NAND Flash Memories
and
Programming NAND Flash Memories Using Dataman Device Programmers

Application Note

August, 2008
an_dataman_nand_flash, version 2.10

Table of contents

List of tables

List of figures

Introduction

The NAND Flash architecture was introduced by Toshiba in 1989. Today, NAND Flash architecture together with NOR Flash architecture dominates the non-volatile Flash market. NAND Flash's high-density, low power, cost effective, and scalable design make it an ideal choice to fuel the explosion of new multimedia products that are entering the market. Advances in system design techniques also enable the more cost effective NAND Flash to replace NOR Flash in a significant percentage of traditionally NOR Flash applications.

Differences between NAND and NOR

There are major differences between NAND and NOR highlighted in *Table 1*. It shows why NAND memories are ideal for high-capacity storages, while NOR memories are used for code storage and execution.

	NAND	*NOR*
Capacity*[1]	~ 32Gbit	~1Gbit
Access method	Sequential	Random
Interface	I/O interface	Full memory interface
Performance	Fast read (serial access cycle) Fast write Fast erase (approx. 2ms/block)*[2]	Fast read (random access) Slow write Slow erase (approx. 1s/block)*[3]
Life Span	100 000 – 1 000 000	10 000 – 100 000
Price	Low	High

Table 1: The major differences between NAND and NOR

*[1] – By NAND Flash manufacturers materials (available at 01/2006)
*[2] – Toshiba TH58NVG1S3A (1 block is 16kB)
*[3] – Intel StrataFlash® P30 family (1 block is 128kB)

Physically, the NAND architecture uses a smaller transistors, because it doesn't have to "pull-down" a whole bit-line. A NAND bit line is a series of transistors so each transistor only has to pass a small amount of current. *Figure 1* shows, how the transistors are connected for NAND and NOR architecture and difference of the cell size.

Due to the efficient architecture of the NAND Flash, its cell size is almost half the size of a NOR cell. This enable NAND Flash architecture to offer higher densities with larger capacity on a given die size, in combination with a simpler production process. The NAND architecture is more cost-effective for higher capacities than the NOR architecture. As mentioned above, the NAND Flash memories are ideal for data storage (for instance in MP3 players or digital cameras).
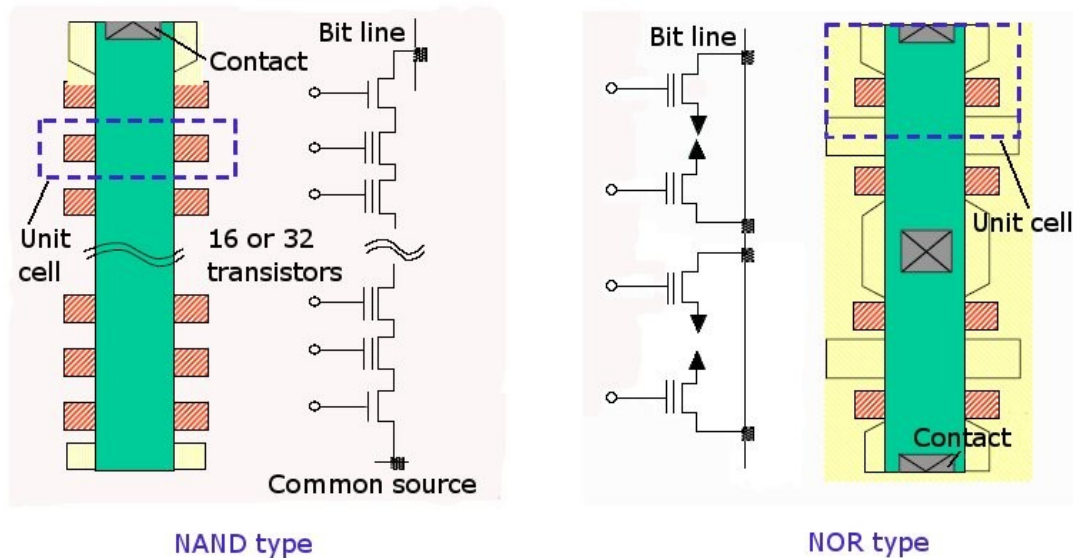
Figure 1: NAND and NOR cell architecture (by Samsung [4])

Above depicted technology is well-known Single-Level Cell (SLC) technology. One memory cell can hold one bit of information expressed by voltage level "H" or "L". Sometimes also Multi-Level Cell (MLC) technology is used. In this case one memory cell can hold two or more bits of information. By storing more bits per cell, MLC memories achieve slower transfer speed, higher power consumption and lower cell endurance than SLC based memories. However, both technologies use the same I/O interface and externally act in the same way (except timing and power requirements).

In the NOR Flash all memory locations are guaranteed to be good and to have the same level of endurance, so a relatively large amount of extra memory cells are fabricated on the die – these are used to repair defects in the memory array in order to produce a device that has all good memory locations. To improve yields and keep costs down, the NAND Flash devices contain randomly located invalid blocks in the array. These blocks must be identified before programming the device to avoid losing data stored in the bad memory cells. In some publications, a term "bad block" is used instead of "invalid block", however, the meaning remains the same.

From the practical standpoint, the main difference between NOR Flash and NAND Flash lies in the interface. The NOR Flash has a full memory-mapped random access interface (like an EPROM) with dedicated address and data lines. On the other hand the NAND Flash has no dedicated address lines. It is controlled by sending command, addresses and data through a 8/16 bits wide bus (I/O interface) to an internal registers.

The NAND Flash memory array description

The NAND Flash memory is composed of the blocks of pages, one block is usually composed of 16, 32 or 64 pages. For most NAND Flash devices there are 512 bytes / 256 words in the "Cell Array" page area (also called "data area") and an extra 16 bytes / 8 words in the "Spare Cell Array" page area (also called "spare area"). There are total 528 bytes / 264 words per page (see *Figure 2*) and such page is called "small page".

For huge capacities (usually 1 Gbit and more) "large page" is used (see *Figure 3*). It contains 2048 bytes / 1024 words of data area and 64 bytes / 32 words of spare area (2112 bytes / 1056 words total).
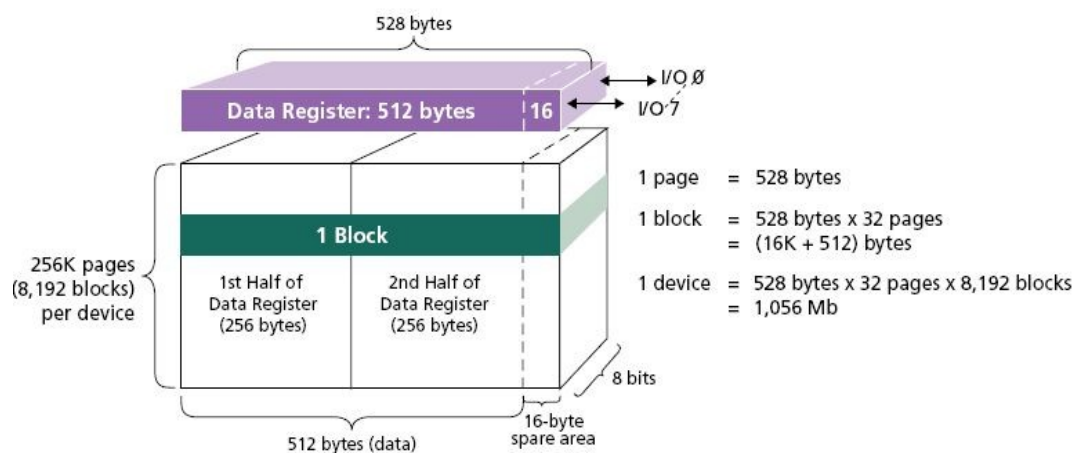

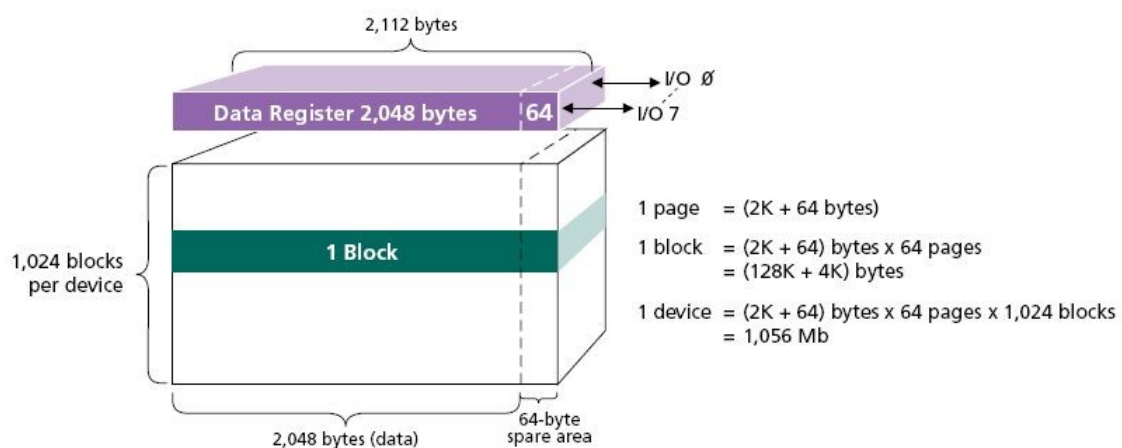*Figure 2: 1 Gbit small page architecture example (by Micron [6])*

Figure 3: 1 Gbit large page architecture example (by Micron [6])

In the NAND Flash the read and program operation takes place on a page basis (i.e. 528 bytes at a time for most of NAND devices) rather than on a byte or word basis like NOR Flash – the size of data I/O register matches the page size. The erase operation takes place on a block basis. There are only three basic operations in a NAND Flash: read a page, program a page and erase a block.

In a page read operation, a page of 528 bytes is transferred from memory into the data register (see *Figure 2* and / or *Figure 3*) for output. In a page program operation, a page of 528 bytes is written into the data register (see *Figure 2 and / or Figure 3*) and then programmed into the memory array. In a block erase operation, a group of consecutive pages is erased in a single operation. In a brand new ("virgin") device all usable (not marked as INVALID) blocks are in erased state.

Usually, there are various extended features offered in addition to those basic ones. These mainly are (see your device datasheet for more info):
- manufacturer ID read
- device operation status read
- reset command
- CE don't care (de-selecting the device doesn't terminate operation in progress)
- copy-back (internal movement of the block into another memory location avoiding time-consuming data transfers from and back to chip)
- partial page program (only a portion of a page may be programmed at a time and the rest may be programmed at some other time, avoiding block erasing)
- cache operation (pipe-lined program or read operation using additional cache register)
- OTP area (one-time-programmable area to store vendor unique data (serial number, digital rights...))
- auto page 0 read (boot-like feature, page 0 is loaded into data register automatically after power-on or reset)
- block lock / unlock (locking / unlocking the blocks to prevent data loss on unintended software behaviour)

The Spare Cell Array description

The NAND Flash manufacturers use the spare area for marking invalid blocks during production process, so that marked devices are shipped to the customers. In function, all bytes from spare area are equivalent to bytes from data area and can be used to store user's data. We recommend to assign "spare" bytes in compliance with Samsung's [5] standard (see next page).

