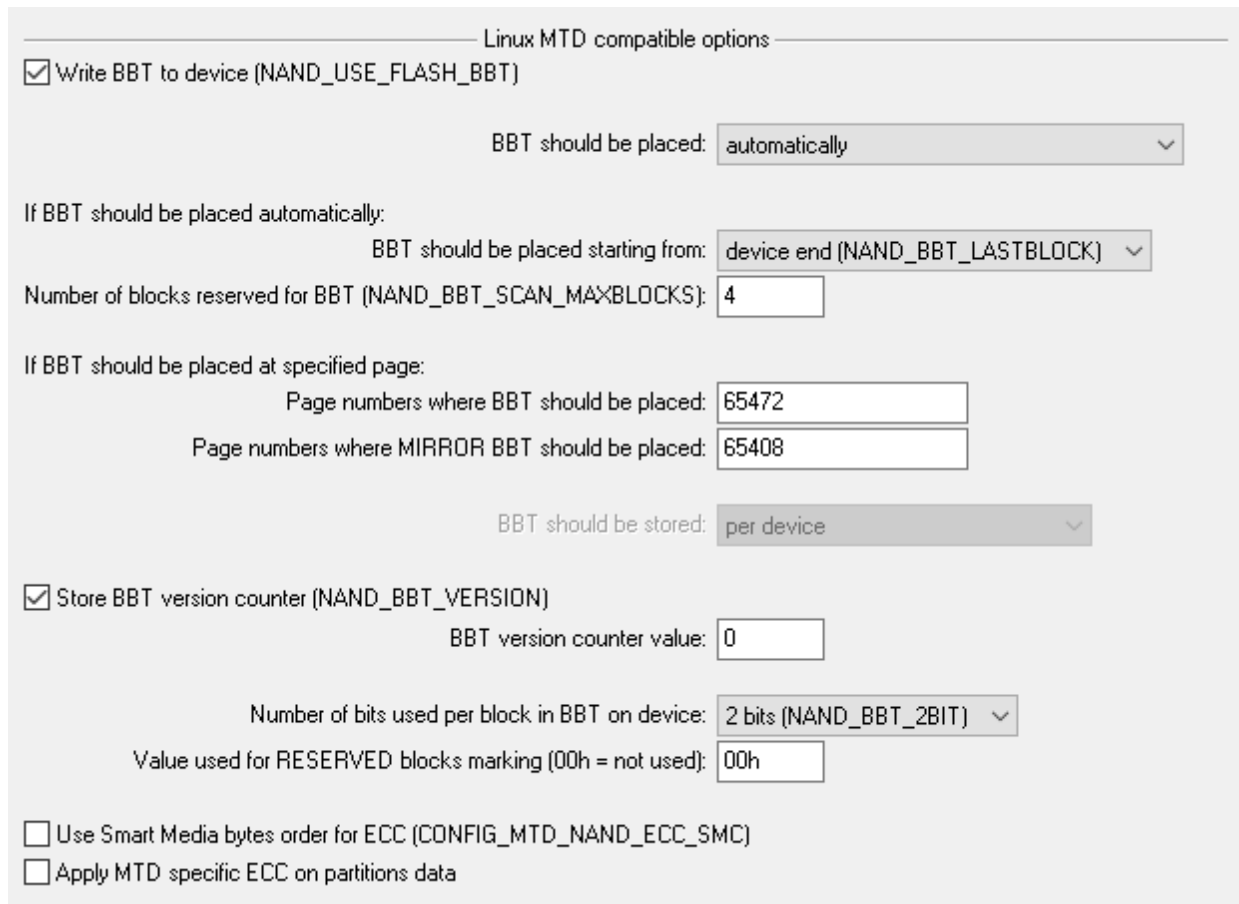
Default value: after Block Reservoir

7.12. Linux MTD Compatible Options

Notes: Related invalid blocks management technique is not supported for devices that require ECC with more than 1 bit error correction in 256 byte frame (only Hamming ECC algorithm is supported).

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.

See also chapter **Linux MTD compatible**.



Linux MTD compatible options

☒ Write BBT to device (NAND_USE_FLASH_BBT)

BBT should be placed: automatically

If BBT should be placed automatically:

BBT should be placed starting from: device end (NAND_BBT_LASTBLOCK)

Number of blocks reserved for BBT (NAND_BBT_SCAN_MAXBLOCKS): 4

If BBT should be placed at specified page:

Page numbers where BBT should be placed: 65472

Page numbers where MIRROR BBT should be placed: 65408

BBT should be stored: per device

☒ Store BBT version counter (NAND_BBT_VERSION)

BBT version counter value: 0

Number of bits used per block in BBT on device: 2 bits (NAND_BBT_2BIT)

Value used for RESERVED blocks marking (00h = not used): 00h

☐ Use Smart Media bytes order for ECC (CONFIG_MTD_NAND_ECC_SMC)

☐ Apply MTD specific ECC on partitions data

Figure 29: Linux MTD compatible options.

7.12.1. Write BBT to device

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

Default value: 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.

7.12.2. 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 value: Automatically.

7.12.3. 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 value: Device end.

7.12.4. Number of blocks reserved for BBT

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

Default value: 4 (MTD driver default value)

Note: Both, decimal and hexadecimal forms are accepted for this option (e.g. 1023, 3FFh, 0x3FF – you can use the form whichever you prefer). For hexadecimal, suffix “h” or prefix “0x” must be used, as is shown in example.

Example:

For device with 1 024 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.

7.12.5. 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(s) here, counting from page 0.

Default value: first page of last target block

Example:

List of pages example for multi target devices:

page_in_target0, page_in_target1, page_in_target2, ..., page_in_target(N-1)

Let have a device with 4 targets (CE# pins), with 8 192 blocks per target and 64 pages in a block. We want to put BBT in first page of block #8191, i.e. page #524224. The pages list will be as follows:

524224, 524224, 524224, 524224

7.12.6. 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(s) here, counting from page 0.

Default value: first page in last but one target block

7.12.7. 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 value: for single chip devices: per device (irrelevant); for multi chip devices: per chip

7.12.8. Store BBT version counter

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

Default value: Enable.

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.

7.12.9. BBT version counter Value

Specifies initial BBT version counter value.

Default value: 0

Note: Both, decimal and hexadecimal forms are accepted for this option (e.g. 1023, 3FFh, 0x3FF – you can use the form whichever you prefer). For hexadecimal, suffix “h” or prefix “0x” must be used, as is shown in example.

7.12.10. 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, 11110000b = worn-out block, 11111111b = good block.

Default value: 2 bits

7.12.11. 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 value: 00h (reserved block = good block)

7.12.12. Use Smart Media bytes order for ECC

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

Default value: Disabled.

7.12.13. 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 (2×256 byte frame) variant 1 and 2** for more detailed information.

Default value: Disabled.

7.13. Special Device Features

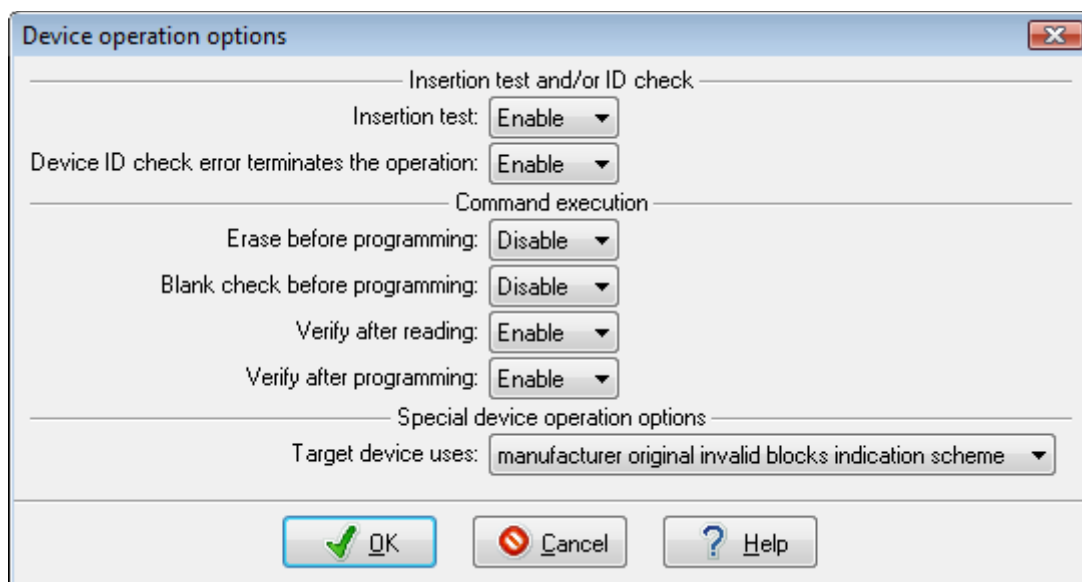
Figure 30: Special device features menu example.

Special device features is a common container for plentiful device dependent features. Particular feature is available in the menu only if following conditions are both met:

- device supports the feature;
- corresponding support is implemented for your device.

Note: Many of these features are only supported by a few devices, features with the same name may be used by different manufacturers for different purposes, and also the same function from the same manufacturer may have different parameters on different devices. Therefore, the description of particular features is out of the scope of this application note. Please, always follow the information from your device datasheet carefully.

