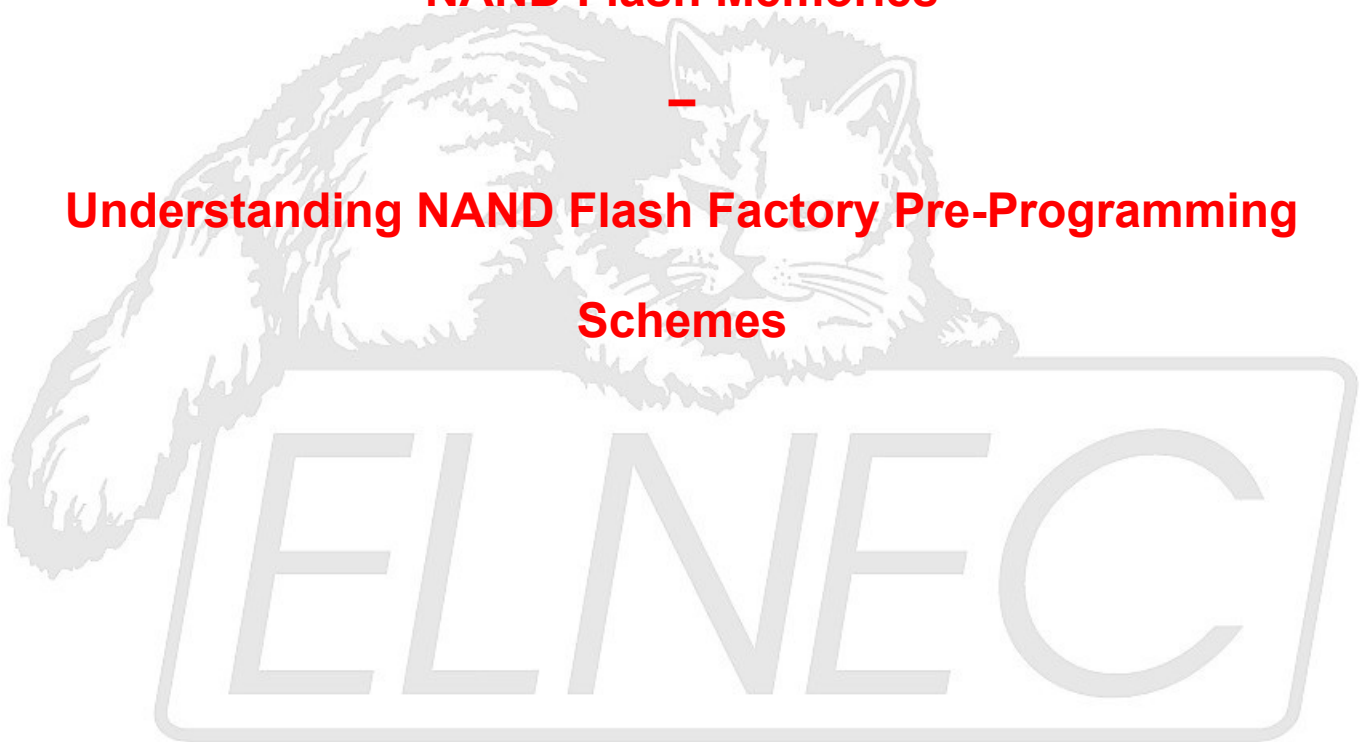


## **NAND Flash Memories**

# **Understanding NAND Flash Factory Pre-Programming Schemes**



**Application Note**

February 2009  
an\_elnec\_nand\_schemes, version 1.00

NAND flash technology enables memory manufacturers to produce memory devices with high density at low cost. From that reason, NAND flash memories are very popular in various electronic equipments that need to store a large amount of data, such as music players, cameras, mobile phones, PDAs and many others.

There are several important differences between (traditional) NOR and NAND flash memories, widely discussed through world-wide-web articles and forums. The most evident one is the presence of invalid blocks in NAND flash memory device. It means that not whole device address range can be used for data storage. Some memory locations are defective and cannot be programmed and read reliably. However, these defective locations don't affect the reliability of the rest of memory device.