- Small page (528B) 8-bit organization NAND Flash:

Cell Array 512B												Spare Cell Array 16B			

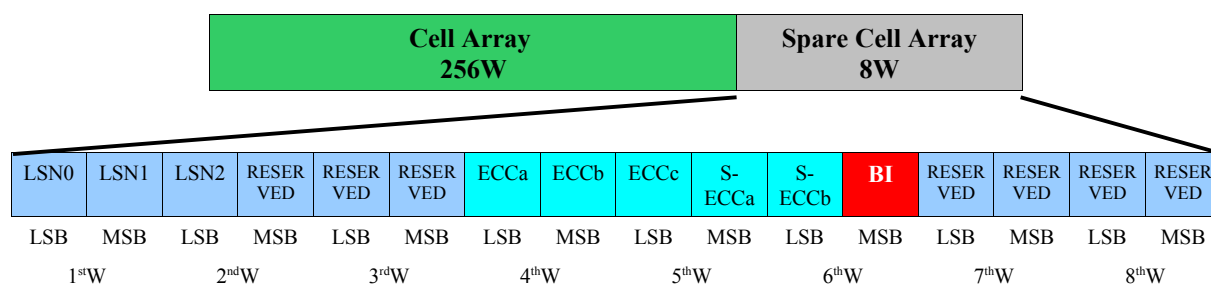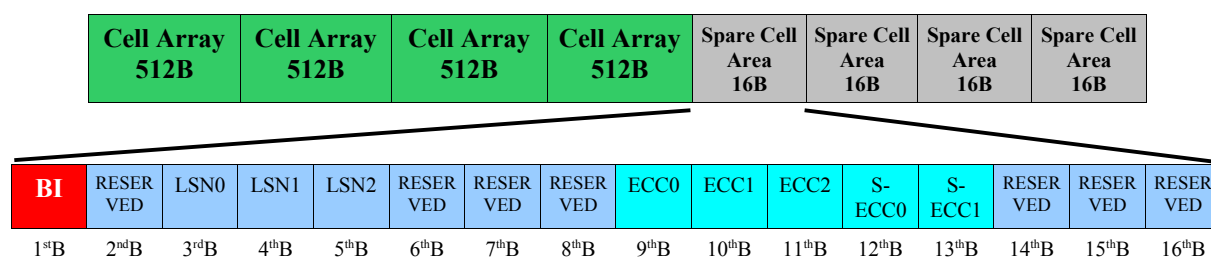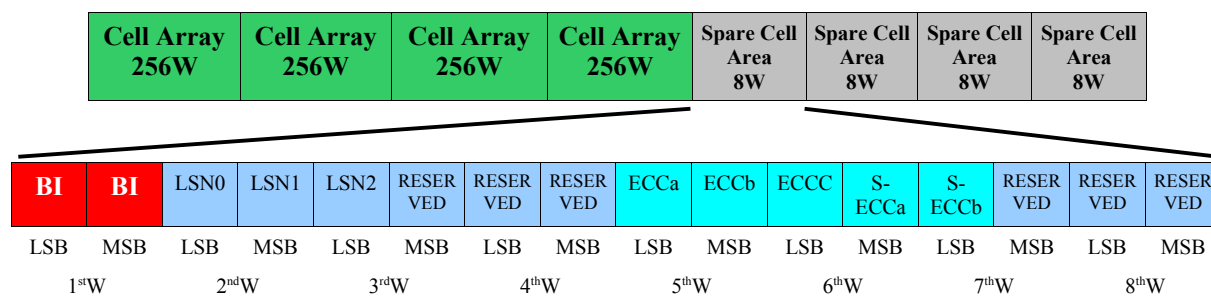| LSN0	LSN1	LSN2	RESERVED	RESERVED	BI	ECC0	ECC1	ECC2	S-ECC0	S-ECC1	RESERVED	RESERVED	RESERVED	RESERVED	RESERVED
1stB	2ndB	3rdB	4thB	5thB	6thB	7thB	8thB	9thB	10thB	11thB	12thB	13thB	14thB	15thB	16thB
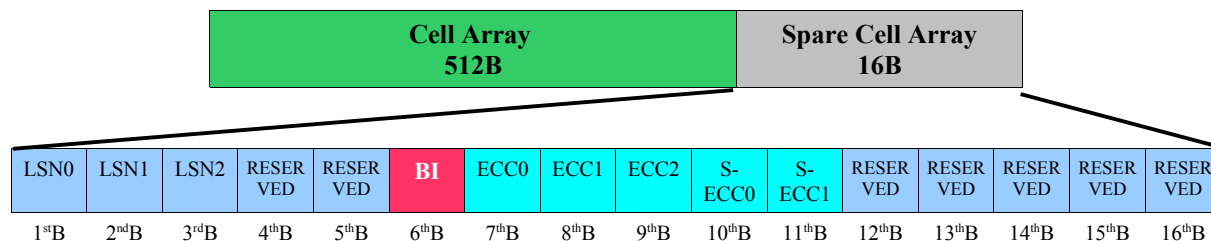
- Small page (264W) 16-bit organization NAND Flash:

Cell Array 256W	Spare Cell Array 8W

LSN0	LSN1	LSN2	RESERVED	RESERVED	RESERVED	ECCa	ECCb	ECCc	S-ECCa	S-ECCb	BI	RESERVED	RESERVED	RESERVED	RESERVED
LSB	MSB	LSB	MSB	LSB	MSB	LSB	MSB	LSB	MSB	LSB	MSB	LSB	MSB	LSB	MSB
1stW		2ndW		3rdW		4thW		5thW		6thW		7thW		8thW	

- Large page (2kB) 8-bit organization NAND Flash

Cell Array 512B	Cell Array 512B	Cell Array 512B	Cell Array 512B	Spare Cell Area 16B	Spare Cell Area 16B	Spare Cell Area 16B	Spare Cell Area 16B

BI	RESERVED	LSN0	LSN1	LSN2	RESERVED	RESERVED	RESERVED	ECC0	ECC1	ECC2	S-ECC0	S-ECC1	RESERVED	RESERVED	RESERVED
1stB	2ndB	3rdB	4thB	5thB	6thB	7thB	8thB	9thB	10thB	11thB	12thB	13thB	14thB	15thB	16thB

- Large page (1kW) 16-bit organization NAND Flash

Cell Array 256W	Cell Array 256W	Cell Array 256W	Cell Array 256W	Spare Cell Area 8W	Spare Cell Area 8W	Spare Cell Area 8W	Spare Cell Area 8W

BI	BI	LSN0	LSN1	LSN2	RESERVED	RESERVED	RESERVED	ECCa	ECCb	ECCC	S-ECCa	S-ECCb	RESERVED	RESERVED	RESERVED
LSB	MSB	LSB	MSB	LSB	MSB	LSB	MSB	LSB	MSB	LSB	MSB	LSB	MSB	LSB	MSB
1stW		2ndW		3rdW		4thW		5thW		6thW		7thW		8thW	

LSN	...	Logical sector number
ECC	...	ECC code (described below) for Cell Array data
S-ECC	...	ECC code for LSN data
BI	...	invalid block information

Invalid block management

Since NAND architecture was designed to serve as a low cost mass storage medium, the standard specification for the NAND allows the existence of invalid blocks in a certain percentage (less than 2 % max.). The block is marked being invalid in case it contains bad memory location. The invalid block table can be stored in one of the good blocks on the chip or in another chip in the system. Invalid block table is required because there is a finite number of write and erase cycles that NAND Flash can achieve. Because all flash memories will eventually wear-out and no longer be usable, the table needs to be maintained to track blocks that fail during use. Allowing the existence of invalid blocks increases the effective chip yield and enables a lower cost. The existence of invalid blocks does not affect the good blocks because each block is independent and individually isolated from the bit lines.

Invalid blocks can be assorted into two groups: inherent invalid blocks and acquired invalid blocks. Inherent invalid blocks arise during production process at factory. This includes a failure of intentionally isolated block type and/or cell failures, which occur during electrical test. These blocks are identified in invalid block information at the time of shipment of Flash, for maximum number of inherent invalid blocks see product datasheet. Blocks that are considered to be invalid are marked, usually by writing non FFh value (typically 00h) in byte 517 in first two pages of a invalid block (device dependent, see product datasheet for details). A common invalid block map building algorithm flowchart is shown on figure below.
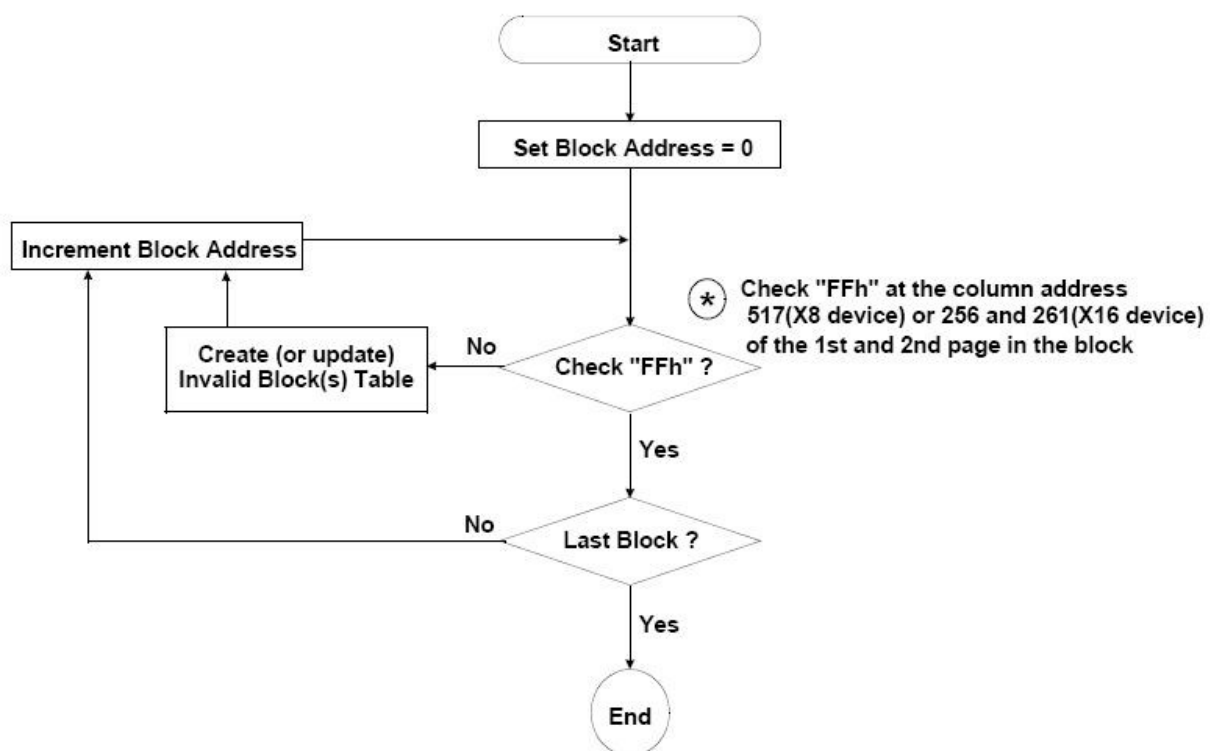

Figure 4: Invalid block map building algorithm flowchart

Acquired invalid blocks are not identified by the factory, these blocks originate at a customer site. It is because NAND Flash has a finite lifetime and will eventually wear-out. Since each block is an independent unit, each block can be erased and reprogrammed without affecting the lifetime of the other blocks. Each good block can be typically reprogrammed for 100 000 up to 1 000 000 times before the end of life. Therefore blocks should be marked as invalid and no longer accessed if there is either a block erase or a page program failure. Acquired invalid blocks are usually marked in the same way as the inherent invalid blocks.