8. Device operation Options Window



8.1. Insertion Test And / Or ID Check

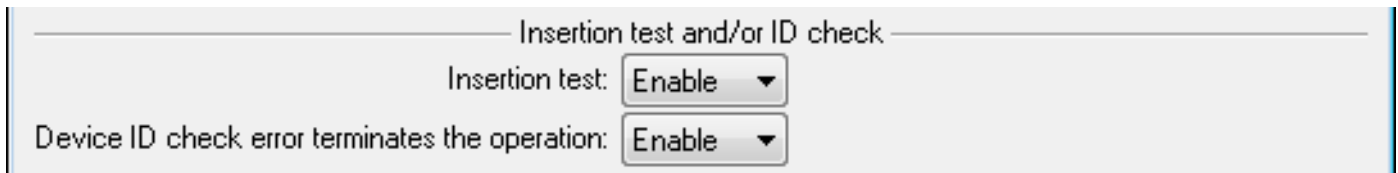


Figure 31: Insertion test and ID check options.

8.1.1. Insertion test

Enables / disables signal continuity check.

Default value: Enable.

Note: Please, be aware of static character 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, adapter's ZIF socket lifetime specified in adapter manual.

8.1.1.1. Basic test of IC functionality

This is extended test of signal integrity performed automatically as an integral part of insertion test. Programmer successively selects each device chip, activates first page reading and tries to read few bytes. On first read, data bus is held in pull-up, while on second in pull-down.

Possible errors are reported as follows:

Basic test of IC functionality – error!

Possible reason: bad or unreliable contacts (CE=xx/EC=yy).

where xx stands for chip number (a number of pin CE#);

and yy stands for following errors:

- 01 – respective R/B# pin did not fall to L after READ command entering;
- 02 – respective R/B# pin did not come back to H after READ command entering within expected time-out;
- 03/Data=PU/PD – read with data bud in pull-up differs from read with data bus in pull-down; respective data are shown.


8.1.2. Device ID check error terminates the operation

Enables / disable operation halting on ID check error.

Default value: 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.

8.2. Command Execution



Command execution	
Erase before programming:	Disable ▼
Blank check before programming:	Disable ▼
Verify after reading:	Enable ▼
Verify after programming:	Enable ▼

Figure 32: Command execution options.

8.2.1. Erase before programming

Enables / disables device erasing before programming. It is possible to erase all blocks, or all valid blocks, depending on invalid block management technique set, before the programming begins. The entire device is always erased, settings specified in **User Area options** section are ignored.

Default value: Disable.

8.2.2. Blank check before programming

Enables / disables device erase check before programming. It is possible to check the erasure of all blocks or all valid blocks, depending on invalid block management technique set, before the programming begins. The entire device is always checked, settings specified in **User Area options** section are ignored.

Default value: Disable.

Note: Invalid block has, at least, one non-FFh byte. Therefore blank checking of invalid block will always fail to blank check error. We do not recommend to use **Blank check before programming** together with **Treat All Blocks** invalid blocks management technique.

Note: NAND flash devices contain internal controller that manages all device 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**. See also chapter **Two factors that programmer relies on**.

8.2.3. Verify after reading

Enables / disables read data verification. During the read operation, data are stored in buffer. After successful read operation, all accessed blocks are read again and data are compared against previous read data in buffer.

Default value: Enable.

8.2.4. Verify after programming

Enables / disables verification of programmed data against data in buffer. After successful programming operation, all accessed blocks are read and data are compared against data in buffer.

Default value: Enable.

Note: Programmer verifies all pages in a block, including blank pages.

8.3. Special Device Operation Options

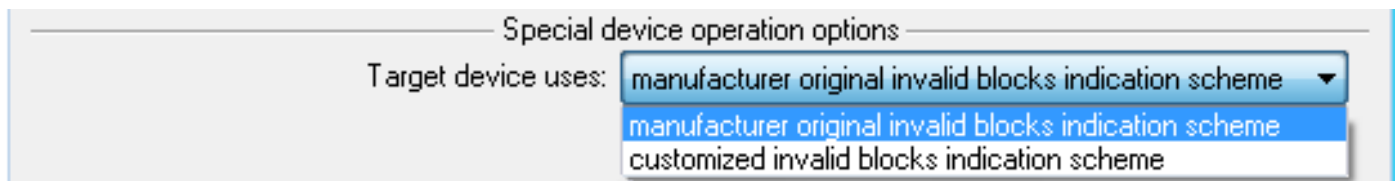


Figure 33: Special device operation options.

8.3.1. 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 Block Indication options (Extended version)**.

Default value: Manufacturer original invalid blocks indication scheme.

9. Special NAND Flash Commands

Note: Following commands can be accessed through menu **Device**.

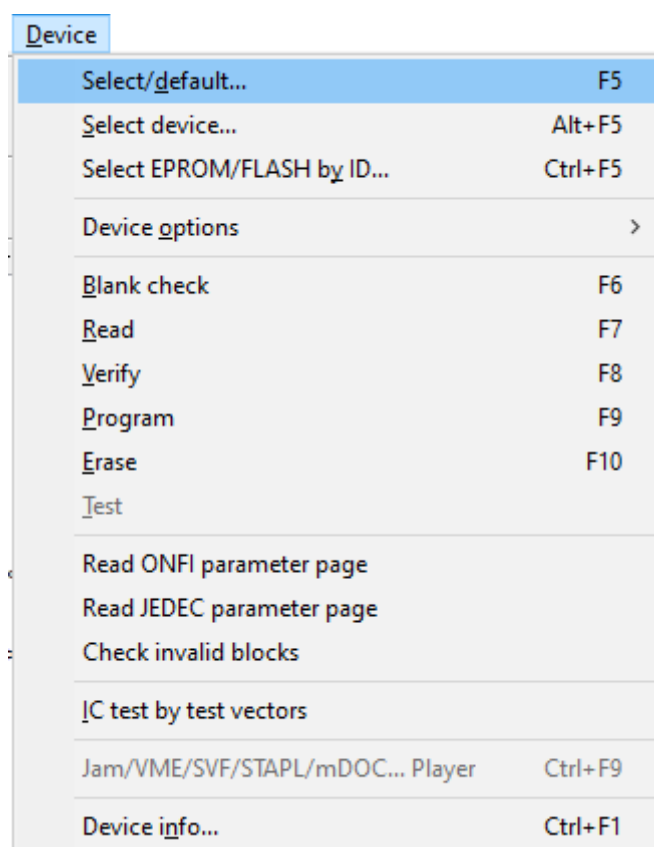


Figure 34: Menu device.

9.1. Read ONFI Parameter Page

Note: The feature availability is device dependent.

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
2018.Nov.19 13:03:39

ID read from device from address 0x00:
2C 44 44 4B A9 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: 0x003E
  ONFI version 1.0
  ONFI version 2.0
  ONFI version 2.1
  ONFI version 2.2
  ONFI version 2.3
Features supported: 0x01D8
  supports multi-plane program and erase operations
  supports odd to even page Copyback
  supports multi-plane read operations
  supports extended parameter page
  supports program page register clear enhancement
Optional commands supported: 0x03FF
  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
  supports Change Read Column Enhanced
  supports Change Row Address
  supports Small Data Move
  supports Reset LUN
Extended parameter page length: 0x0003
Number of parameter pages: 0x1D

Manufacturer information block
Device manufacturer: 'MICRON'
Device model: 'MT29F32G08CBADWP'
JEDEC manufacturer ID: 0x2C
Date code: Y:0 W:0

Memory organization block
Number of data bytes per page: 8192
Number of spare bytes per page: 744
Obsolete - Number of data bytes per partial page: 0
Obsolete - Number of spare bytes per partial page: 0
Number of pages per block: 256
Number of blocks per logical unit (LUN): 2128
Number of logical units (LUNs): 1
Number of address cycles: 0x23
  Row address cycles: 3
  Column address cycles: 2
Number of bits per cell: 2
Bad blocks maximum per LUN: 74
Block endurance: 3 x 10^3
Guaranteed valid blocks at beginning of target: 1
Block endurance for guaranteed valid blocks: 0
Number of programs per page: 1
Obsolete - Partial programming attributes: 0
Number of bits ECC correctability: 255
Number of plane address bits: 1
Multi-plane operation attributes: 0x1E
  no block address restrictions
  program cache supported
  Address restrictions for cache operations
  read cache supported
EZ NAND support: 0x00
  none

Electrical parameters block
I/O pin capacitance, maximum: 7
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
Obsolete - Asynchronous program cache timing mode support: 0
tPROG Maximum page program time (us): 2500
tBERS Maximum block erase time (us): 12000
tR Maximum page read time (us): 90
tCCS Minimum change column setup time (ns): 200
Source synchronous timing mode support: 0x0000
none
Source synchronous features: 0x00
none
CLK input pin capacitance, typical: 0
I/O pin capacitance, typical: 53
Input pin capacitance, typical: 42
Input pin capacitance, maximum: 10
Driver strength support: 0x07
supports driver strength settings
supports 25 Ohm drive strength
supports 18 Ohm drive strength
tR Maximum multi-plane page read time (us): 90
tADL Program page register clear enhancement tADL value (ns): 70
tR Typical page read time for EZ NAND (us): 0

Vendor block
Vendor specific Revision number: 0x0001
Vendor specific (in Hex form):
01 00 00 00 04 10 01 81
04 02 02 01 1E 90 08 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 02