There must various precautions been taken into account to cope with that defective locations, already in development phase of the target product. The set of such precautions can be called "pre-programming scheme". This application note tries to explain all aspects of NAND flash factory pre-programming. Please, read it carefully before you contact us as and ask for special support for your NAND flash project. It will help you to provide us with all information that we need for successful implementation, in exact and accurate form. This will help us to implement your needs in as short time as possible, that will further reduce your pre-production costs.

## A brief introduction to NAND flash

The flash memory was invented by Dr. Fujio Masuoka in 1984, when working for Toshiba. Intel was the first who has recognised massive potential of new technology and introduced the first commercial NOR type flash memory device in 1988.

NOR-based flash memory has long erase and write times, but has full address / data interface that allows random access to any memory location. These properties make it a convenient way for storage of a program code that doesn't need to be updated frequently, such as a hand-held device firmware or computer BIOS.

First NAND-based flash memory device was introduced by Samsung and Toshiba in 1989. Its I/O interface allows only sequential data access, however, the memory has faster erase and write times, higher density, lower cost-per-bit than NOR and ten times the lifetime. These properties make it suitable for mass-storage devices such as memory cards (for PCs, cameras, etc.) or actually boomed solid hard-discs.

Figure 1 compares the memory cells of NAND and NOR-based flash memories.

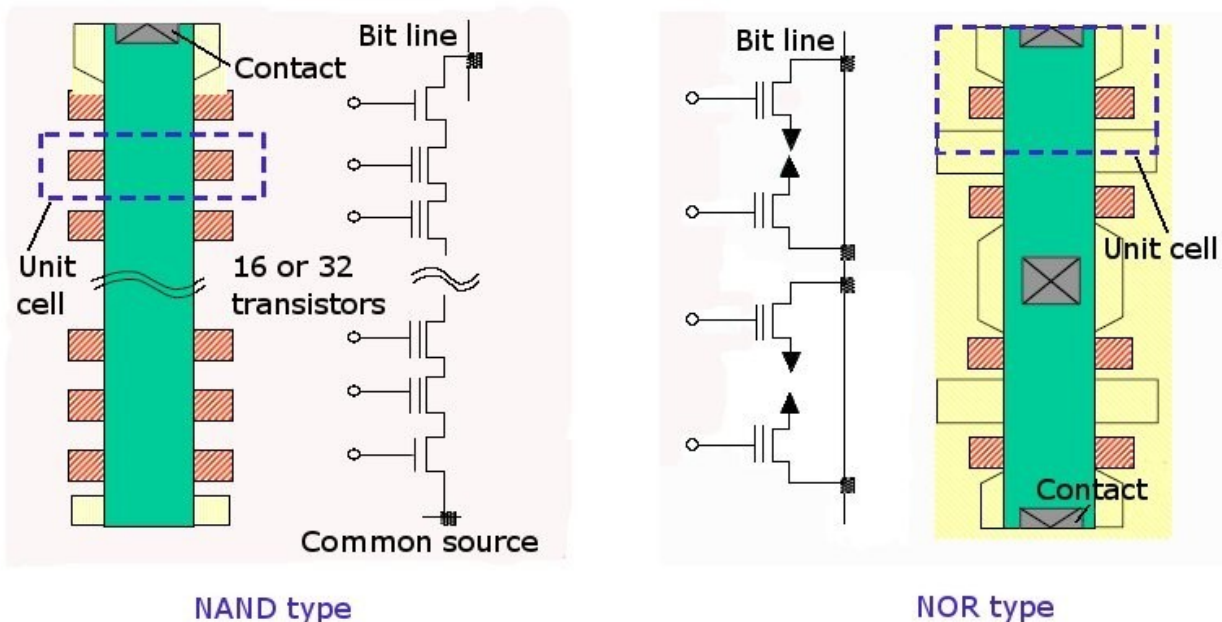


Figure 1. NAND versus NOR flash memory cell comparison (by Samsung)