Once you have erase invalid block, the non FFh bytes will be also erased. If this occurs, the re-identification of invalid blocks is very difficult without having blocks tested at special conditions. So if the list of invalid blocks is lost, recovering invalid block locations is extremely difficult. Therefore, it is recommended to collect informations about invalid block before erasing Flash, and after entire device is erased, re-identify and mark invalid blocks in the memory again.

To handle invalid block in the embedded system a special layer of the software is required. Therefore, to program NAND Flash devices, it is necessary to handle invalid blocks in exactly the same way as they are handled by the embedded system. There are many ways how to handle invalid blocks, but no one is accepted as standard method.

For example, one common method is just to skip over the invalid blocks and place the data into the known good blocks – this method is called "Skip Block" method. Another common method is "Reserved Block Area", which replaces invalid blocks with known good blocks from block reservoir and keeps the replacement table at known position in the memory.

In addition, some applications require Error Checking and Correction (ECC) to be calculated for each page of data. The ECC data is used to detect when a memory location goes bad and fix the bad bit (if ECC algorithm enables this feature). The ECC data should be written into spare area. Algorithms used for this purpose are frequently called Error Correction / Error Detection (EC/ED) algorithms.

Skip Block method

This method is very straightforward. The algorithm starts by reading the entire spare area (only) of whole memory. The addresses of the marked invalid blocks are collected (see flowchart on *Figure 4*). Next, the data are sequentially programmed (page by page) into the target Flash device. When the target address corresponds to an invalid block address, appropriate pages are stored in the next good block, skipping the invalid block (see *Figure 5*). Since this invalid block is skipped, the data in the spare array indicating the presence of the invalid block are kept. Therefore the user's system can build a table of invalid blocks addresses by reading the spare area of all the blocks at boot time. Because the invalid block exists, the user's data should not exceed the size of good blocks.
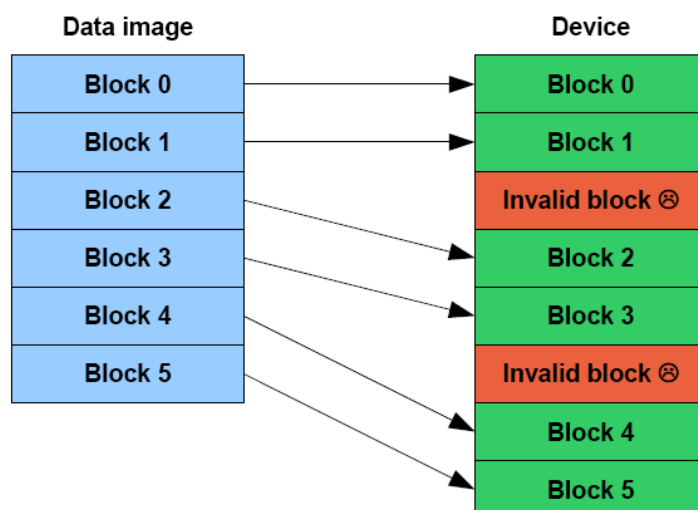

Figure 5: Skip block method

Reserved Block Area method

The Samsung's "Reserved Block Area" method is based on the principle that the invalid blocks in the user's system can be replaced with good blocks by re-directing the system to the known good block. The programming algorithm works by first determining which blocks will be used as the User Block Area (UBA), which blocks will be used for RBA map table (Reserved Block Area - RBA) and which blocks will act as block reservoir (see *Figure 7*). Next, the algorithm reads the spare area of the device and constructs a map table in the RBA. Only the first and second (valid) blocks from RBA are used for this table and his backup copy (respectively). The map in the RBA contains information on how to substitute invalid blocks in the UBA with good blocks from the block reservoir. The data fields in the map table are shown below:
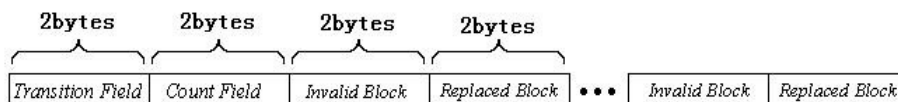

Figure 6: The RBA map table

The *Transition Field* is always FDFEh. The *Count Field* is incremented by one for each page of the map table. The data pair *Invalid Block / Replaced Block* shows the address of the invalid block and corresponding address of the replacement block, respectively. The rest of the page consists of this kind of data pairs. Since there are 512 bytes, the maximum number of data pair entries is 127. Usually this will be sufficient since the number of invalid blocks is typically less than 1% of the total number of blocks for new devices being programmed. Of course, acquired invalid blocks can be generated during usage, so the system (using this method) will have to identify these blocks and update the map table.

The second block is used to duplicate the RBA table in case of the first block becomes corrupted during usage. (However, the page numbering continues from original stored in the first block, so it is not an exact copy.)

All fields of RBA map table use *little endian* (lower significant byte first) protocol.
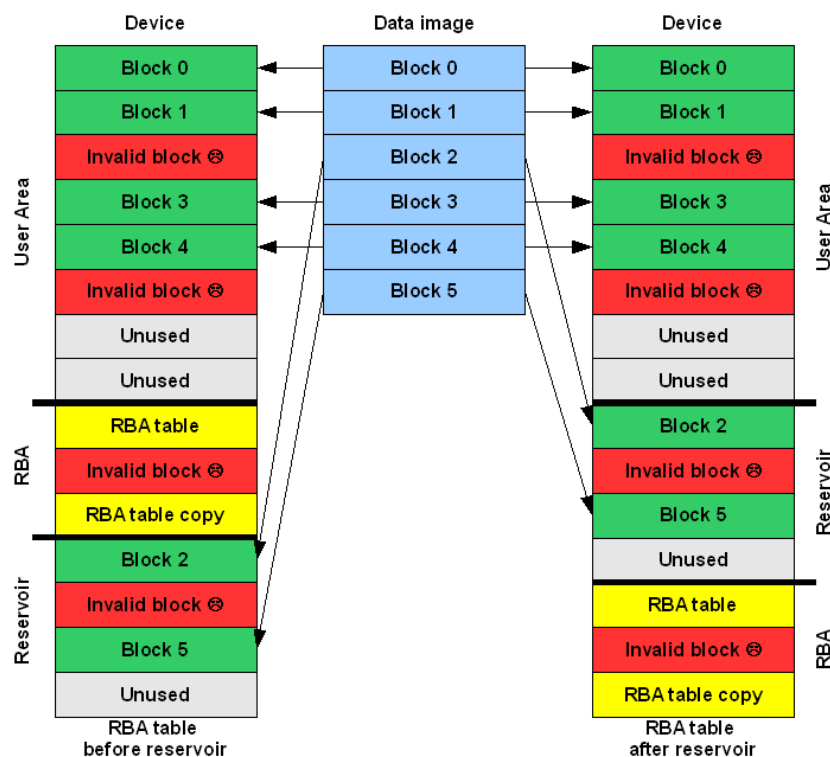

Figure 7: The Reserved Block Area method

Error Checking and Correction

The usage of an error correcting mechanism is essential in order to maintain the integrity of stored code. Soft errors (especially during programming) occur at a rate of approximately 10^{-10} . It is recommended to use single error correcting / double error detecting (SECDED) ECC algorithm to fully utilize the potential of NAND Flash memory. The most common used one is Hamming ECC algorithm recommended by Samsung (see documentation available at http://www.dataman.com).

File systems

Some embedded products use NAND Flash to store "files" as well as the boot code, OS code or simple as data files (e.g. MP3 players, USB keys, Digital Cameras...). Therefore additional software is required to manage the NAND Flash memory used as a file storage system. This may include a format function, wear leveling, garbage collection and the ability to defragment the memory area. Most of the NAND manufacturers offer software that manages these higher level functions and are willing to license the software to an end user. There also exist variety of both, proprietary software drivers available from third parties and free software drivers available from open source projects (see *Table 2* for some software drivers, many other can be found on World Wide Web).

Product Name	Company/Sponsor	URL link
F1Pack Angel & Jet	TOKYO ELECTRON DEVICE	http://www.inrevium.jp/nand/f1packangel.html
FlashFX	DataLight	http://www.datalight.com
JFFS2	Red Hat	http://sources.redhat.com/jffs2/
NAND File system	Kyoto Software Research	Contact Toshiba (TAEC)
smxFFS	Micro Digital	http://www.smxinfo.com
TargetFFS-NAND	Blunk Microsystems	http://www.blunkmicro.com/ffs_nand.htm
TrueFFS	Wind River Systems	http://www.windriver.com/products/device_technologies/middleware/true_ffs/
YAFFS	Toby Churchill	http://www.aleph1.co.uk/yaffs/

Table 2: Third party software drivers manufacturers

Programming NAND Flash memories using Dataman device programmers

Dataman, manufacturer of the world's leading device programmers, cannot ignore your needs in the field of NAND Flash devices. Therefore, a function-reach support for NAND Flash programming was added to our product portfolio. Currently, NAND Flash memories are supported by 48Pro, 48Pro+, 48Pro2C and 48Pro2 universal programmers, as well as 448Pro, 448Pro+ and 448Pro2 multiprogramming systems. The following pages are intended to give you a guide of how to get the most from your Dataman programmer.

First of all, select your NAND Flash device by *[Select]* button or *<Alt+F5>* key-press. Then click the *Access Method <Alt+S>* link in the bottom of the program window or simply press short-cut *<Alt+S>* for extended device options menu. You will be faced with dialog window shown below (see *Figure 8*). There are two main settings you need to define – Invalid block management and Spare area usage. The following paragraphs will give you an explanation.
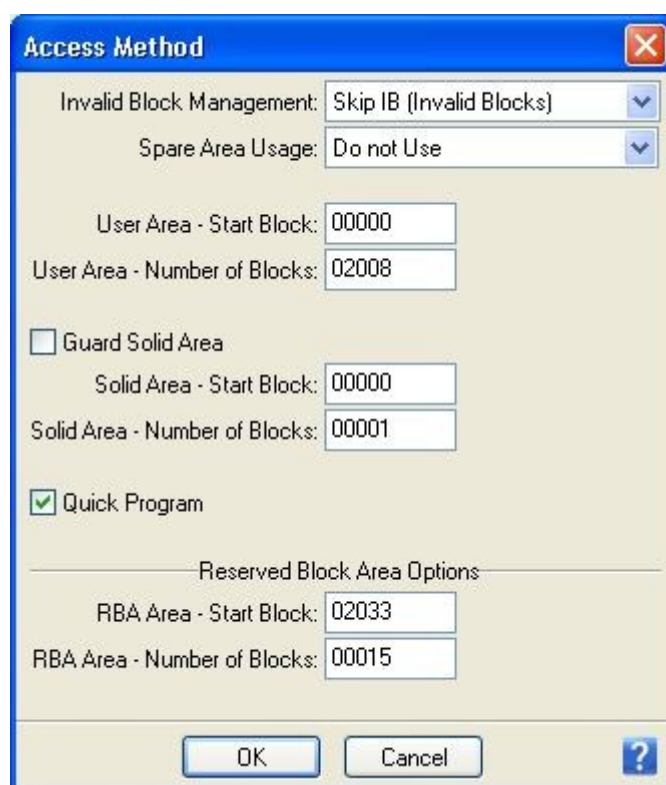

Figure 8: Access Method dialog window
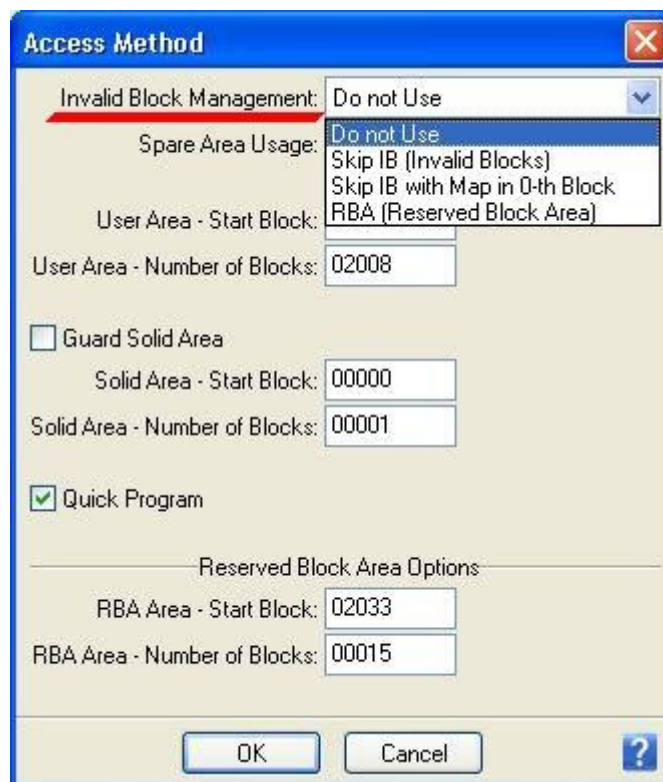
Invalid Block Management drop-down menu

Figure 9: Invalid Block Management drop-down menu

Invalid Block (IB) management drop-down menu offers four basic settings:

Do not Use

No management method will be applied. All memory blocks will be treated as being good. In other words, all blocks will be programmable and erasable.