Integrity CRC: 0x1D55

Parameter page dump (in Hex form):
4F 4E 46 49 3E 00 D8 01 FF 03 00 00 03 00 1D 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 33 32 47 30 38 43 42 41 44 41 57 50 20 20 20
2C 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 20 00 00 E8 02 00 00 00 00 00 00 00 01 00
50 08 00 00 01 23 02 4A 00 03 03 01 00 00 01 00
FF 01 1E 00 00 00 00 00 00 00 00 00 00 00 00
07 3F 00 00 00 C4 09 E0 2E 5A 00 C8 00 00 00 00
00 00 35 00 2A 00 0A 07 5A 00 46 00 00 00 00 00
00 00 00 00 01 00 01 00 00 00 04 10 01 81 04 02
02 01 1E 90 08 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 02 55 1D
```

9.2. Read JEDEC Parameter Page

Note: The feature availability is device dependent.

JEDEC standard (see <https://www.jedec.org> for more information) is very similar to ONFI standards. It also specifies 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 JEDEC 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.

JEDEC parameter page report example:

```
ELNEC JESD Decoder
2018.Nov.19 13:09:21

ID read from device from address 0x00:
2C 44 44 4B A9 00 00 00
ID read from device from address 0x40:
4A 45 44 45 43 01 00 00

Revision information and features block
Parameter page signature: 'JESD'
Revision number: 0x0003
Reserved (0)
supports vendor specific parameter page
Features supported: 0x0000
none
Optional commands supported: 0x000000
none
Secondary commands supported: 0x0000
Number of Parameter Pages: 0

Manufacturer information block
Device manufacturer: 'MICRON'
Device model: 'MT29F32G08CBADAWP'
JEDEC manufacturer ID: 0x2C0000000000

Memory organization block
Number of data bytes per page: 8192
Number of spare bytes per page: 744
Number of data bytes per partial page (MICRON specific): 1024
Number of spare bytes per partial page (MICRON specific): 93
Number of pages per block: 256
Number of blocks per logical unit (LUN): 2128
Number of logical units (LUNs): 1
Number of address cycles: 0x23
Row address cycles: 3
Column address cycles: 2
Number of bits per cell: 2
Number of programs per page: 1
Multi-plane addressing: 0x01
Number of plane address bits: 1
Multi-plane operation attributes: 0x07
No multi-plane block address restrictions
program cache supported
read cache supported

Electrical parameters block
Asynchronous SDR speed grade: 0x003F
supports 100 ns speed grade (10 MHz)
supports 50 ns speed grade (20 MHz)
supports 35 ns speed grade (~28 MHz)
supports 30 ns speed grade (~33 MHz)
supports 25 ns speed grade (40 MHz)
supports 20 ns speed grade (50 MHz)
Toggle Mode DDR and NV-DDR2 speed grade: 0x0000
none
```

```
Synchronous DDR speed grade: 0x0000
none
Asynchronous SDR features: 0x00
Toggle-mode DDR features: 0x00
Synchronous DDR features: 0x0000
none
tPROG Maximum page program time (us): 2500
tBERS Maximum block erase time (us): 12000
tR Maximum page read time (us): 90
tR Maximum multi-plane page read time (us): 90
tCCS Minimum change column setup time (ns): 200
I/O pin capacitance, typical: 53
Input pin capacitance, typical: 42
CK pin capacitance, typical: 0
Driver strength support: 0x07
  supports 35ohm/50ohm driver strength
  supports 25 Ohm drive strength
  supports 18 Ohm drive strength
tADL Program page register clear enhancement tADL value (ns): 0
```

```
ECC and endurance block
Guaranteed valid blocks at beginning of target: 1
Block endurance for guaranteed valid blocks: 0
ECC and endurance information block 0
  Number of bits ECC correctability: 40
  Codeword size: 10
  Bad blocks maximum per LUN: 74
  Block endurance: 771
  Reserved (0): 0
ECC and endurance information block 1
  Number of bits ECC correctability: 0
  Codeword size: 0
  Bad blocks maximum per LUN: 0
  Block endurance: 0
  Reserved (0): 0
ECC and endurance information block 2
  Number of bits ECC correctability: 0
  Codeword size: 0
  Bad blocks maximum per LUN: 0
  Block endurance: 0
  Reserved (0): 0
ECC and endurance information block 3
  Number of bits ECC correctability: 0
  Codeword size: 0
  Bad blocks maximum per LUN: 0
  Block endurance: 0
  Reserved (0): 0
```

```
Vendor specific block
Vendor specific Revision number: 0x0001
Vendor specific (in Hex form):
08 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 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
```

```
Integrity CRC: 0xD0AA
```

```
Parameter page dump (in Hex form):
4A 45 53 44 03 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
4D 49 43 52 4F 4E 20 20 20 20 20 20 4D 54 32 39
46 33 32 47 30 38 43 42 41 44 41 57 50 20 20 20
2C 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 20 00 00 E8 02 00 04 00 00 5D 00 00 01 00 00
50 08 00 00 01 23 02 01 01 07 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
3F 00 00 00 00 00 00 00 00 C4 09 E0 2E 5A 00 5A
00 C8 00 35 00 2A 00 00 00 07 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
01 00 00 28 0A 4A 00 03 03 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

```

9.3. 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.

Note: The feature accepts settings in section **Invalid Block Indication options (Extended version)** if enabled also in **Target device uses** option.

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)

```

10. Using Multi-Project Feature For NAND Flash

Multi-Project feature was intended for multi-chip devices, like e.g. NAND flash and NOR flash packed in single package. The feature allows to create individual project files for each particular memory in such multi-chip device, pack them together into single Multi-project file and then program all chips on single button click. If we consider several partitions in single NAND flash instead of several independent chips in one package, we will get new possibilities how to solve complex requirements for programming NAND flash partitions, even on programmers that do not support partitioning invalid blocks management techniques. For detailed information of Multi-Project feature, consult our application note freely available from our website https://www.elnec.com/sw/an_elnec_multi_prj.pdf.

10.1. Working With Multi-Project Wizard

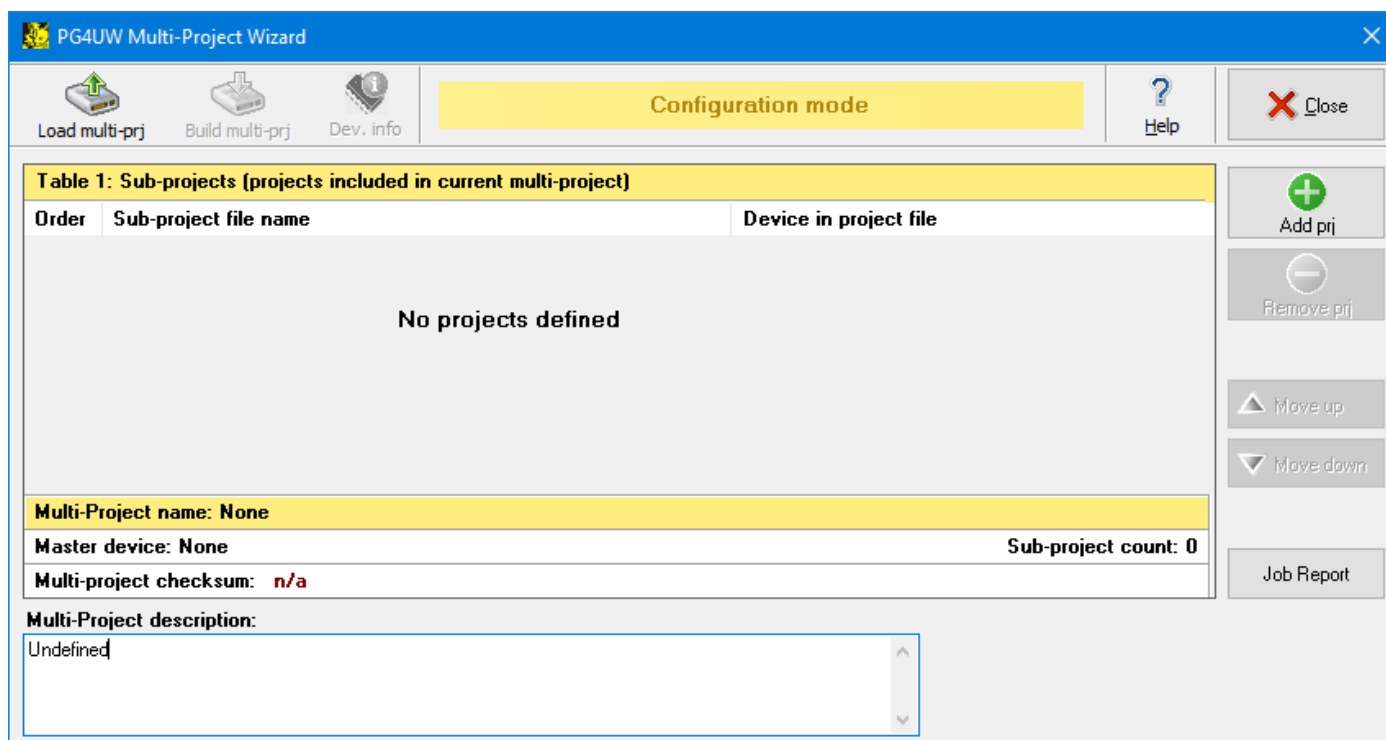


Figure 35: Empty Pg4uw Multi-Project Wizard window.