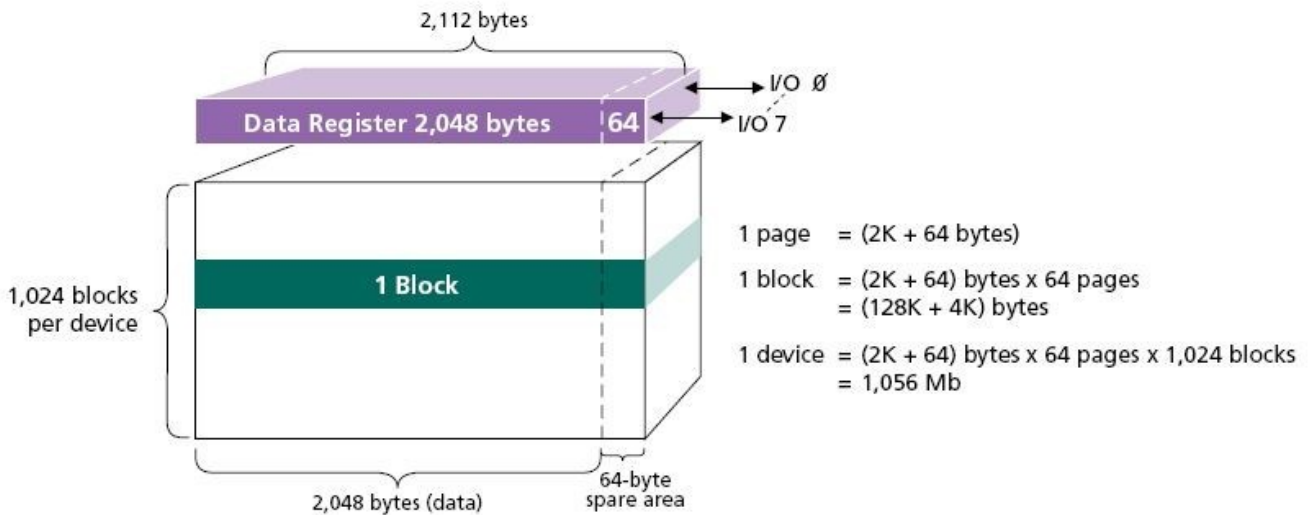


Figure 2. An 1-Gbit NAND flash memory architecture example (by Micron)

Figure 2 shows an internal organization scheme of real NAND flash memory device. The main attributes are as follows:

- Bytes are organized into pages. One page consists of data bytes area (typically multiple of 512) and spare bytes area (typically multiple of 16). Typical page configurations are 512+16, 2048+64 and 4096+128 bytes. Data area is used for storage of payload data (executable code, photos, MP3s, etc.), while spare area is used for memory management purposes (block validity labelling, operating system flags, error recovery data, etc.). The spare area is not included in device capacity and cannot be directly addressed. The page comprises the smallest programmable unit.
- Several subsequent pages (typically 32, 64, 128 or 256) comprise a block. The block represents the smallest erasable unit. If any defective bit is found, respective block is declared being invalid and excluded from further use.
- Some amount of blocks comprise a logical unit (LUN). The memory device can contain one or several LUNs (mainly modern NAND flash devices). LUNs improve memory performance by introducing some level of parallelism, but are not in concern during factory pre-programming.

The memory is accessed via data register. Some memory devices use also parallel cache register that improves data throughput by utilizing device busy time during internal data transfers.

Data register utilization:

- Read operation: After confirming the read command, internal data transfer from memory array to data register begins. During the transfer, the device is in busy state. After the transfer completes, the host can read the data from data register in sequential manner.
- Program operation: Firstly, the host must upload the data being programmed to data register. After confirming the program command, internal data transfer from data register to memory array

begins. During the transfer, the device is in busy state. After the transfer completes, the host can continue programming other page.

The errors can occur during both, programming as well as read operation. Typically, multi-level cell NAND flash devices are more predisposed to error generation due to more complicated cell structure with more vulnerable thresholds. Moreover, error in one memory cell can affect two or more pages. To workaround the problems with NAND flash reliability, the host typically uses:

- for program operation: various invalid block management techniques
- for read operations: various error detection and recovery mechanisms

An invalid block represents a special kind of NAND flash memory device error. Invalid block contains permanently defective bit location(-s) that cannot be reliably programmed / read. Invalid blocks can arise in two different ways:

- During the manufacturing process at factory (inherent invalid blocks). These invalid blocks are discovered during final tests. Typically, such blocks are disconnected from power line and cannot be accessed (not programmable, read as all 0x00). Often, they are just labelled being invalid using some invalid block labelling scheme specified by manufacturer (or by ONFI group - see <http://www.onfi.org>). Inherent invalid blocks must be recognised and treated during factory pre-programming process. It is necessary to define what should be done if invalid block is reached during programming - see "Invalid block replacement scheme" paragraph later.
- During the device lifetime (acquired invalid blocks). NAND flash memory 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 labelled and treated in the same way as the inherent invalid blocks. There are various techniques how to minimize the block wear-out. They must be taken into account for target device system design, but typically are out of concern during the factory pre-programming.