There is a hazard of invalid block information loss. Use this option very carefully.

Note:
We recommend it only for development purposes, since it is possible to read "true" memory image (byte-by-byte) and repair fake (but formerly known) IB information in spare area caused by software failures in debugging phase.

Skip IB (Invalid Blocks)

Skip Block method will be used (see page 10). Firstly, the IB map will be made, based on block validity information in spare area. Invalid blocks will be no longer accessed during operation (they become "invisible"). If invalid block should be accessed, it will be skipped over and operation will continue from next good block.

Skip IB with Map in 0-th Block

The principle is identical with previous option. During programming, IB map will be made, based on IB information in spare area. After finishing program operation, this map will be written into the 0-th block. This is

for compatibility issues, if upgrading older HW systems. (Old NAND Flash memory chips didn't use spare area and IB map was written into the 0-th, manufacturer guaranteed good block. This approach is not further used in new chips.) Also Blank check and Erase operations will use spare area based IB map. During Read / Verify operations, IB map will be build based on copy in the 0-th block. If you need a different approach, please contact Dataman.

Note:
Using this method, the 0-th block becomes "implicitly booked area" and you cannot include it into your User Area. You will obtain an error message in such case.

RBA (Reserved Block Area)

Reserved Block Area method will be used (see page 11). Please, find more information about User Area and RBA Area settings in later paragraphs.

Following settings are closely related to the options described in previous:
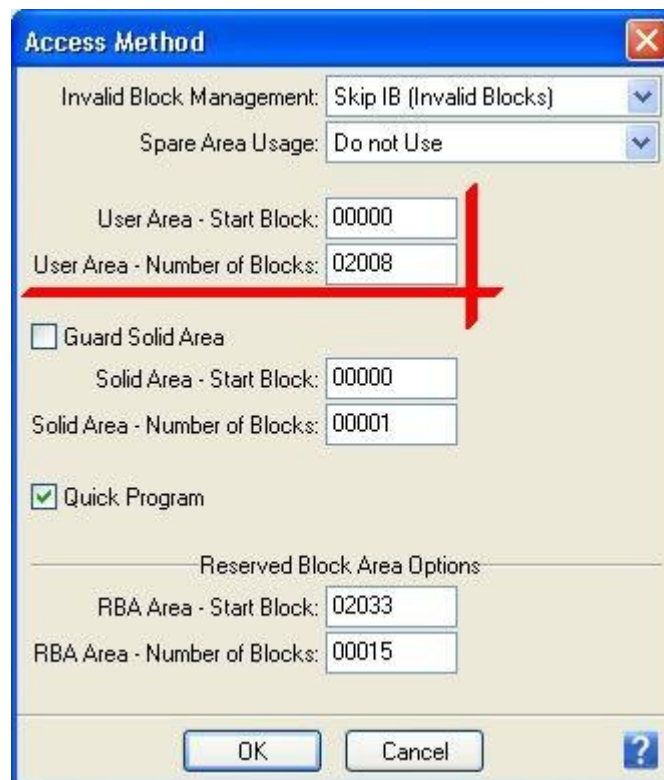
User Area Settings

Figure 10: User Area settings

User Area is a part of memory, where data have to be written to or read from.

Two edit-fields allow entering start block and number of blocks of User Area. Using *Figure 10* as example, User Area will start from the 0-th block and will cover 2008 blocks. It means, it should be spread from block No. 0 up to block No. 2007. But really, it can cover more blocks, if there are some invalid blocks in the chip and some skip-based method is used (e.g., if there are 2 invalid blocks, the last block will be block No. 2009). Please, remember this in case of debugging process.

User Area settings are mandatory for Read, Verify and Program operations, not regarding the IB management method selection. Erase and Blank check operations are always performed on whole chip, not regarding User Area settings.

For your comfort, you can use both, decimal and hexadecimal radix while entering the numbers. The fields are independent on each other. Please, use $ (dollar sign) as hexadecimal prefix (see *Figure 11*). This feature is valid for all edit-fields in Access Method window.

Note:

Your NAND Flash device can contain invalid blocks. These blocks reduce the usable chip size. If there are too many invalid blocks, it is possible, that your User Area cannot fit into decreased chip capacity. Such case is reported as soon as it is detected (programming of the device will be aborted).

Figure 11: You can use also hexadecimal radix

Solid Area Settings

Figure 12: Solid Area settings

Solid Area is a part of User Area that cannot be interrupted by invalid block(s). There can be various reasons for necessity of continuous memory space (e.g. boot code).

It is possible to enter start block and number of blocks, likewise in case of User Area. The settings are accepted only if Guard Solid Area check-box is checked (they are don't care leaving it unchecked).

Solid Area feature is possible to use with any IB management method (except ***Do not Use***, of course).

Note:

It is possible to locate Solid Area anywhere within User Area. However, it is impossible to safely guarantee Solid Area requirements in case of skip-based IB management method in conjunction with different Solid Area start block and User Area start block. Therefore, in such case, Solid Area is automatically extended to the beginning of User Area. This is the reason, why you can obtain error message even in case, that you are sure the blocks you want to protect are all valid. This inconsistence doesn't occur using RBA.

Example:

Having defined:
User Area – Start Block: 0
User Area – Number of Blocks: 1000 (i.e. 0-999)
Solid Area – Start Block: 120
Solid Area – Number of blocks: 12 (i.e. 120-131)

will internally result in:
User Area – Start Block: 0
User Area – Number of Blocks: 1000 (i.e. 0-999)
Solid Area – Start Block: 0
Solid Area – Number of blocks: 132 (i.e. 0-131)
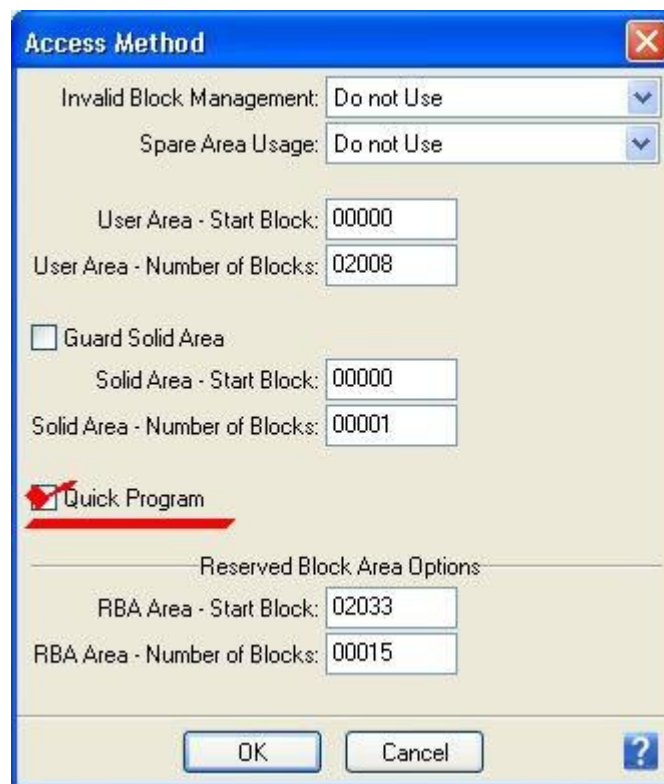
Quick Program Check-box

Figure 13: Quick Program check-box

NAND Flash memories are modern chips with highly sophisticated internal controller. As was mentioned in previous paragraphs, there is a data register, that acts as I/O register between the system and memory. After entering the program command, internal controller will perform all actions required to write data from the register into the appropriate memory cells. After the process is completed, it performs internal validation. If process have failed, it re-tries to write data several times. If internal timeout is over, it reports program failure. This means, the block becomes invalid (e.g. due to over-using).

But there is another, rather opposing feature of NAND Flash memories – partial page write. You can write only several bytes / words into page without affecting other bytes / words on the page or erasing the block. It assumes, these bytes / words are not programmed yet (see your memory datasheet for more details). In such case data register contains only partial page data, so validation cannot be fully performed.

From this reason the compromise was accepted – only those bits having been programmed to "zeros" are internally validated. So internal validation process can detect, if "zero" is not "zero", but cannot detect, if "one" is not "one".

Having Quick Program check-box checked, programming action will be performed in the way described above. Leaving the check-box unchecked, an additional software verification will be performed after completing the page programming process, so "all ones are ones" will be checked too. Of course, it will take some additional time (on that account "quick" or "non-quick" programming, respectively).

Note:

In similar way also erase command is performed. Therefore, options **Blank check after erasing** and **Verify after programming** (see **Device Operation Options** – use shortcut **<Alt+O>** in your Dataman program) is not compulsory with this kind of device.

Reserved Block Area Options

Figure 14: Reserved Block Area options

Using Reserved Block Area options you can define the areas for RBA method. There are 3 areas:
- User Block Area – for user data, explicitly defined using User Area Settings, see page 31.
- Reserved Block Area – for RBA table, explicitly defined here
- Block Reservoir – implicitly defined between the end of User Area and the beginning of Reserved Block Area (if **after Block Reservoir** is set), or between the end of Reserved Block Area and the end of device (if **before Block Reservoir** is set).

It is possible to enter start block and number of blocks, likewise in case of User Area. The settings are accepted only if **RBA (Reserved Block Area)** method is selected in Invalid Block Management drop-down menu. Otherwise, they are not used.

RBA Table – Start Block: it must be defined behind the end of User Area. There must be some free blocks leaved (at least one block) for block reservoir before or after Reserved Block Area, regarding to RBA Table location selection. You will obtain an error message, if it is not fulfilled.

RBA Table – Number of Blocks: two valid blocks are really used, one for RBA table and other for its copy. One block is added to assure there will be sufficient space also in case of invalid block in RBA Area. So you need define at least three block for this setting. You will obtain an error message, if it is not fulfilled.