Multi-Project Wizard is a dedicated tool for working with Multi-Project files. The Multi-Project file can be created only using Multi-Project Wizard (see Figure 35). The Wizard can be accessed from Pg4uw menu **Options / Multi-Project Wizard**, or using a key short-cut **<Ctrl+M>**.

The Wizard allows following main features:

- building new Multi-Project file;
- loading existing Multi-Project file;
- running the multi-chip device operation, according to actual Multi-Project file.

The Multi-Project Wizard window contains following controls:



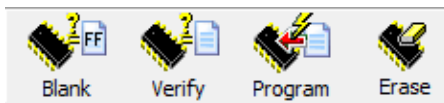
Button **Load Multi-Project file**
Loads existing Multi-Project file.



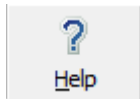
Button **Build Multi-Project file**
Builds new Multi-Project file from selected Project files.



Button **Device info**
Shows short device info (if available).



Buttons for device operations – **Blank, Verify, Program, Erase**
These buttons are used to run the selected multi-chip device operation according to Project files. Read operation is not supported in Multi-Project mode.



Button **Help**
Shows Help window with brief assistance on using Multi-Project Wizard.



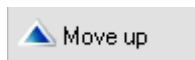
Button **Close**
Terminates the Multi-Project Wizard. After closing, the “unselected” device is automatically selected in Pg4uw software.



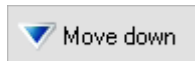
Button **Add project**
Adds existing Project file into selected Project files list, making it ready for later Multi-Project file building.



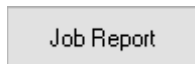
Button **Remove project**
Removes the Project file from selected Project files list.

**Button Move up**

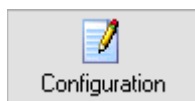
Moves the Project file one position up in selected Project files list. When running the Multi-Project file, individual Project files are processed in order according to selected Project files list.

**Button Move down**

Moves the Project file one position down in selected Project files list.

**Button Job Report**

Creates job report. For more information consult your programmer manual or **File / Make Job Report** paragraph in Pg4uw help.

**Button Configuration**

The button is displayed only if Multi-Project file is loaded and ready for operation. It switches the Wizard back to Multi-Project file build state.

10.1.1. Defining the Project files for individual NAND partitions

Firstly you need to create respective project files for each one desired partition. You can employ any options available for NAND flash (in appropriate manner). You can mix partitions with various **Invalid Block Management** techniques, **Spare area usage** modes, you can even embed partitions into a partition...

Notes:

The projects are executed in an order as they are arranged at **Building the Multi-Project file**. Please, keep this in mind when using features like **Erase before programming** and / or **Blank check before programming**, which are always performed on entire device. Do not allow these operations in a project other than the first on the list.

Similarly, there may be various protections and device / block locks available for target device. Once enabled, they may prevent from further programming. Do not allow such options in a project other than the last on the list.

If Spare Area Usage setting is set to User Data for certain partition, a plenty of invalid blocks may be found by programmer when processing all subsequent partitions (it depends on programmed data content). The only effect will be a long listing in control software log window. This will not affect the quality of the programming process for subsequent partitions, since invalid blocks before start block are ignored.

- Firstly, display the Select device window and list for desired NAND flash device. Select the device.
- Display **Access method window**, key short-cut <Alt+S>. Set appropriate features according to your needs. Follow the information in earlier chapters of this application note for particular options.
- Display **Device operation options window**, key short-cut <Alt+O>. Select the proper settings.
- Load the data file(s) into buffer (see chapter **Loading data into pg4uw control software buffer**).
- Save (and test) the Project file for the partition.

This way prepare the Project files for all your partitions. Once all Project files are saved you can build the Multi-Project file.

10.1.2. Building the Multi-Project file

Having Project files available for all partitions, you can build the Multi-Project file.

- Launch Multi-Project Wizard (see Figure 35) using **<Ctrl+M>** shortcut or **Options / Multi-Project Wizard** menu.
- Add required Project files using **Add project** button.
- If necessary, rearrange the Project files execution order using buttons **Move up** and **Move down**.
- An accidentally added Project file(s) can be discarded using **Remove project** button.
- After completion of Project files selection, use **Build Multi-Project file** button to start the building process.

Once having Multi-Project file available, you can simply load it into Multi-Project Wizard using **Load Multi-Project file** button.

10.1.3. Running the multi-chip device operation

After building or loading Multi-Project file into Multi-Project Wizard, you can run desired operation by clicking on respective **Blank**, **Verify**, **Program**, **Erase** button that are now available in Wizard window (see Figure 36). This will start the sequence of Project files loading and operations execution.

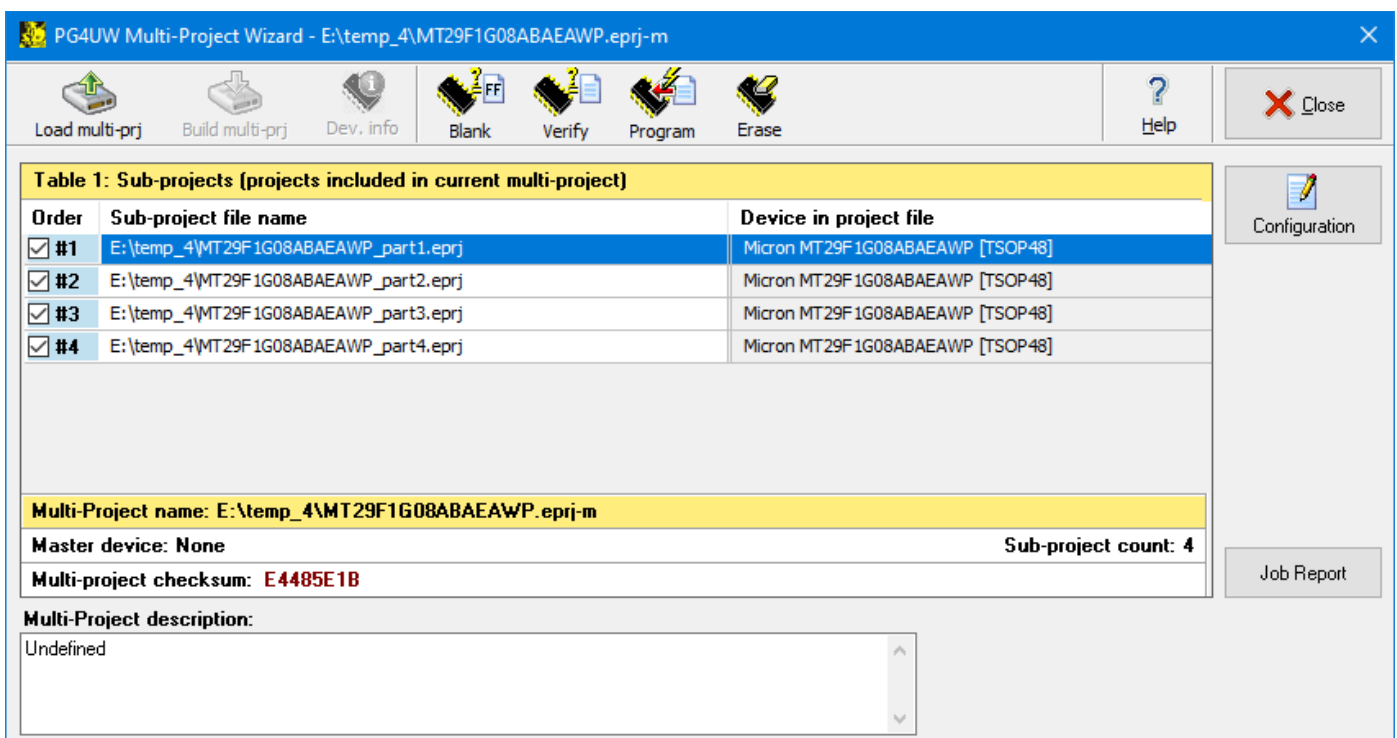


Figure 36: Multi-Project Wizard window with Multi-Project file loaded.

11. Customized NAND Flash Support

Sometimes it is not possible to achieve the required parameters of programming / processing the device invalid blocks / loading (decoding) input data using freely available functions. In such case it is necessary to proceed to a customized implementation. At Elnec, we have extensive experience in supporting custom algorithms.

In order for this support to be successful and its implementation as fast as possible, it is necessary to specify a few things, which are addressed in following chapters. We called these things “schemes”.

The requirements can be so diverse that we cannot put together a simple form where it would be enough to click a few boxes. Use your own words when compiling the specification, attach the source code if necessary. It is important that all seven schemes are explained (or marked as unused).

11.1. Partitioning Scheme

Typically, different types of data are programmed into a NAND flash memory device. It may be necessary to process these different types of data using different approaches. Each type of data may also need to start at a specific location. See chapter **Multiple partitions with Skip IB** for detailed information about partitions.

The following must be specified:

- the number of partitions;
- each partition start block and last block (also the number of used blocks if it is known);
- type of invalid blocks management technique for each partition;
- first block in partition must be good, max. allowed number of invalid blocks and other partition specific requirements (see chapter **Comma separated values format (*.csv)** for more details);
- partition table definition file format;
- any other partition related technicalities not mentioned here.

11.2. Invalid Blocks Management Scheme

It is necessary to specify in detail how the programmer should proceed if he encounters an invalid block during the operation. See chapter **Invalid Block Management** for more details about standard techniques available.