## Seven elements of NAND flash factory pre-programming scheme

NAND flash factory pre-programming scheme consists of seven basic elements that need to be specified to cover all NAND flash issues. These are:

- Partitioning scheme
- Invalid blocks replacement scheme
- Dynamic meta-data scheme
- Spare area arrangement scheme
- Error control and correction (ECC) scheme
- Unused blocks formatting scheme
- Input data file scheme

See following pages for details on individual schemes.



## Partitioning scheme

Typically, there are various kinds of data programmed into NAND flash memory device. These various data types may need to be processed using different approaches. Also, each data type may need to start at exactly specified location.



Figure 3. NAND flash memory partitioning example

Figure 3 shows an example of simple device partitioning scheme. There are four partitions defined in device for bootstrap, bootloader, operating system and file system in this example.

One can see that some partitions contain padding area. Padding area represents the “reserve” meant for invalid block treatment or for future use. E.g. if bootloader code needs 4 blocks, it is good practice to allow 1 or 2 other blocks for invalid blocks replacement.

Padding area can also cover whole partition, if some special pre-formatting of blocks that are ready for use is necessary. For more information on padding area formatting see “Unused blocks formatting scheme” paragraph later.

Please, note the bright-blue coloured areas between the partitions. They represent areas that are not intended for use, so-called “gaps”. There is no meaningful reason to waste the memory capacity and left the gaps between the partitions. However, there can be some other partitions defined, out of scope during factory pre-programming (will be programmed later, e.g. after target device burning-in).

The opposite situation - partitions overlap - is forbidden.

To define the partition, the following values must be specified:

- Partition start - physical address (block number) where partition starts.
- Partition end - physical address (block number) where partition must stop. Typically, an error is reported if it is not possible to program all required data to area between partition start and partition end.
- Partition size - the size (in bytes or blocks) of partition data. Area behind the partition size until the partition end comprises the padding area. If there is none invalid block in partition, only the area corresponding to partition size will be used. But in case of invalid blocks occurrence, blocks from padding area will be used to replace those invalid ones.