RBA Table should be located: you can specify the device layout. If after Block Reservoir is set, the data order is User Area – Block Reservoir – RBA Table (see *Figure 11* on right-hand). If before Block Reservoir is set, the data order is User Area – RBA Table – Block Reservoir (see *Figure 11* on left-hand).
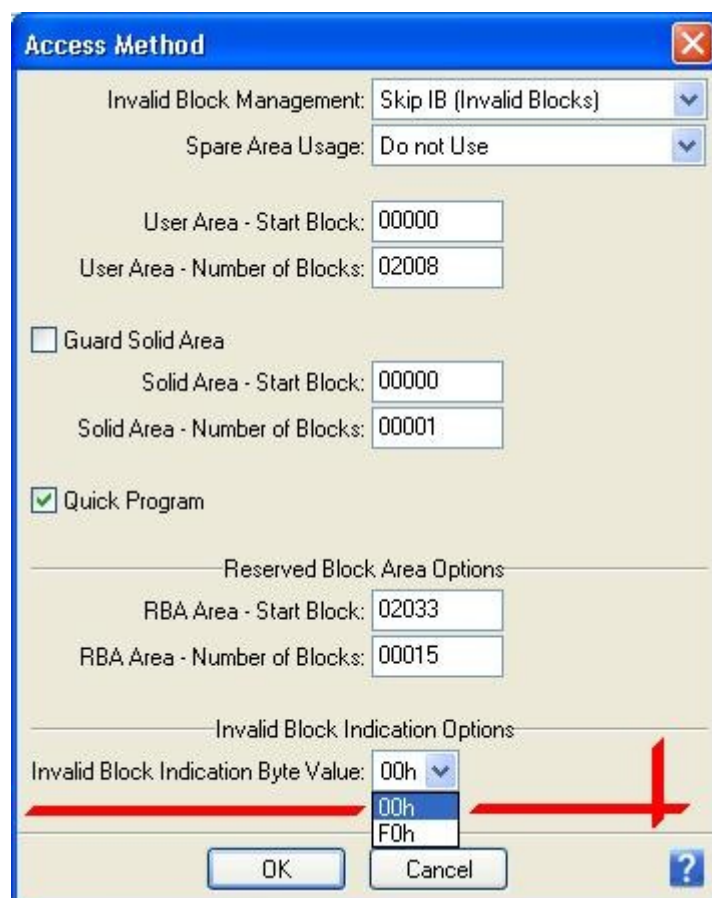
Using *Figure 14* as example:
User Area spreads in blocks: 0 – 1004
The blocks reserved for RBA Table are: 1009 – 1023
Block Reservoir spreads in blocks: 1004 – 1008 (automatically placed)

Note:
We highly recommend to use RBA method together with ECC coding. It provides the maximum chip performance for your system, with easy software implementation.

Invalid Block Indication Options

Figure 15: Invalid Block Indication Options

There is new setting available in pg4uw software version 2.36 and newer – **Invalid Block Indication Options.** It enables to define **Invalid Block Indication Byte Value** that will be written into **BI** byte/word to indicate the block validity (consult pages 8 and subsequent for more information about BI byte/word usage). There are two values possible: **00h** (commonly used) and **F0h** (specified within SmartCard specification to indicate invalid blocks that become invalid during use).

Spare Area Usage drop-down menu
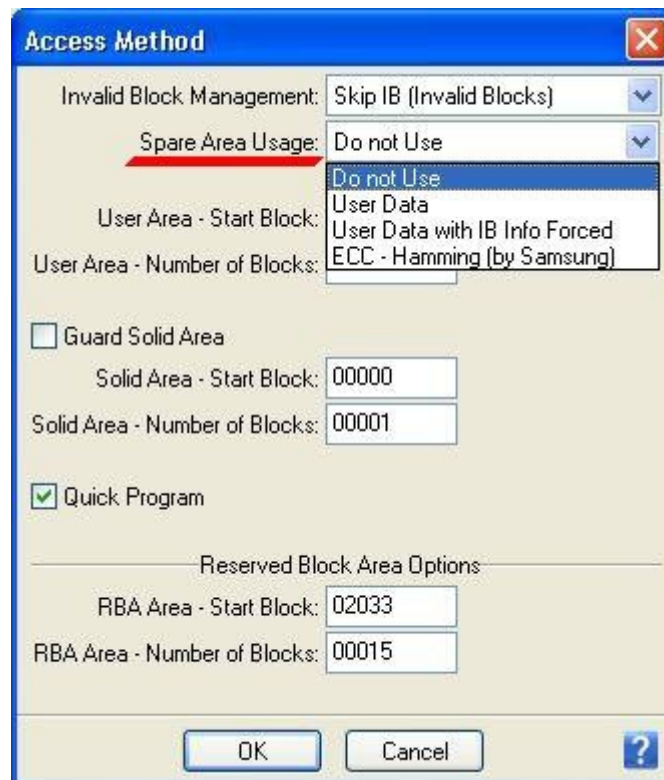
Figure 16: Spare Area Usage drop-down menu

Spare Area Usage drop-down menu offers four basic settings:

Do not Use

Spare area will be not used. Only data programmed into spare bytes / words will be block validity information. Samsung's standard [5] recommends to use "non FFh" value. Our programmers use "00h" to indicate both, inherent and acquired invalid blocks (if you need different symptom system, please contact Dataman).

User Data

From programmer point of view, there will be no difference between data area and spare area. Data from buffer will be programmed continuously into both page segments.

There is a hazard of invalid block information loss. Use this option very carefully.

Note:
Using this option, programmed data are not checked in any way. Therefore, block validity information can be easy overwritten and lost. Very probably, it will be impossible correctly read and verify your device in this case. Selecting *Do not Use* in both, Invalid Block Management and Spare Area Usage drop-down menus can help, but invalid block occurrence in User Area will ruin also this chance.

This option can be useful, if you need to use your own error correction (or cryptography, etc...) algorithm. We strongly recommend do not use *Quick Program* option in such case. It is the way, how to get the most complete verification of programming process.

User Data with IB Info Forced

This is very similar to previous option. To avoid troubles with block validity information loss, bytes / words at positions corresponding to block validity information (see Samsung's standard [5]) are ignored and correct values are forced instead of user data. Using this option, there are no problems with device reading and / or verification. You still can write your own data into remaining spare bytes / words.

Note:

User data (loaded into buffer) at block validity information position are ignored, anyway they are expected. Therefore, in image creation phase, you cannot omit these bytes / words. You may write any values at relevant positions, they will be don't care.

ECC – Hamming (by Samsung)

Single error correction with double error detection Hamming algorithm is used, in accordance with Samsung's recommendation (see documentation available at http://www.dataman.com).

A 512 byte / 256 word version is used. No special settings or assumptions must be taken into account using this option.

Tolerant Verification

Figure 17: Tolerant Verification options

NAND flash array is erroneous data storage medium in its nature. To overcome random bit errors the error detecting / correcting algorithm (ECC) must be employed. Our software supports Hamming ECC code (as specified by Samsung) for data area. This code is capable to detect 2 single-bit errors and correct 1 single-bit error in frame of 512 bytes.

However, there are plenty of reasons for using another ECC algorithm. For example, ECC algorithm capable to correct 4 single-bit errors in frame of 512 bytes is mandatory for new multi-level cell (MLC) devices. In order to let your hands free while choosing the ECC algorithm, there is new feature implemented in pg4uw software from SW version 2.38 – **Tolerant Verification**. This kind of verification accepts specified number of single-bit errors in frame of specified length. If such error is found, acceptable error is reported in log window instead of verify error and verification operation continues. It is assumed, that such error will be in real application detected and corrected by on-the-fly ECC algorithm. This rule applies only for data area. Spare area is verified without error tolerance, since it contains ECC control words (and, may be, some LSN or another user-specific data)

Supported ECC algorithm must meet following requirements:
- the data area is subdivided into several frames of identical length (usually 512 bytes);
- ECC control words are written in spare area.

If you want to use tolerant verification, you must prepare the data image firstly. The buffer is considered to be a continuous space of pages. Each page contains data area and spare area. ECC control words must be pre-computed in spare area. Our software doesn't perform any ECC computation.

You must use **User Data** or **User Data with IB Info Forced** (recommended) option for **Spare Area Usage** setting to ensure that your ECC control words will be programmed into spare area.

It is necessary to specify **ECC frame size** (in bytes) and **Acceptable number of errors**. This number of errors should be equal to number of single-bit errors that your ECC algorithm is capable to correct. Using settings from *Figure 17*, the ECC algorithm capable to correct 4 single-bit errors in frame of 512 bytes is used (according to MLC requirements).

Finally, you must enable the feature by checking the checkbox **Use Tolerant Verify feature**.

Block Protection

Some NAND flash devices support a permanent block protection mechanism. Since NAND flash market is intensively growing-up, one can expect there will be more protection mechanisms in the future. Usually, they have no special names. Therefore there is a list of devices that uses concrete mechanism specified for each one protection mechanism.

Block Protection mechanism 1

Known devices using it:

Samsung: K5D1257ACC, K5D1213ACE, K5D12121CA

Note:

This device list is very probably not complete. Please, compare your device datasheet with description below.

Description:

The block protection status is written into special area in the block (please, distinguish this special area from spare area). The blocks can be prohibited from:

– **programming** – if enabled, the block cannot be programmed. However, the block still can be erased in usual way, erasing also the special area of the block. In this way, the programming protection is disabled.

– **erasing** – if enabled, the block cannot be erased ever more. The same applies also for special area of the block. In this way, once enabled, erase protection cannot be disabled. If both, programming and erasing protections are enabled, they cannot be disabled (since erasing is not allowed).

Commands used by Samsung:

41h – programming is prohibited

42h – erasing is prohibited

43h – both, programming and erasing is prohibited

7Ah – read block protection status

How the block protection is handled by Dataman's programmers:

There are **Block Protection Settings** in **<Alt+S>** dialog (see *Figure 18*). The list of blocks for both kinds of protections can be specified there, using following syntax:

– all blocks: 'all'

– none block (default setting): 'none'

– individual blocks 1, 4, 7 and 9: '1, 4, 7, 9'

– continuous area from block 0 up to block 4: '0-4' (the same as '0, 1, 2, 3, 4')

– various combinations of individual and continuous blocks can be used, e.g.: '0-7, 16-23, 31, 35'

– the spaces are ignored: '1 , 4' = '1,4', also '1 - 4' = '1-4'

– the commas are mandatory, see all examples above

Figure 18: Block Protection Settings

Blank operation:

If there are any blocks prohibited from **programming** in device, they can be erased and allowed for programming again. Therefore they are included into blank test.

If there are any blocks prohibited from **erasing** (or both, programming and erasing) in device, they cannot be erased and written again. Therefore they are treated like being invalid and skipped during blank test.

Read operation:

All blocks are read, not regarding the blocks protection status. Also, the blocks protection status is read and transformed into the lists in **<Alt+S>** dialog. In this way, they will be automatically saved into project file if you want to produce the one from master-chip. Remember, read operation can modify your own settings of these lists.

Verify operation:

Also the blocks protection status is verified. All differences from actual lists are reported and verify error is displayed.

Program operation:

The blocks protection is programmed after programming the data area. Only the blocks specified in the lists are affected in device, i.e. none block management method is used for protections.

If there are any protected blocks in device, they will be recognized during invalid block(s) table building and treated as being invalid. However, the blocks prohibited from programming only can be erased by allowing **Erase before programming** in **<Alt+O>** dialog. In this way, the programming protection will be disabled and the blocks can be used for programming again. This feature is intended for evaluation purposes and hobbyists to allow to use the device repeatedly, not for mass-production.