The following must be specified:

- how to recognize an existing bad block;
- how to mark a new bad block that appears during an operation;
- how to select a replacement block;

- how and where to record the replacement (exact format of the redirection table, if used);
- any other invalid blocks related technicalities not mentioned here.

11.3. Dynamic Meta-data Scheme

Dynamic meta-data are all the data that need to be prepared on-the-fly during the operation, because their essence does not allow to prepare them into buffer in advance. Mainly those are various serial numbers, MAC addresses, file system headers, bad block tables,...

Note: ECC and other error protection techniques are part of a separate chapter, see **Error control and correction scheme**.

The following must be specified:

- what types of dynamic meta-data are used;
- exact location and size of each dynamic meta-data unit;
- exact procedure for obtaining the value of each dynamic meta-data;
- any other dynamic meta-data related technicalities not mentioned here.

11.4. Page Arrangement Scheme

Different application processors work with NAND flash page in different ways – some respect the natural technological page segmentation into data and spare areas, others work with the page as a whole. Some processors add their own meta-data to the page. Sometimes it is necessary to apply some way of BI byte repositioning. ECC checksums are placed at different offsets, sometimes concentrated in one place, other times scattered throughout the page.

Sometimes the situation is further complicated by requests from higher layers of the file system for various headers in dedicated pages in a block.

If the programmer should do anything more than simply take a page from the buffer and write it to the device, all of the above must be clearly specified.

The following must be specified:

- all used types of pages (block header, data,...);
- all types of data on each page type (meta-data, data, checksum, any counter, padding, unused,...);
- offsets and sizes for all corresponding data entries on a page;
- BI byte (block validity indication byte) position, if is changed from datasheet default;
- any other page arrangement related technicalities not mentioned here.

11.5. Error Control And Correction Scheme

Usually one of the three ECC algorithms is used – Hamming, BCH, RS. Each of them is characterized by a number of parameters that the programmer must know if it should calculate the checksums on its own. ECC is

sometimes extended by adding some checksum of CRC type. The best way how to communicate the ECC / CRC algorithm is to provide us with the source code.

The following must be specified:

- the type of ECC and / or CRC algorithm in use;
- all parameters of that algorithm(s) – bit recovery capability, generator polynomial, frame size,...;
- checksums positioning on a page;
- any other error protection related technicalities not mentioned here.

11.6. Unused Blocks Formatting Scheme

Sometimes it is necessary to format unused (empty) blocks in a partition in some way. Such requests are usually introduced by the file system (JFFS2, UBI / UBIFS).

The following must be specified:

- what the required formatting should look like (what data to which positions in the block should be written);
- how to calculate or from where to draw the relevant formatting data;
- it is very helpful to indicate who is the requester of that formatting (file system type,...);
- any other unused blocks related technicalities not mentioned here.

11.7. Input Data File Scheme

Ideally, the input data file should contain a “mirror” of the contents of the memory. Because NAND flash memory capacities are often really huge (several gigabytes), such input files have become quite cumbersome. Therefore, various techniques are used to “pack” unused (empty) areas. If any such technique is used, the exact specification must be given.

The following must be specified:

- which file belongs to which partition (if more than one input file is used);
- which data the input file contains and which data need to be calculated;
- unpacking / decrypting algorithm (if used);
- any headers, versioning,... if should be taken into account;
- any other input file(s) related technicalities not mentioned here.

12. Frequently Asked Questions

12.1. Device / Buffer Conversions

You may get a partition table specified in such a way that the beginnings of the individual partitions are specified by the offset in device. Typically, you then need to convert this device offset to the offset in buffer (to where the corresponding input data file needs to be loaded) and to the block number (to compile a partition table definition file).

Example:

```
0000 0000h boot
0010 0000h kernel
1000 0000h kernel_bkp
```

Device page organization is of 2 048 data bytes and 128 spare bytes. There are 64 pages in a block.

12.1.1. Conversion of the device offset to the block number

From a firmware perspective, the spare area is outside the address space, so it is not included in the specified device offset. It is therefore necessary to work with the page size without the spare area – i.e. 2 048 bytes in our example.

kernel:

The easiest way is to convert the offset to a page number, and then to a block number:

0010 0000h = 1 048 576(dec) bytes

page number = 1 048 576 bytes / 2 048 bytes in page = 512

block number = 512 pages / 64 pages in block = **8**

kernel_bkp:

For those who like formulas:

block_number = **device_offset** / (**data_page_size** x **pages_in_block**)

1000 0000h = 268 435 456(dec) bytes

target block number = 268 435 456 / (2 048 x 64) = **2 048**

12.1.2. Conversion of the device offset to the buffer offset

From a firmware perspective, the spare area is outside the address space, so it is not included in the specified device offset. Depending on **Spare area usage** setting, buffer offset must or must not include the spare area size –

see relevant chapter according to the actual spare area usage mode to see if the programmer expects spare area data in the buffer. In partitioning mode, spare area data must always be included in the buffer, so we will consider them in the next.

It means: $\text{buffer_page_size} = \text{data size} + \text{spare size} = 2\,176 \text{ bytes}$

kernel:

The easiest way is to convert the device offset to a page number, and then to the buffer offset:

$0010\,0000\text{h} = 1\,048\,576(\text{dec}) \text{ bytes}$

$\text{page number} = 1\,048\,576 \text{ bytes} / 2\,048 \text{ bytes in page} = 512$

buffer offset = $512 \text{ pages} \times 2\,176 \text{ bytes in page} = 1\,114\,112(\text{dec}) \text{ bytes} = \mathbf{0011\,0000\text{h}}$

kernel_bkp:

For those who like formulas:

buffer_offset = $(\text{device_offset} / \text{data_page_size}) \times \text{buffer_page_size}$

$1000\,0000\text{h} = 268\,435\,456(\text{dec}) \text{ bytes}$

buffer offset = $(268\,435\,456 / 2048) \times 2\,176 = 285\,212\,672(\text{dec}) = \mathbf{1100\,0000\text{h}}$

12.1.3. Conversion of the file size to the blocks count

Sometimes it is necessary to convert input data file size or partition size specified in bytes to the count of blocks in device. In principle, there is no difference between these two tasks.

Example:

`boot.bin size = 710 656 bytes`

`kernel.bin size = 71 234 560 bytes`

Device page organization is of 2 048 data bytes and 128 spare bytes. There are 64 pages in a block.

It is always necessary to clarify whether the file (or the size of the partition) contains also the spare area.

If it does, the page size = data size + spare size = 2 176 bytes.

If it does not, page size = data size = 2 048 bytes.

In partitioning mode, spare area data must always be included in the buffer, so we will consider them included in the next.

It should be remembered that the last incomplete block, if any, must also be included in the count of blocks.

boot.bin:

The easiest way is to convert the offset to a pages count, and then to a blocks count:

$\text{pages count} = 710\,656 \text{ bytes} / 2\,176 \text{ bytes in page} = 326\,588 \text{ pages}$

Last incomplete page must be included too, therefore pages count = 327.

$\text{blocks count} = 327 \text{ pages} / 64 \text{ pages in block} = 5.109 \text{ blocks}$

Last incomplete block must be included too, therefore **blocks count = 6**.

kernel.bin:

For those who like formulas:

$$\text{blocks_count} = \text{ceil}(\text{file_size} / (\text{page_size} \times \text{pages_in_block}))$$

$$\text{blocks count} = \text{ceil}(71\,234\,560 / (2\,176 \times 64)) = \text{ceil}(511.50735) = \mathbf{512 \text{ blocks}}$$

12.1.4. Conversion of the block number to buffer offset

This is a common task when loading separate input data image files for individual partitions.

Example:

Let have a partitions start_block specified as follows:

```
#0000 boot
#0008 kernel
#2048 kernel_bkp
```

Device page organization is of 2 048 data bytes and 128 spare bytes. There are 64 pages in a block.

Depending on **Spare area usage** setting, buffer offset must or must not include the spare area size – see the relevant chapter according to the actual spare area usage mode to see if the programmer expects spare area data in the buffer.

If it does, the page size = data size + spare size = 2 176 bytes.

If it does not, page size = data size = 2 048 bytes.

In partitioning mode, spare area data must always be included in the buffer, so we will consider them included in the next.

Required offset can be obtained as a simple product according to the following formula:

$$\text{buffer_offset} = \text{block_number} \times \text{pages_in_block} \times \text{page_size}$$

kernel:

$$\text{buffer offset} = 8 \times 64 \times 2\,176 = 1\,114\,112(\text{dec}) = \mathbf{0011\,0000h}$$

kernel_bkp:

$$\text{buffer offset} = 2\,048 \times 64 \times 2\,176 = 285\,212\,672(\text{dec}) = \mathbf{1100\,0000h}$$

12.2. Copying NAND Flash Memory

Whether you are trying to repair a broken device at a workshop or you are about to mass copy a master device as you are used to do with a NOR flash, please remember that NAND flash memory is a faulty storage medium. Reading an error-free data image from NAND flash memory without knowing and using all appropriate algorithms (invalid block management, ECC,...) is a matter of luck. Therefore, we do not consider copying NAND flash memory to be a suitable technique, we cannot recommend it at all for mass production. However, in the service, copying memory from a functional device is often the only option available, and we are often asked for advice on how to do this. Here it is:

Step 1: reading master-device:

- Select original device from list and then open Access Method <Alt+S> menu and change the following settings:
 - Invalid blocks management = **Treat All Blocks**;
 - Spare Area Usage = **User data**;
 - User Area – Number of Blocks = number of blocks in device.
- Leave the other settings in the default state. If available for the device, consider enabling special areas and / or features.
- Run Read operation. After Read is done, save the buffer content on disk (or save the project file if any special area and / or feature is enabled).