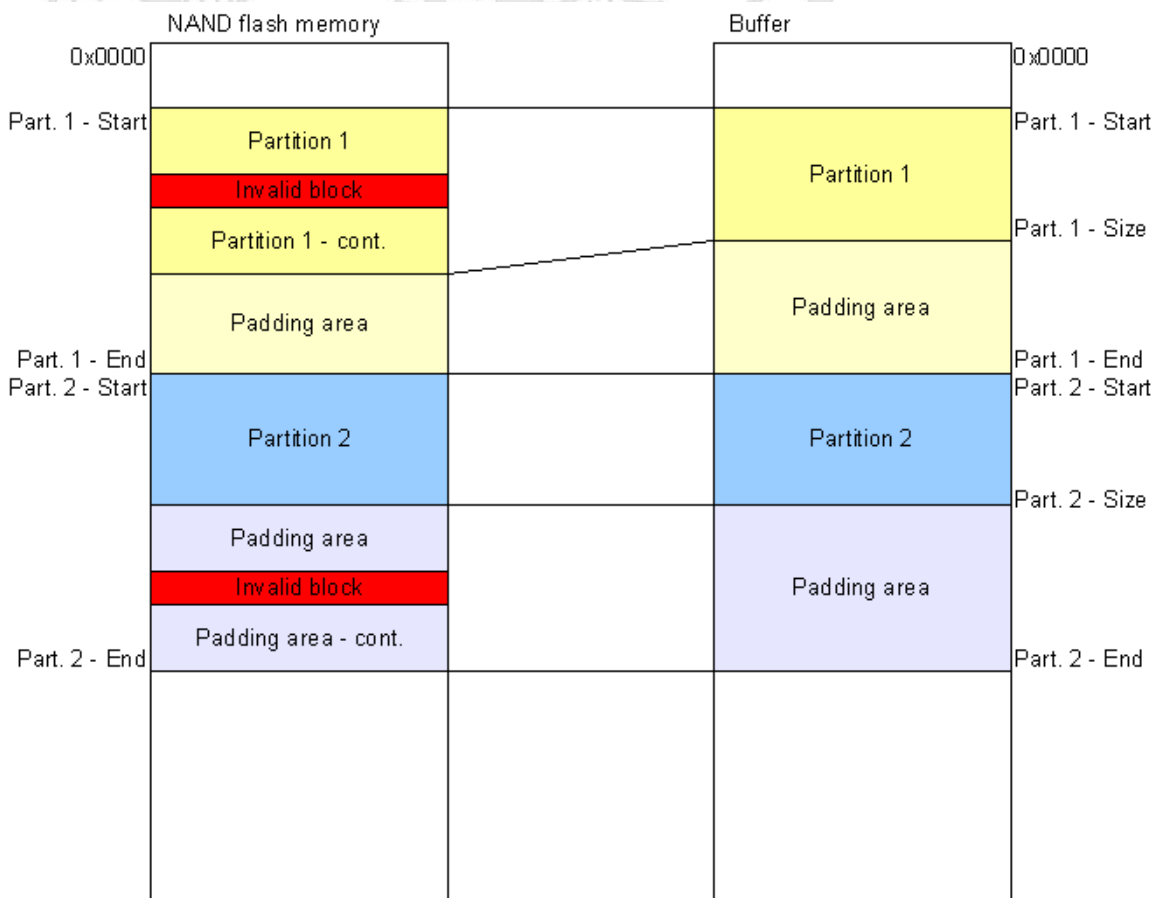


Figure 4. Partitioned device versus buffer mapping

Figure 4 shows an example of device versus buffer mapping. The scheme with two partitions is used. There is one invalid block in data area for Partition 1 and another one in padding area for Partition 2.

One can see that input data file must cover also padding areas. Typically, these areas are left blank. If any formatting of padding blocks is required, there are two possibilities:

- all necessary formatting data can be included in input data file (simplifies algorithm implementation so it can be done faster), or
- special formatting must be specified in Unused blocks formatting scheme (input file still must contain blank bytes for that areas).

There are two aspects that must be taken into account when defining partitioning scheme:

- How many data types with different processing requirements will be programmed?
- How many data areas need to start at exact address?

After the number of partitions is determined, also the start, end and size of each partition must be specified. Also, all other schemes must be specified for each one partition. However, very often all partitions use the same scheme.



## Invalid blocks replacement scheme

Invalid blocks replacement scheme specifies how to proceed if invalid block is reached during programming. It is often so-called Invalid blocks management scheme.

Three generally used schemes are discussed in following paragraphs.

### Ignore invalid blocks

This scheme is used mainly in older systems. Using ignore invalid blocks scheme, invalid blocks are ignored (treated as being valid). In consequence, respective data are lost. It is necessary to back-up system critical data - but this raises demand on additional memory capacity.

As the scheme is rather inconvenient in complex system, it is used very rarely. However, it can be still used for storage of the unimportant data, if eventual data loss can be forgiven.