Erase operation:

If there are any blocks prohibited from ***programming*** in device, they will be erased automatically. In this way, programming protection will be disabled and the blocks can be used for programming again.

If there are any blocks prohibited from ***erasing*** (or both, programming and erasing) in device, they will be recognized during invalid block(s) table building and treated as being invalid (i.e. not erased).

The lists of protected blocks from **<Alt+S>** dialog don't affect the erase operation in any way.

One Time Protect

This feature is the special property of Spansion's S30MS-R MirrorBit® ORNAND™ devices family. See appropriate device datasheet for more information about **One Time Protect**. The Feature is supported from pg4uw version 2.50.

Figure 19: One Time Protect Area settings

Check **Process One Time Protect Area** checkbox if you want to treat the One Time Protect block. It is possible to set the write protection for individual pages and One Time Protect block default overlaying mode (Overlay/Removed).

Blank operation:

During **blank test** operation, also the pages in One Time Protect block are checked.

Read operation:

During **read** operation, also the pages in One Time Protect block are read, as well as protection and default overlaying status. After the operation completion, the write protection of individual pages can be seen in **List of pages that should be protected** edit box. The overlaying status is displayed in **One Time Protect Area default mode** scroll-down menu. In this way, the *read* operation affects your original **<Alt+S>** settings. The data belonging to One Time Protect pages are placed at the end of buffer, starting from address that is equal to device size (including spare area size).

Verify operation:

During **verify** operation, also the pages in One Time Protect block are verified, as well as the write protection status of individual pages and default overlaying mode status. The original data are fetched from the end of buffer or **<Alt+S>** settings respectively, as mentioned in the paragraph for *read* operation.

Program operation:

During **program** operation, firstly the protection status of the One Time Protect pages is checked. If there is any protected page, the program operation is canceled (the selective programming is not supported). Then, individual One Time Protect pages are programmed. The spare area usage is common for both, main data and One Time Protect area (the setting does not affect the One Time Protect area data start in buffer). Finally, the page protection and default overlaying mode is programmed.

Erase operation:

During **erase** operation, firstly the protection status of the One Time Protect pages is checked. If there is any protected page, the device is erased except of One Time Protect block (it is not possible to erase the block with protected page). Otherwise, also One Time Protect area is erased.

Remember: write protection and default overlaying mode are not erasable.

Leaving **Process One Time Protect Area** checkbox unchecked, the One Time Protect block will be still temporarily removed when processing the device, to assure correct processing of block No. 0.

Buffer data vs. device data mapping

It is very important to understand, how data in programming buffer are organized and how to set-up the programming process as efficiently as possible. Before explanation, take a look on following picture.
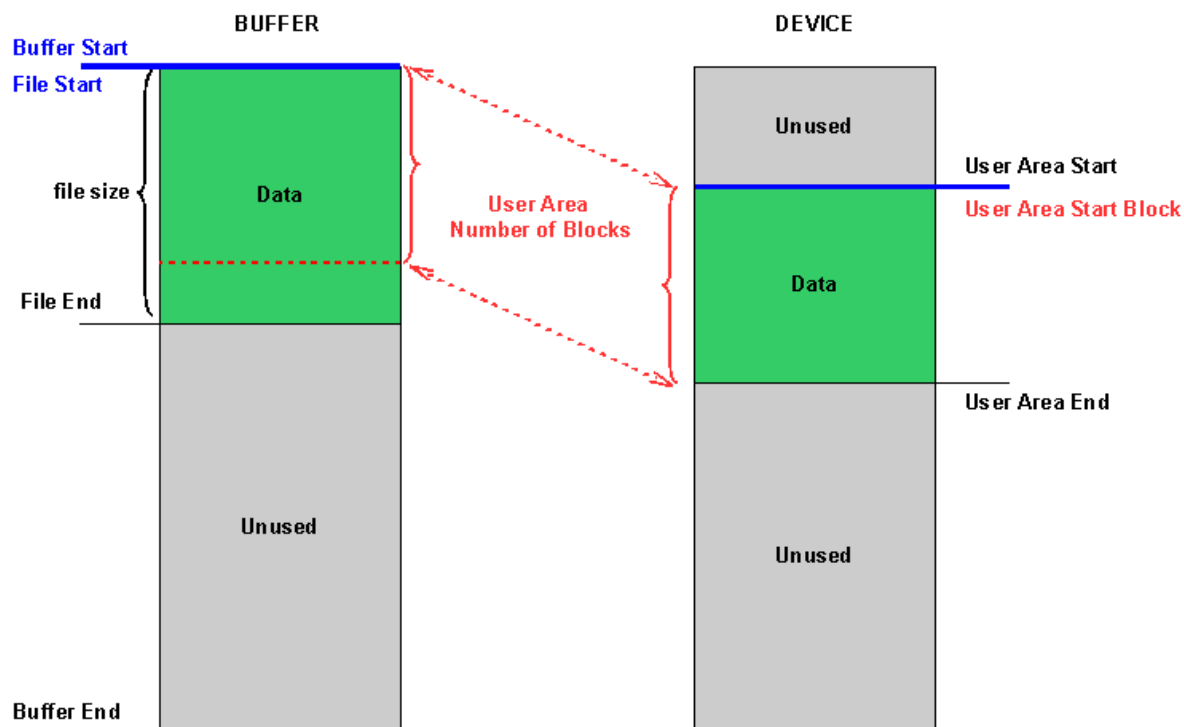
Figure 20: Buffer data vs. device data mapping

The file is loaded into buffer always from beginning (start address is $00000000), so the buffer start and file start are always the same. This address is later mapped into User Area Start. Note, you can place your data anywhere in your device range. The **User Area – Start Block** setting defines, where data will be placed during programming operation. You are allowed to specify starting block instead of starting address, because the block is basic entity in NAND Flash memory devices. See FAQ section (page 36) for address to block number convert method.

Figure 21: More efficient settings recommendation example

Starting from buffer/file start you can determine the amount of data that will become your User Area. This amount of data is expressed in quantity of blocks, again. The **User Area – Number of Blocks** setting is dedicated to this purpose. Note, it is not mandatory that User Area size must be equal to file size. It is on your choice how many data blocks will be treated with. Our software will display a recommendation in case of more efficient setting is possible, see *Figure 21* (feature available from SW version 2.29). However, our software in no way can predict, what do you want to do. You can accept or decline its recommendations. See FAQ section (page 36) for data size to blocks count convert method.

Note:

If you have only small piece of data that have to be placed at the end of device range, you are not forced to program large data file. You can rearrange the data file in more efficient way. All you need to do is following:
– cut off blank area, keeping block alignment
– set **User Area – Start Block** using count of discarded blocks
set **User Area – Number of Blocks** appropriate to valuable data size.

Similarly, if you have small piece of valuable data at the beginning of device range, you can cut off blank area from the end of data file or choose **User Area – Number of Blocks** to be equal just to valuable data size instead of file size. In this case you are in need of discarding the recommendation, since it is simply based on loaded file size, not on valuable data size.

This way only small portion of device range will be treated, just that one containing your data.

Using this practice, invalid blocks preceding your valuable data do not impact their beginning. I.e. if you have specified **User Area – Start Block** to be equal $400, they will start at block No. $400. Having blank area included in your data file and e.g. 3 invalid blocks before block No. $400, your valuable data will be shifted out 3 blocks; they will start at block No. $403.

If you are production engineer, consult the possibility of increasing programming throughput mentioned here with appropriate development department.

Automatic operation feature

There is automatic operation feature implemented in our universal and production device programmers. There is possibility to tight together related operations. In consequence, you are able to start multi-operation (e.g. **Read + Verify**) with just single button click. See User manual of your programmer for more information. Use **Device Operation Options** (shortcut **<Alt+O>**) to define this feature.

Figure 22: Device operation options

As shown on *Figure 22*, there are following settings available:

Insertion Test

Test of proper device insertion. See your product documentation for more details (there are no special NAND Flash considerations).

Device ID check

Check of device ID. See your product documentation for more details (there are no special NAND Flash considerations).

Erase before programming

Having this option enabled, the device will be erased before programming will take place. NAND Flash device manufacturers identically say that they stock the chips in erased (blank) state. So, essentially, there is no need for device erasing if it is first time used.

Our device programmers always perform erase operation on whole chip, not regarding User Area settings. The reason is simple – it is unfeasible to predict number of blocks that will be really used, as individual blocks can indeterminably turn into invalid state.

Note:

It takes only few seconds to erase whole chip (e.g. approx. 4 seconds on 256Mbit device). Physically, erasing is performed by on-chip controller on block basis. The controller tries to erase the block and subsequently performs operation validation (this repeats several times in case of failure) . Error status messages are processed by our software. Blocks, where erase operation fails, are marked to be invalid. Blocks, where erase operation succeeds, are blank. So consequent blank check operation is not necessary and may be omitted (saving approx. 39 seconds on 256Mbit device). This in truth occurs, if you enable both, erase and blank check before programming.

Blank check before programming

Having this option enabled, the device will be blank-checked before programming will take place. NAND Flash device manufacturers identically say that they stock the chips in erased (blank) state. So, essentially, there is no need for device blank check if it is first time used.

Our device programmers always perform blank check operation on whole chip, not regarding User Area settings. The reason is simple – it is unfeasible to predict number of blocks that will be really used, as individual blocks can indeterminably turn into invalid state.

Blank check is also performed as consistent element of erase operation. See note on page 32 for information on how to save operation time.

Verify after reading

Having this option enabled, the device will be verified after reading will take place. See your product documentation for more details (there are no special NAND Flash considerations).

Verify after programming

Having this option enabled, the device will be verified after programming will take place. Only the area marked as User Area will be verified. See note on page 31 for information on how to save verification operation time.

From SW version 2.29 new automatic feature was added. If verification operation is performed as a part of automatic programming, blank areas are neither programmed nor verified. In rare cases programming of one page can impact another page within the block (it is the particularity of NAND Flash technology). There is some possibility this appears when you have small portion of block being written with valuable data and the rest of block remains blank. The damage in blank portion of block will be not detected in this way, since it will be skipped. The probability of happening so is marginal. On that account we can disregard this. However, new feature does not appear neither in stand-alone verify operation, nor having **Quick Program** checkbox unchecked (thus having Quick Programming disabled).

Error Messages

Configuration error messages

Configuration error: Incorrect input: XXX : YYY

Value YYY entered in field XXX is irregular. Check typing errors – irregular characters, missing hexadecimal radix prefix (use dollar sign - $), etc... Check logical errors – negative values, zero ranges, etc...

Configuration error: Device range exceeded.

Some area definition exceeds device capacity. Check (recalculate) start blocks and / or numbers of blocks defined for User Area, Solid Area and RBA Area.

Configuration error: Attempt to access implicitly booked area.

Some area definition interferes with area, that is internally reserved. Read the appropriate paragraph about IB management method used and check the settings. E.g. your User Area definition starts from the 0-th block, while you are using **Skip IB with Map in 0-th Block**.

Configuration error: Solid area must belong to User Area.

Blocks assigned to Solid Area must be a subset of blocks assigned to User Area. Check (recalculate) margins of User Area and Solid Area.

Configuration error: RBA must be behind of User area.

RBA Area must be defined behind the User Area. Check (recalculate) the end of User Area and beginning of RBA Area.

Configuration error: There must be some free space behind the User area for IB replacement.