Now you have master device byte-by-byte image stored on your disk. Remember, please, that this image includes all invalid blocks existing in master device, as well as all single-bit errors. They both will propagate into target device, thus reducing its capacity (invalid blocks propagation) and data reliability (single-bit errors propagation).

Step 2: writing a copy:

- Select target device from list and load data image file from disk. If you saved the project in previous step, load the project file. In both cases, open Access Method <Alt+S> menu and change the following settings:
 - Invalid blocks management = **Skip IB**;
 - Spare Area Usage = **User data**;
 - User Area – Number of Blocks = see below.
- Run Program operation.

In ideal case, the number of all blocks available in device should be entered for User Area – Number of Blocks, similarly to the read step. But this will work only if target device for copying is free of invalid blocks.

So, you must reduce the number of programmed blocks by some value in practice. We recommend to determine some threshold for maximum invalid blocks count in device, and enter the value of total blocks in device reduced by this threshold. The number of blocks entered here cannot be less than the number of blocks really occupied by data in master device.

12.3. Problems With Too Many Invalid Blocks

12.3.1. How does your programmer identify invalid blocks?

When scanning for invalid blocks:

The programmer strictly follows the scheme of invalid blocks marking according to the manufacturer's specification. In practice this means, that it reads given byte / word on given pages in a block. Only if all these bytes / words are FFh / FFFFh, the block is considered valid. Otherwise, the programmer considers the block to be invalid.

When marking new invalid blocks:

The programmer writes 00h to all bytes on the pages specified by device manufacturer to mark invalid blocks.

The standard behaviour described above can be modified by changing the settings in section **Invalid Block Indication options (Extended version)**.

12.3.2. When working with device, programmer reports tens (hundreds, thousands) of invalid blocks. Is it normal?

This is not normal when working with **new device**. For SLC devices, the number of invalid blocks should not exceed 2% of the total. For MLC devices, manufacturers tend to specify a minimum number of valid blocks in device datasheet. An unusually high number of invalid blocks may indicate a problem with device, adapter, programmer, ...

This may be normal when working with an already **programmed device** (including solo-verification after programming). Many host-controllers work with a different page format than the classic segmentation to data and spare area. Block validity markers are then often overwritten with user data – if programmer finds a non-FFh value at expected location, it will consider the block to be invalid. In such cases, however, it is customary to use some other byte on the page to indicate the validity of the blocks. Try to find out which one is it and set **Invalid Block Indication options (Extended version)** section accordingly.

12.3.3. I need to do a test with a bad block so I want to make a sample with a bad blocks I make on my own.

The simplest way how to "create" invalid block is to fill buffer with all zeros (00h), set **Invalid Block Management = Skip IB** or **Treat All Blocks**, **Spare area usage = User data**, **User Area – Start Block** = the block you want to make invalid, **User Area – Number of Blocks** = 1.

If you want to use exact one-byte BI marks as are specified in datasheet, you need to prepare invalid block image in buffer and write it into chosen block. Erase buffer (fill buffer with all FFh) and rewrite specific locations to 00h. See your device datasheet for BI byte location – e.g. BI mark may be expected in first spare area locations on

first and/or second page, i.e. on block offsets 800h and 1040h. This invalid block image may be written into device using the same settings as in previous.

Programmer will take data from buffer start and program them starting from specified block in device, so using one block image you may produce as many invalid blocks as you want.

After finishing, you can restore the blocks using erase with Invalid Block Management = Treat All Blocks (see following question).

12.3.4. I have made a lot of invalid blocks in my device. Can I fix it somehow?

You can erase entire device including block validity information using the following setting:

Invalid Block management = **Do not use**

Note: There is a risk of losing information about the initial invalid blocks!

If initial invalid blocks have been disconnected from the line by the manufacturer, they will be recognized again. But if they were only marked in the spare area, the original information about the block invalidity will be lost.

12.4. Command Execution Dilemmas

12.4.1. Erase before programming

NAND flash devices are sold in a deleted state. Therefore, it is not necessary to activate Erase before programming for new devices (although, we have already solved the problems when there were some data in new devices – probably the seller repeatedly sold some returned devices). However, erasing the device will ensure that it will be truly blank and is incomparably faster than Blank check. Many customers therefore use Erase before programming instead of Blank check before programming. On the other hand, when erasing, the cells are much more stressed than when reading...

12.4.2. Blank check before programming

As in the previous question, it is not necessary to activate Blank check before programming for new devices. Blank check is not necessary even in connection with Erase before programming, as then the device is checked by the internal controller. In fact, if both Erase before programming and Blank check before programming are activated at the same time, the programmer will not perform a blank check (see also notes in chapter **Blank check before programming**).

12.4.3. Verify after programming

Verify after programming checks all programmed blocks, including the blank pages (skipped on programming due to a programming speed optimization). It does not check unused (padding) blocks between the data end and user area end (or partition end).

12.4.4. Pg4uw software recommends me to set more User Area blocks than I have set, saying it is more effective. Is it OK?

Yes, it can happen. The control software checks the size of the input data file and compares it with the size according to **User Area – Number of Blocks** setting. If it finds that the file is smaller or larger, it will suggest adjusting the setting according to the file size.

The file may also contain blank data for unused blocks at the end. In this case, the software will offer more blocks than necessary as a more optimal setting. Just do not accept such recommendation.

13. Appendix A: Errors In NAND Flash – The Background

NAND flash is more prone to errors than NOR flash due to its structure. The errors in NAND flash can be classified into two major categories: permanent (non-correctable) and temporary (correctable) errors. Memory wear is the permanent error. Temporary errors in NAND flash are Program Disturb, Read Disturb, Over-programming and Retention errors. A few words on each type of errors follow.

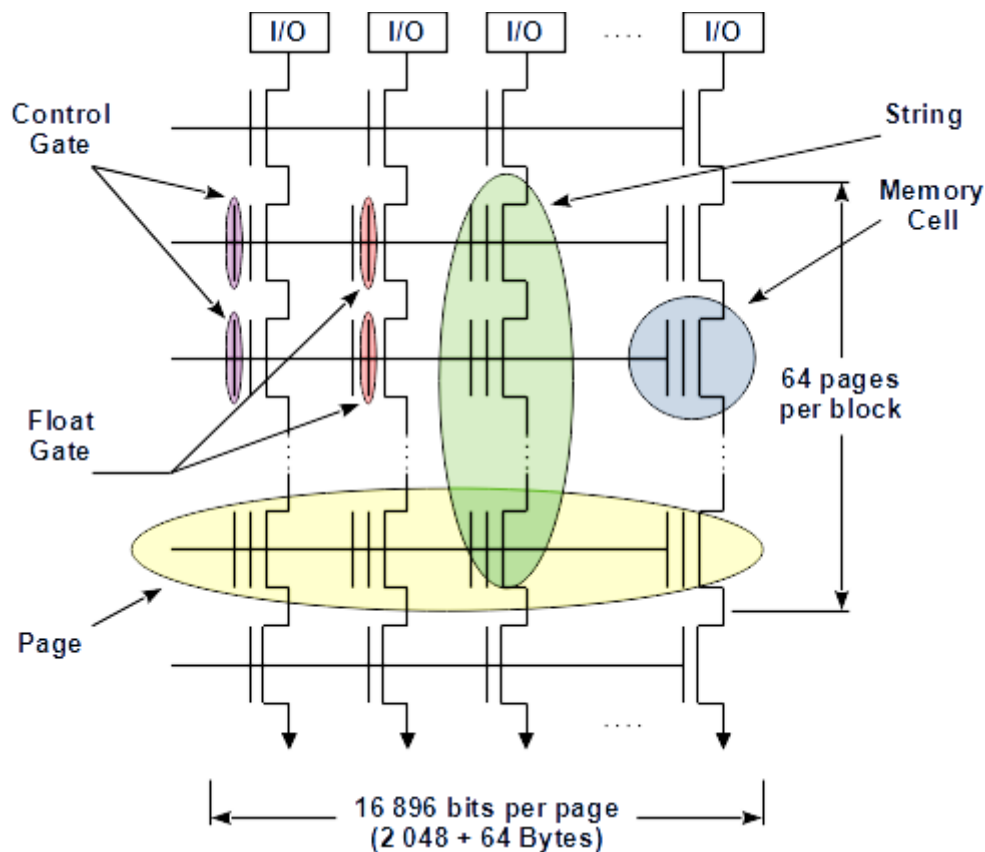


Figure 37: A NAND flash block architecture.

13.1. Memory Wear (Endurance) Errors

Memory wear is caused by program and erase operations. Every time a cell is programmed or erased, a few electrons get trapped in the dielectric. This causes a permanent shift in cell characteristics – not recovered by erase. Once the cell reaches a point where the controller can no longer reliably distinguish between programmed and erased states (observed as program or erase fail status), the cell is considered as bad or worn out. The block with the worn out cell is considered as a bad block and is not used any longer.