RBA Area cannot follow User Area immediately. There must be some free space (at least one block) for block reservoir. Remember, that invalid blocks from User Area are replaced with good blocks from block reservoir. Check (recalculate) the end of User Area and beginning of RBA Area.

Configuration error: RBA needs 3 blocks at least.

The reason for this was explained in previous (see page 19). Define more blocks for RBA table (at least three blocks).

Programming error messages

There is not enough valid blocks.

The first action always performed is building the IB map. Simultaneously, recognized invalid blocks are counted. If this message appears, there are too many invalid blocks and your User Area cannot be fitted into decreased chip capacity. You cannot program your data image into that chip.

There is invalid block inside of Solid area.

The first action always performed is building the IB map. If this message appears, your Solid Area requirements cannot be fulfilled due to invalid block occurrence. You cannot program your data image into that chip.

There is invalid block inside of implicitly booked area.

The first action always performed is building the IB map. If this message appears, it is impossible to use internally reserved block(s) due to invalid block occurrence. E.g. the 0-th block is invalid while you are using **Skip IB with Map in 0-th Block**. You cannot program your data image into that chip.

Try again. If problem remains, your device may be broken.

There is certain hardware problem in your programming system. The chip does not respond to programmer actions. Check thoroughly pin contacts on your ZIF, adapter (if it is used) and memory chip and try to repeat the action. If it is still unsuccessful, try another chip. Probably, this one is broken.

If you get this message often and random-like, check **Device info** (click **See also Device info <Ctrl+F1>!** link in the bottom of your Dataman program or press a shortcut **<Ctrl+F1>**) to read how to connect auxiliary capacitor to your device.

ECC error messages

ECC control words error found.

An error in ECC control words (checksum) was found. This is critical error, your data are no longer reliable.

Uncoverable ECC error(s) found.

Double bit error(s) was detected in your data. ECC algorithm is not capable to recover it. This is critical error, your data are no longer reliable.

Repairable ECC error(s) found and restored.

Simple bit error(s) was detected in your data. ECC algorithm was capable to recover it. Data loaded into buffer are repaired (but not data in your chip). You can use them and make a good copy of your data image into another chip.

Uncoverable ECC error(s) in IB map.

If you use **Skip IB with Map in 0-th Block** in conjunction with ECC, IB map is covered with ECC, too. If this message appears, IB map in 0-th block is not further reliable.

ECC errors were found...

A new ECC control algorithm is used from SW version 2.45 for some devices. All errors are displayed continuously as they are found. A message above is final confirmation message.

RBA error messages

Invalid RBA table.

Your RBA table cannot be used. There is an error in both copies of RBA table (blocks used are invalid, or there are uncoverable ECC errors). Make sure your RBA Area setting is correct.

Common error messages

Required number of blocks not processed due to IB occurrence.
  - During programming: It is impossible to program required number of blocks due to excessive acquired invalid blocks occurrence. The programming is aborted immediately after detecting the condition.
  - During reading / verifying: It is impossible to read / verify required number of blocks due to excessive inherent invalid blocks occurrence. As many blocks are loaded into buffer as possible. If **Verify after reading** is enabled (see **Device Operation Options** – use shortcut **<Alt+O>** in your Dataman program), verify operation will be aborted.

It is impossible to satisfy Solid Area requirements.

There is some invalid block(s), that disturbs Solid Area (read more about Solid Area considerations on page 17). Reading / verifying process will be completed regarding selected method and User Area and Solid Area settings. This message informs you, that data image in buffer is not reliable.

Frequently asked questions

1. How to convert address into block number?

You need information about your device internal organization (see your device datasheet or our *Device Info* – shortcut *<Ctrl+F1>*) and programming method.

Example:
Address to be converted: ADR = $00005231

Device organization of K9F5608U0D:
Page size: PS = 512 bytes (+ 16 bytes are for spare area)
Number of pages per block: PN = 32

Programming method:
Invalid Block Management: Skip IB (Invalid Blocks) – *not important for calculation*
Spare Area Usage: User Data – *important for calculation*

Algorithm:

1. Calculate your page size. In case of your Spare Area Usage method of *User Data* or *User Data with IB Info Forced* you need to consider also spare area bytes. So used page size in this example will be:
   PS1 = 512 + 16 = 528 bytes
2. Calculate block size BS:
   BS = PS1 x PN = 528 x 32 = 16896
3. Convert it into hexadecimal format:
   BS1 = $4200
4. Divide your address ADR by block size BS1, round to nearest smaller integer:
   BN = floor(ADR / BS1) = floor($00005231 / $4200) = $1

That is all. Our example starting address belongs to block No. $1. You can enter this number into *User Area – Start Block* edit filed directly ($1) or in decimal form (1).

Note:
Your calculator may cut-off decimal fraction in hexadecimal mode. In this case, you obtain properly rounded result directly.

2. How to convert data size into blocks count?

You need information about your device internal organization (see your device datasheet or our *Device Info* – shortcut *<Ctrl+F1>*) and programming method.

Example:
Data size to be converted: DS = $00FFFFFF

Device organization of K9F5608U0D:
Page size: PS = 512 bytes (+ 16 bytes are for spare area)
Number of pages per block: PN = 32

Programming method:
Invalid Block Management: Skip IB (Invalid Blocks) – *not important for calculation*
Spare Area Usage: User Data – *important for calculation*

Algorithm:

2. Calculate your page size. In case of your Spare Area Usage method of **User Data** or **User Data with IB Info Forced** you need to consider also spare area bytes. So used page size in this example will be:
$PS1 = 512 + 16 = 528$ bytes
2. Calculate block size BS:
$BS = PS1 \times PN = 528 \times 32 = 16896$
3. Convert it into hexadecimal format:
$BS1 = \$4200$
4. Divide your data size DS by block size BS1, round to nearest greater integer:
$BC = ceil(DS / BS1) = ceil(\$00FFFFFF / \$4200) = \$3E1$

That is all. Our example data size can squeeze into $3E1 blocks. You can enter this number into **User Area – Number of Blocks** edit field directly ($3E1) or in decimal form (993).

Note:

If you know only start address (ADRstart) and end address (ADRend) of your data, you can compute data size (DS) using following formula:
$DS = ADRend – ADRstart.$

Your calculator may cut-off decimal fraction in hexadecimal mode. In this case you need to add $1 to the result from your calculator display.

3. Erase before programming – yes or not?

As it was mentioned in note on page 32, it is not necessary to use this feature if you use your memory chip for the first time. Certainly, you must erase chip that was programmed formerly.

4. Blank check before programming – yes or not?

As in previous question, there is not necessity to perform this if you use your memory chip for the first time. Similarly, there is no need to check just now erased device for blank, since this check is performed internally by on-chip controller.

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

Yes, it is OK. Our software detects file margins. If it recommends to use larger area, it is because the data between file start and file end takes more space than you have defined using **User Area – Number of Blocks**. If you want to process only the part you have selected, cancel the recommendation. Accept the recommendation otherwise.

6. I want to make exact copy of the chip with unknown content. What settings should I use?

The chip-copy operation can be subdivided into two steps – reading original device and programming target device.

Step 1 – reading original device:
Several facts must be considered:
– *invalid block management scheme* – since the scheme is unknown, the only surely useful option for **Invalid Block Management** setting is **Do not Use**. This way, complete original device will be read, valid blocks as well as invalid blocks.

– *spare area* – since spare area assignment is also unknown, the only surely useful option for **Spare Area Usage** setting is **User Data**. This way, complete pages will be read, data area as well as spare area, without any interventions.

– *user area size* – since data position in range of device is unknown, user area must cover whole device. Therefore **User Area – Start Block** must be **0** and **User Area – Number of Blocks** must be equal to the number of blocks in device.

Using these settings, complete device will be read, byte-by-byte. This mode is sometimes called "RAW mode".

Step 2 – programming target device:
All previous statements are valid, except of one:

– *invalid block management scheme* – the successfulness of programming operation can be impaired by invalid block(s) occurrence. Therefore, target device must be free of invalid blocks. You can quickly exclude the chip with invalid block(s) using **Invalid Block Management** option of **Skip IB (Invalid Blocks)** and let the algorithm preserve whole chip to be free of invalid blocks having **Guard Solid Area** checked, with **Solid Area – Start Block** equal to **0** and **Solid Area – Number of Blocks** equal to the number of blocks in device.

Using these settings, solid area will be created equal to user area (covering whole chip). Since skip invalid block method is involved, invalid block(s) map will be created firstly. Afterwards, solid area (i.e. whole chip for now) will be checked for presence of invalid blocks. If any invalid block will be found, the operation will be terminated immediately with "There is not enough valid blocks." message displayed. Such chip cannot be used for byte-by-byte copy creation. Try another one.

7. After programming, the verification fails. It seems like there is "address shift" in device compared to buffer. What should I have to do?

The problem is actually observed on several devices manufactured by Hynix, STM and/or Intel. Verification errors report is similar to the following one:

```
L0076: Verify - error(s)
L0077:
L0078: ADDRESS  DEVICE BUFFER | ADDRESS  DEVICE BUFFER | ADDRESS  DEVICE BUFFER
L0079: ----------------------|-----------------------|-----------------------
L0080: 000077B  73    AF  | 000078F  24    3C  | 00007AA  00    05
L0081: 000077C  C8    73  | 0000790  08    44  | 00007AB  00    40
L0082: 000077D  02    C8  | 0000791  25    24  | 00007B3  40    00
L0083: 000077E  0C    02  | 0000793  00    25  | 00007B4  06    00
L0084: 000077F  00    0C  | 0000794  00    08  | 00007B6  00    06
L0085: 0000785  BF    00  | 0000795  01    00  | 00007B7  00    40
L0086: 0000786  04    BF  | 0000797  00    01  | 00007BE  FC    00
L0087: 0000787  00    8F  | 000079B  68    00  | 00007BF  FF    00
L0088: 0000788  BD    04  | 000079C  04    00  | 00007C0  BD    FC
L0089: 0000789  27    00  | 000079D  40    68  | 00007C1  27    FF
L0090: 000078A  0B    BD  | 000079E  00    04  | 00007C2  00    BD
L0091: 000078B  80    27  | 000079F  00    40  | 00007C3  00    27
L0092: 000078C  08    0B  | 00007A7  70    00  | 00007C4  BF    00
L0093: 000078D  3C    80  | 00007A8  05    00  | 00007C5  BF    00
L0094: 000078E  44    08  | 00007A9  40    70  | 00007C6  C8    BF
```

Device content appears to be shifted by one or more bytes in respect to buffer content. Some bytes are missing or doubled, altering the seeming shift. Usually, the device works fine in target application. Using 448Pro, 448Pro+, 448Pro2 multiprogrammer, the problem is more frequent on sites #1 and #3.

Cause:
Data reading is sequential operation driven by toggling nRE pin. The data is transferred from I/O register to the pins on falling edge of the pulse. In the same time the device address counter is incremented. If some oscillation/overshoot occurs on nRE pin, it can cause false pulse recognition and void data access.

Elimination:
If such behavior is observed, it is recommended to connect multilayer ceramic capacitor between pins 8 (nRE) and 13 (Vss). Usually, the sufficient capacity is 180-220pF.

8. I can see plenty of invalid blocks in my device (using blank test, read and/or verify operation). Is it normal?

Probably, you are working with the device that have some data written in spare area and this data is not-compliant with manufacturer specific invalid block labeling scheme. Try options **Do not Use** and **User Data** for **Invalid Block Management** or **Spare Area Usage** settings, respectively.