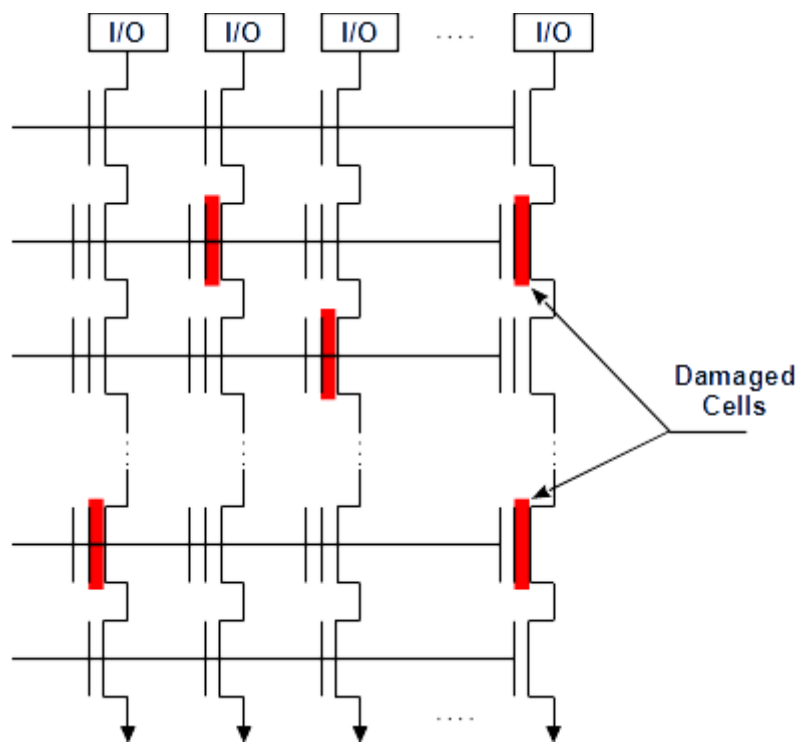


Figure 38: Wear-out (endurance) errors.

Endurance recommendations:

- always check pass / fail status for program and erase operations;
- if fail status after program operation, move all block data to another available block and mark failed block invalid;
- use ECC to recover from errors;
- write data equally to all good blocks (wear levelling);
- protect block management (meta) data in spare area using ECC.

13.2. Read Disturb Errors

To read a memory cell, the charge stored in the floating gate needs to be identified by measuring the threshold voltage of the cell. A reference voltage is applied at the gate terminal of the required cell and the voltage at which the cell starts conducting is measured to identify the threshold voltage. Since the memory cells are connected as strings in NAND flash, all other cells in the string need to be turned on prior to reading the required cell. A readout voltage higher than the maximum threshold voltage of the memory cells is applied to the gate terminal of all other cells in the string to turn them on or unselect the cells. In NAND flash, the gate terminals of multiple memory cells in different strings are connected together as a page. To unselect a cell in the string, the entire page need to be unselected, which means the readout voltage needs to be applied to the gate terminals of all the cells in a page.

Even though the readout voltage is much smaller compared to program or erase voltages, this can still cause a slight shift in the threshold voltage of the cells. These small shifts in threshold voltage accumulate over read cycles,

eventually changing the state of the cell. This unintentional shift in the threshold voltage of a cell due to read operation is known as read disturb error. Read disturb errors affect only the cells in unselected pages in the same block being read. As this is a temporary error, the error can be resolved by copying the block to another block, then erasing failed block to make it available again.

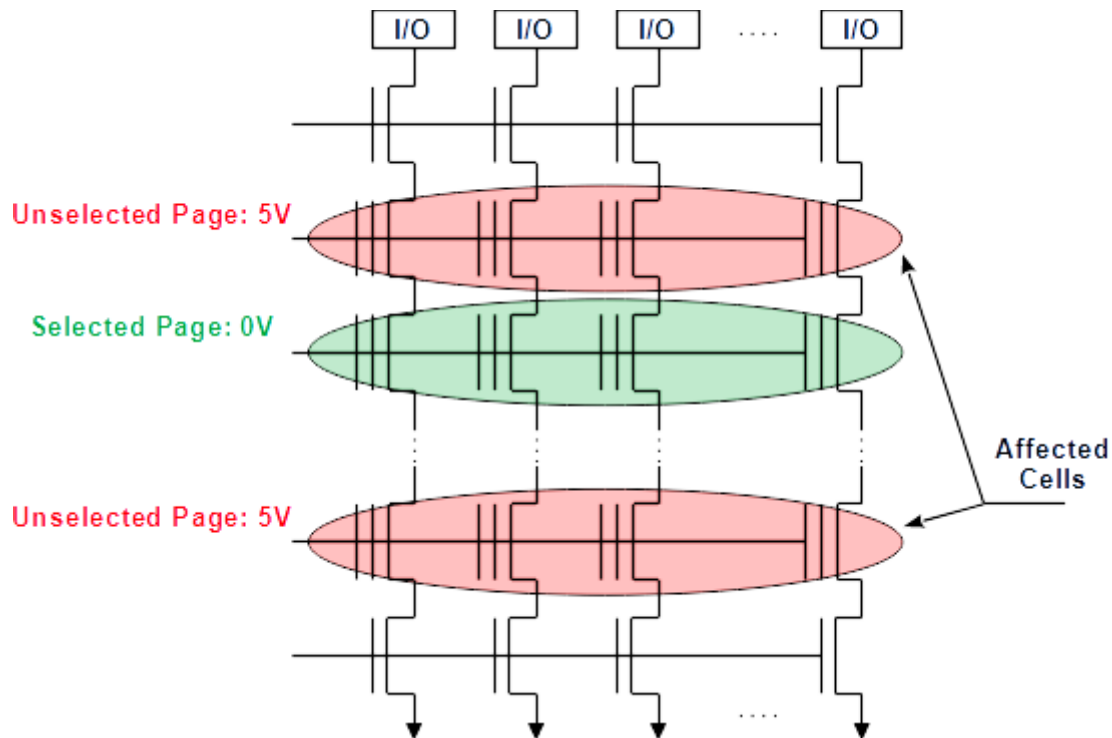


Figure 39: Read disturb errors.

Reducing read disturbs:

- rule-of-thumb for excessive reads in the block between two erase operation:
 - SLC – 1 000 000 read cycles;
 - MLC – 100 000 read cycles.
- if possible, read equally from pages within the block;
- if exceeding rule-of-thumb cycle count, move the block to another location and erase the original block;
- establish ECC threshold to move the data;
- erase resets the read disturb cycle count;
- use ECC to recover from read disturb errors.

13.3. Program Disturb Errors

A high voltage is applied across the memory cell for program and erase operations. Due to parasitic capacitive coupling, the adjacent cells also receive an elevated voltage stress that can alter the threshold level of these neighbouring cells. This unintentional shift in threshold level due to program or erase operations is known as a program disturb error.

A program disturb error affects cells in both selected and unselected pages, but only in the block being programmed. The parasitic capacitive coupling between adjacent cells increases with shrinking lithographic node, the same reason the Raw Bit Error Rate increases for smaller lithographic nodes. To recover from this error, the block needs to be erased after copying its contents to another block.

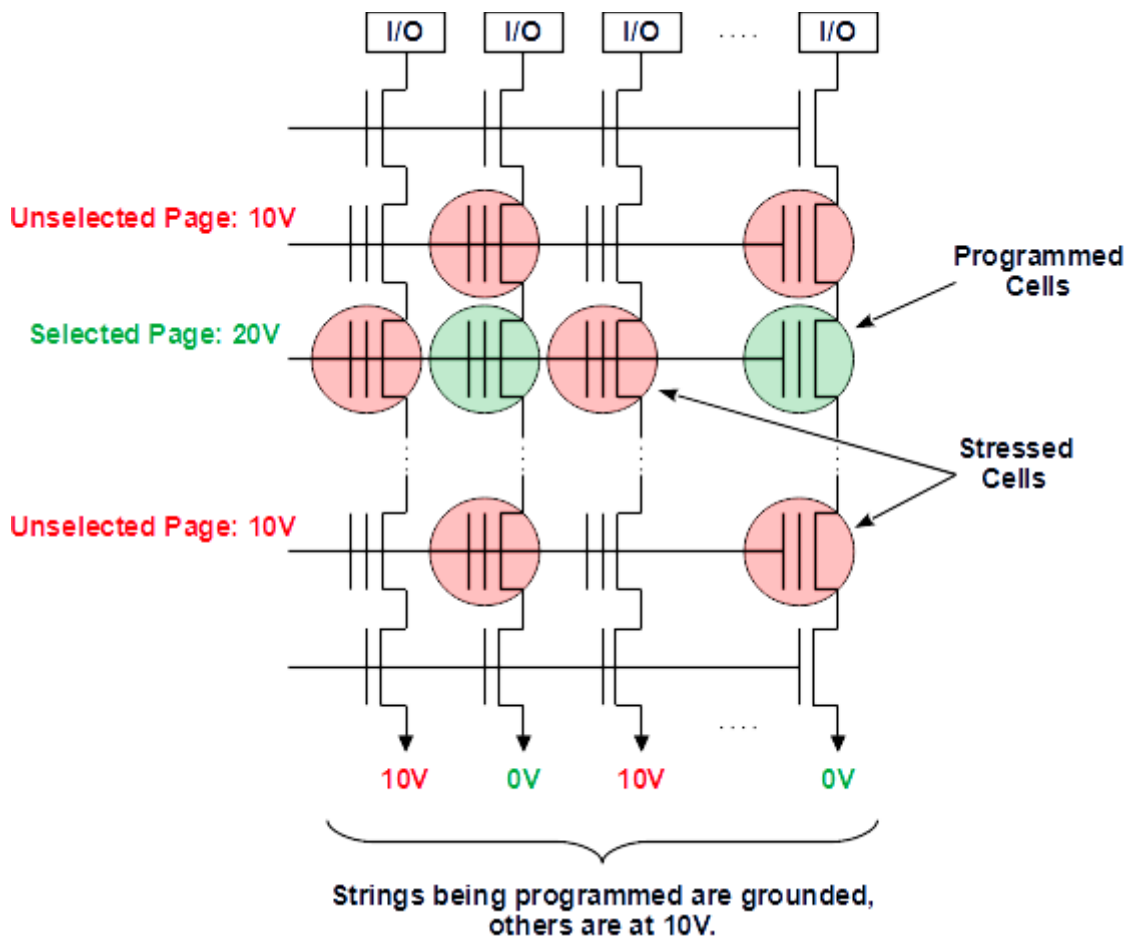


Figure 40: Program disturb errors.

Reducing program disturb:

- program pages in a block sequentially from page 0 to the top-most page in a block;
- minimize partial-page programming operations (SLC);
- it is mandatory to restrict page programming to a single operation (MLC);
- use ECC to recover from program disturb errors.

13.4. Over-programming Errors

Over-programming is another correctable error in NAND flash. While programming, the threshold voltage of some cells can go too high. As explained in the previous sections, memory cells need to be turned on or unselected for read and program operations. Cells with a very high threshold voltage will not turn on as expected when readout voltage is applied. This can result in incorrect read and program operations for other pages in the string. This is known as an over-programming error.

Over-programming errors are often caused by memory cells that hold a higher initial charge on the floating gate due to an improper erase operation. They also occur when memory cells are nearing a worn out state. To recover from this error, the block need to be erased after copying the contents to another block.

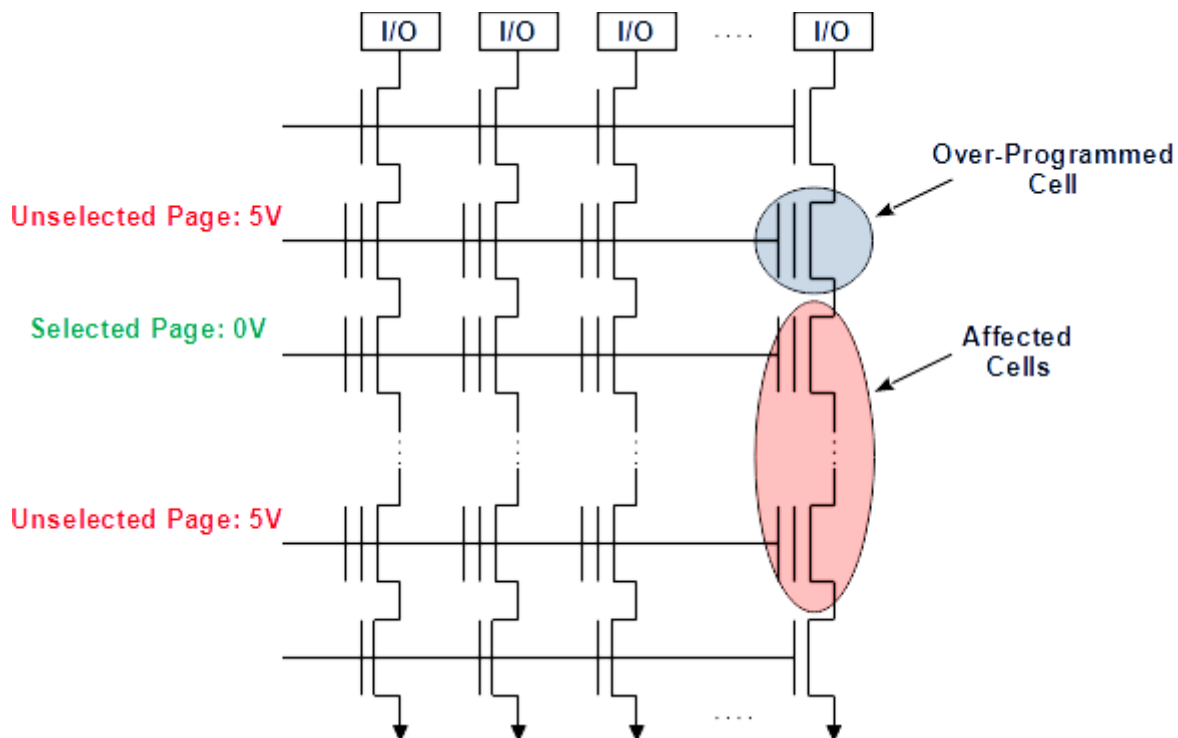


Figure 41: Over-programming errors.

13.5. Data Retention Errors

The data stored in flash memories tend to get corrupted over time. This is known as a data retention error. Retention errors are caused by loss of charge stored in the floating gate. Even though the gate oxide is an insulating layer, electrons stored in the floating gate still leak through it from time to time. With longer durations, the loss of charge accumulates, eventually changing the programmed state of the cell and causing a data error. To recover from this error, the block need to be erased after copying the contents to another block.

Retention errors can happen to any cell in any block of the flash memory. Due to wear of the oxide layer, memory cells with more program erase cycles are more likely to experience retention errors. Temperature is another factor which contributes to retention error; the higher the temperature, the greater the chance for a retention error. In MLC, TLC, and QLC memories that store more bits in each memory cell, the cells with more programmed electrons (closer to binary 0) are more prone to leakage of charge. Retention errors depend on many aspects of the flash manufacturing technology such as lithographic node, oxide thickness, and so on. Data retention is a key parameter in all flash datasheets.

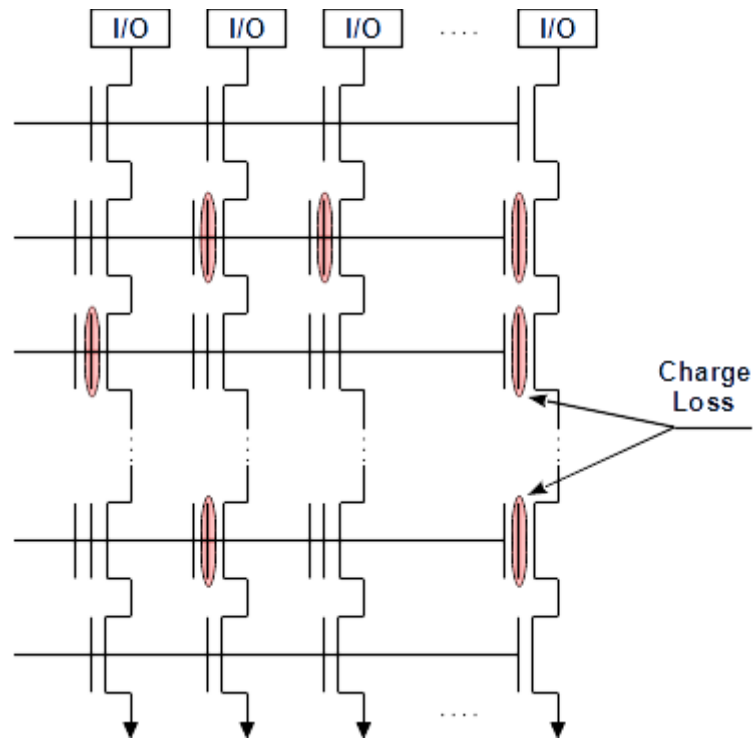


Figure 42: Data retention errors.

Improving data retention:

- limit program / erase cycles in blocks that require long retention;
- limit reads to reduce read disturb.

Note: This chapter draws on the following publications:

- Avinash Aravindan, Flash 101: Errors in NAND Flash: <https://www.embedded.com/flash-101-errors-in-nand-flash/>
- Jim Cooke, The Inconvenient Truths of NAND Flash Memory, Flash Memory Summit, August 2007

Versions History

Revision	Date	Comment
0.1	2013, December 04	<ul style="list-style-type: none"> Initial draft
0.2	2017, October 16	<ul style="list-style-type: none"> MBN file size updated
0.3	2018, November 19	<ul style="list-style-type: none"> Discard invalid block(s) data chapter added Read JEDEC parameters page chapter added Features availability information refined Small text corrections
1.0	2021, June 16	<ul style="list-style-type: none"> Document formatting changed from scratch Added chapter Brief comments on NAND flash inwards Updated chapter Access method window Updated chapter Device operation options window Added chapter Using Multi-Project feature for NAND flash Added chapter Customized NAND flash support Added chapter Frequently asked questions Added appendix A: Errors in NAND flash – the background
1.1	2021, August 18	<ul style="list-style-type: none"> Missing source references fixed
1.2	2022, November 9	<ul style="list-style-type: none"> Added FAQ question I need to do a test with a bad block so I want to make a sample with a bad blocks I make on my own.
1.3	2024, April 5	<ul style="list-style-type: none"> Fixed chapter name User Area – Max. Allowed Number of Invalid Blocks
1.4	2025, April 3	<ul style="list-style-type: none"> Font changes, with no changes of meaning