If you have device programmed by yourself using **User Data** option for **Spare Area Usage** setting and there are some invalid block(s) in your device, the stand-alone verification operation will probably always fail, depending on values at block validity information position.

9. What invalid block labeling scheme does your programmer use?

Our programmers always adheres to manufacturers specifications. Please, see your device datasheet for further informations regarding to block validity labeling (identifying initial invalid blocks).

10. I want to write multiply data files into single memory chip. How can I proceed using your programmer?

Example:
We have three files, each defined by its size, start position in device (in terms of blocks) and some special requirements:

File #1: bootloader
Start Block: 0
Size: 2
Specials: must be placed in blocks no. 0 and 1, cannot be interrupted by invalid block(s)

File #2: system
Start Block: 2
Size: 250
Specials: skip invalid blocks

File #3: initial data
Start Block: 500
Size: 100
Specials: skip invalid blocks

Solution:
(A) File #3 start block is "absolute", i.e. File #3 must start at block 500 at every conditions.
The procedure must be splitted into two steps.

Step 1:
File #1 must be placed in blocks no. 0 and no. 1. Therefore **Guard Solid Area** feature must be involved, with **Solid Area – Start Block** equal to 0 and **Solid Area – Number of Blocks** equal to 2, while **Skip IB (Invalid Blocks)** management method is used. File #2 continues immediately from block no. 2 using **Skip IB (Invalid Blocks)** management method again. Therefore this two files can be processed at once.
Settings for step 1:
Invalid Block Management: Skip IB (Invalid Blocks)
Spare Area Usage: - not concerned by this question

User Area – Start Block: 0
User Area – Number of Blocks: 252 (2 + 250)

Guard Solid Area – checked
Solid Area – Start Block: 0
Solid Area – Number of Blocks: 2

See next question for more informations on how to load multiply files into buffer.

Step 2:

Remaining File #3 uses simple **Skip IB (Invalid Blocks)** management method with plainly defined user area requirements.

Settings for step 2:

Invalid Block Management: Skip IB (Invalid Blocks)
Spare Area Usage: - not concerned by this question

User Area – Start Block: 500
User Area – Number of Blocks: 100

Guard Solid Area – unchecked (needless)

Check also your **<Alt+O>** settings. Enabling **Erase before programming** feature is prohibited now, since it erases previously programmed files #1 and #2!

(B) File #3 start block is "relative", i.e. it should be shifted by number of previously skipped invalid blocks.

Using **Skip IB (Invalid Blocks)** management method, if invalid block is found, actual and all following blocks are shifted by one block. So, if there are N invalid blocks found until block no. 500, block no. 500 will be automatically shifted by number of N blocks. Therefore, all three files can be processed at once using following settings:

Invalid Block Management: Skip IB (Invalid Blocks)
Spare Area Usage: - not concerned by this question

User Area – Start Block: 0
User Area – Number of Blocks: 600 (there are 500 blocks before File #3 which is 100 blocks large)

Guard Solid Area – checked
Solid Area – Start Block: 0
Solid Area – Number of Blocks: 2

See next question for more informations on how to load multiply files into buffer.

11. How can I load multiply files into your SW buffer?

Example:
We have two files, that have been placed in single device:

File #1: bootloader
Start Block: 0

File #2: system
Start Block: 2

Device used: K9F5608U0D
Page size: PS = 512 bytes (+ 16 bytes are for spare area)
Number of pages per block: PN = 32

Spare area is in use. Note: this feature must be unified for all files!

Solution:
There are two settings available at the bottom of **Load File** dialog – **Buffer offset for loading** and **Erase buffer before loading**.

Keep **Erase buffer before loading** unchecked (not enabled) for this procedure, otherwise loading of File #2 will erase previously loaded File #1. If you need to erase buffer at beginning of loading, use manual buffer erase – shortcut **<Ctrl+F2>**, erase value is FFh or FFFFh, depending on device organization (8-bit or 16-bit wide, respectively).

Use **Buffer offset for loading** to specify the file offset in buffer. The offset will be equal to 0 for File #1, since it starts from block no. 0.

How to compute buffer offset for File #2:

1. Compute block size:

Since spare area is used, the page size PS = 512 + 16 = 528 bytes.

Block size BS = PS x PN = 528 x 32 = 16 896 bytes

2. Compute offset:

Offset is equal to start address of file start block: OFFSET = BS x Start_block = 16 896 x 2 = 33 792.

3. Convert computed offset to hexadecimal format:

OFFSETh = 8400h

and fill this number into **Buffer offset for loading** box.

Don't worry about invalid blocks at this stage. You must create an "ideal device image". Invalid blocks will be treated by invalid blocks management scheme on-the-fly during programming operation.

12. I have "produced" lots of invalid blocks during my experiments with my nand flash device. Is it possible to repair them?

Invalid block(s) identification procedure is based on invalid block labeling scheme. This uses spare area to store block validity information. It is very easy to overwrite this information using **User Data** option for **Spare Area Usage** setting.

Remember: **There is a risk of initial block validity information loss!**

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

Invalid Block Management: Do not Use

Erase operation always erases all blocks in device. If initial invalid blocks were disconnected from the line by manufacturer, they will be recognized again. But if they were just labeled in spare area, initial block validity information will be lost.

13. I have strict requirements regarding to the number of invalid blocks and want to discard the devices with too much invalid blocks. What setting should I use?

Some applications may require to not use the devices if the number of invalid blocks is larger than some predefined level. E.g. the device cannot be used, if it contains more than 5 invalid blocks.

How to proceed:

Our control software uses logic, that is inverse to the above stated problem - it guards the number of valid (good) blocks. Therefore, also the problem must be interpreted in inverse manner. The number of blocks that are allowed to be invalid must be converted into the number of blocks, that must be valid.

Example:

There are 1024 blocks in device. The device should be discarded, if there are more than 5 invalid blocks. It means, the device should be processed, if there are at least (1024-5=) 1019 valid blocks. These blocks should be treated as **User Area – Number of Blocks**.

By default, **User Area – Number of Blocks** value is equal to total number of blocks in device minus 2% for invalid blocks (level of invalid blocks accepted by manufacturer for new device). E.g. it is equal to 1004 for device having 1024 blocks. You need to edit this value to the new one, that is computed following the example above.

Usually, the data covers only few blocks. After loading the file and clicking Program button, the software will offer to work with only this small portion of the chip. You must ignore this offer and continue working with user area size computed following the example above.

The first step performed is building the invalid blocks table. During this operation, the number of available valid blocks is computed and compared with the value of **User Area – Number of Blocks**. If there are

not enough valid blocks – or in other words, if there are too many invalid blocks - the operation is aborted and error message is displayed.

There is one restriction using this approach: blocks are processed starting from **User Area – Start Block** and all rules are applied from this block until the end of device. So, in order to expand the rule to whole device, **User Area – Start Block** must be equal to 0. If your data doesn't start from block no. 0, you can create an offset by inserting FFh area with appropriate size before the data. This should be done at data preparation stage. Don't worry about large area of FFh bytes (blank pages) before or after data since they are recognized and skipped by software automatically, not involving any significant time consumption.

14. I can see a plenty of invalid blocks while reading (or verifying) my nand flash device. The device is not read correctly. What am I doing wrong?

This situation typically occurs with **Invalid Block Management Method** set to **Skip IB (Invalid Blocks)**. Usually, the messages like follows are displayed (depending on other settings):

```
L1899: Reading device ...
L1900: There are not enough valid blocks.
L1901: It is impossible to satisfy Solid Area requirements.
L1902: Access with selected device - error!
L1903: Required number of blocks not processed due to IB occurrence.
```

The problem explanation:

The vendor specifies the method of invalid blocks labeling for each device. The rule can be found in device datasheet. There is not the one and only universal rule. Several examples follow:

– non-FFh value at column address 517 of the first and second page in a block (for SLC devices with 512-bytes per page)
– non-FFh value at column address 2048 of the first and second page in a block (for SLC devices with 2048-bytes per page)
– non-FFh value at column address 2048 of the last page in a block (for MLC devices with 2048-bytes per page)

Our programmer expects this label at specified place. One can see, the addresses specified above fall into spare area. Therefore, if spare area is used, the labeling scheme prescribed by vendor can be overwritten by user data. Often there is some kind of user specific invalid block labeling scheme, that come from used file-system. If our programmer doesn't find the correct label (i.e. FFh value) at specified place, it will declare the block being invalid.

The work-around:

This simple principle can be used for reading the complete device memory image, including potential invalid blocks and spare area. Further analysis must be done over the read data to identify the file system and invalid block labeling scheme. The settings stated below can be simply used for making a 1:1 device copy. However, the device that is free of invalid blocks must be used in place of the target one.

Access Method (shortcut **<Alt+S>**) settings for reading the complete image of source device:

Invalid Block Management: Do not Use
Spare Area Usage: User Data

User Area – Start Block: 0
User Area – Number of Blocks: the total number of the blocks in device (can be found in device datasheet or **Device Info** (shortcut **<Ctrl+F1>**) in our software)

All others setting should be left as per default.

Conclusion

The NAND Flash memories are suitable for using in applications where a large amount of data has to be stored in memory, in other words, where it is desired to use a "file" system for stored data. Due to specific characteristics of NAND Flash architecture, there is need to implement the invalid block management system, ECC coding, and eventually requested file system to the target device. In most cases an embedded memory must contain data in requested format before assembling in production (e.g. Boot data, pre-formated file system) with selected invalid block management method and ECC coding. These data has to be programmed into NAND Flash memory using the device programmer.

References

[1] M-System Inc.: Two Technologies Compared: NOR vs. NAND, Rev. 1.1, July 2003

[2] Toshiba America Electronic Components, Inc: NAND Flash Application Design Guide, Revision 1.0, April 2003

[3] Toshiba America Electronic Components, Inc, What is NAND Flash Memory, March 2003

[4] Samsung Semiconductor, Inc.: Selecting the Right FLASH Partner to Turn Technology Advantages into Profits, 2003

[5] Samsung Semiconductor, Inc.: NAND Flash Spare Area Assignment Standard, April 2005

[6] Micron Technology, Inc.: Small Block vs. Large Block NAND Flash Devices, Technical Note TN2907, February 2006

Version history

Version 2.0 – July 2006
- title changed (formerly: NAND Flash Memories)
- term "bad block" converted to "invalid block" (in concordance with Dataman software)
- terms "single-level cell" and "multi-level cell" introduced
- terms "small page" and "large page" introduced
- NAND Flash memory array description figure changed
- extended features list added
- invalid block map building algorithm flowchart figure added
- skip invalid block method figure changed
- RBA method description improved, figure changed
- ECC description improved
- Programming NAND Flash Memories Using Dataman Device Programmers chapter added

Version 2.01 – September 2006
- Buffer data vs. device data mapping section added
- Automatic operation feature notes added
- FAQ section added

Version 2.02 – March 2007
- Invalid Block Indication Options section added

Version 2.03 – May 2007
- Tolerant Verification section added

Version 2.04 – June 2007
- New questions/answers (6-12) added in FAQ section.

Version 2.05 – September 2007
- New question/answer (13) added in FAQ section.

Version 2.06 – December 2007
- Links to external web-pages actualized
- Minor text updates

Version 2.07 – March 2008
- Block Protection section added
- Error Messages section actualized

Version 2.08 – April 2008
- New question/answer (14) added in FAQ section.
- Block Protection section actualized

Version 2.09 – June 2008
- One Time Protect section added

Version 2.10 – August 2008
- Reserved Block Area method enhanced