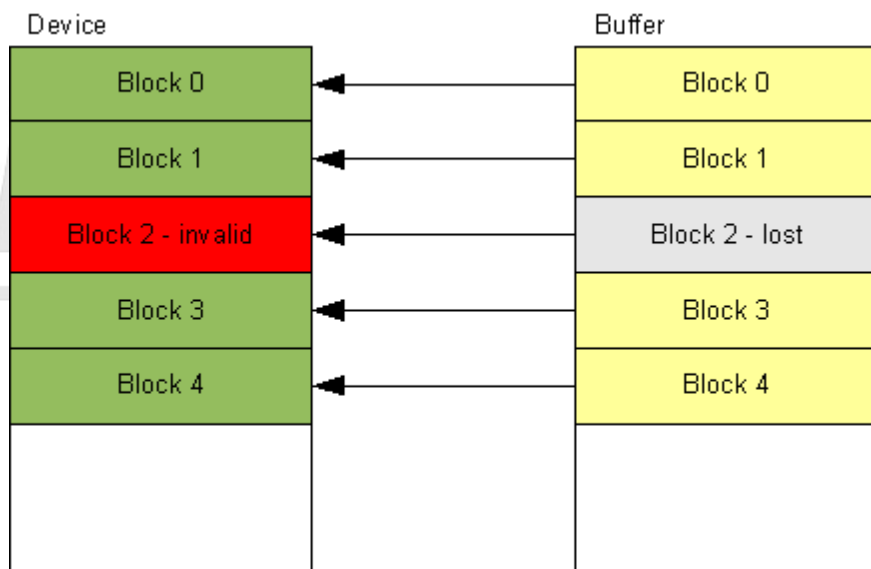


Figure 5. Ignore invalid blocks scheme

### Skip invalid blocks

This is the most commonly used invalid blocks replacement scheme. If invalid block is reached during programming, it is skipped (left on its own) and relevant data are programmed into next valid block. Data for all other blocks are shifted by one block, until other invalid block is reached. This will be skipped again, relevant data will be programmed into next valid block, so data will be shifted by two blocks from now... and so on again. In this way, data end will be shifted by the number of invalid blocks reached during the programming (from that reason there is padding area used if partitioning).

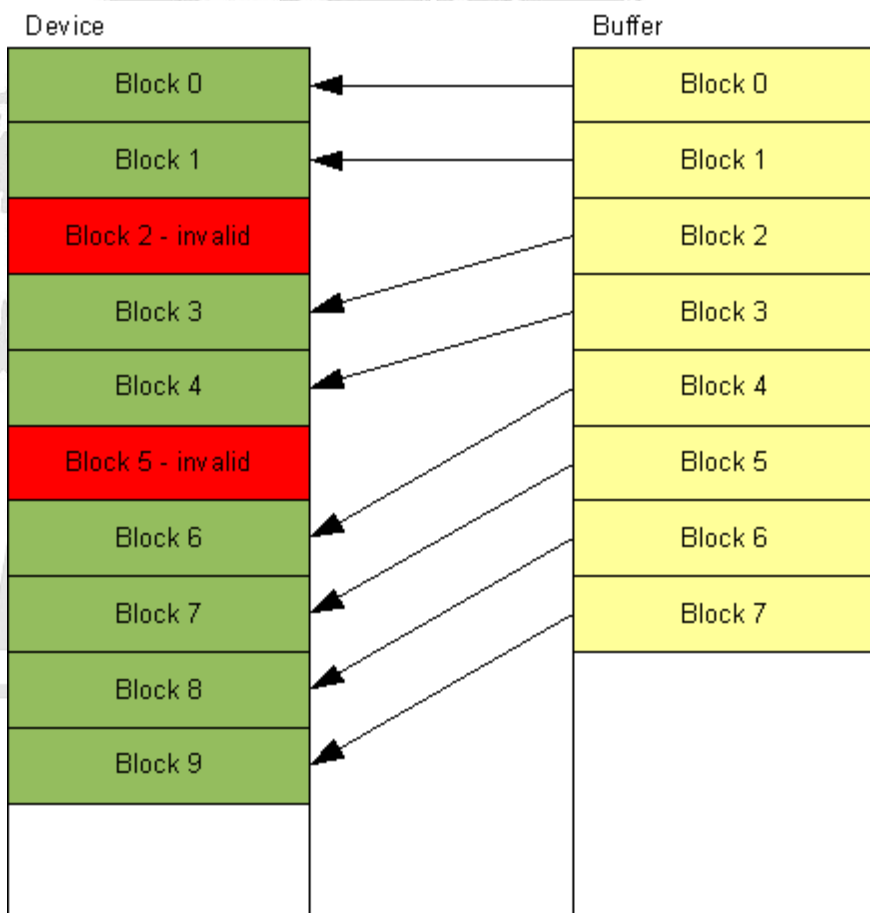


Figure 6. Skip invalid blocks scheme

## Reserved blocks area

Reserved blocks area scheme divides the memory device into three areas (see Figure 7):

- User Area (so-called Data Area)
- Reservoir (so-called Redirection Area)
- Redirection Table Area (so-called Info Area)

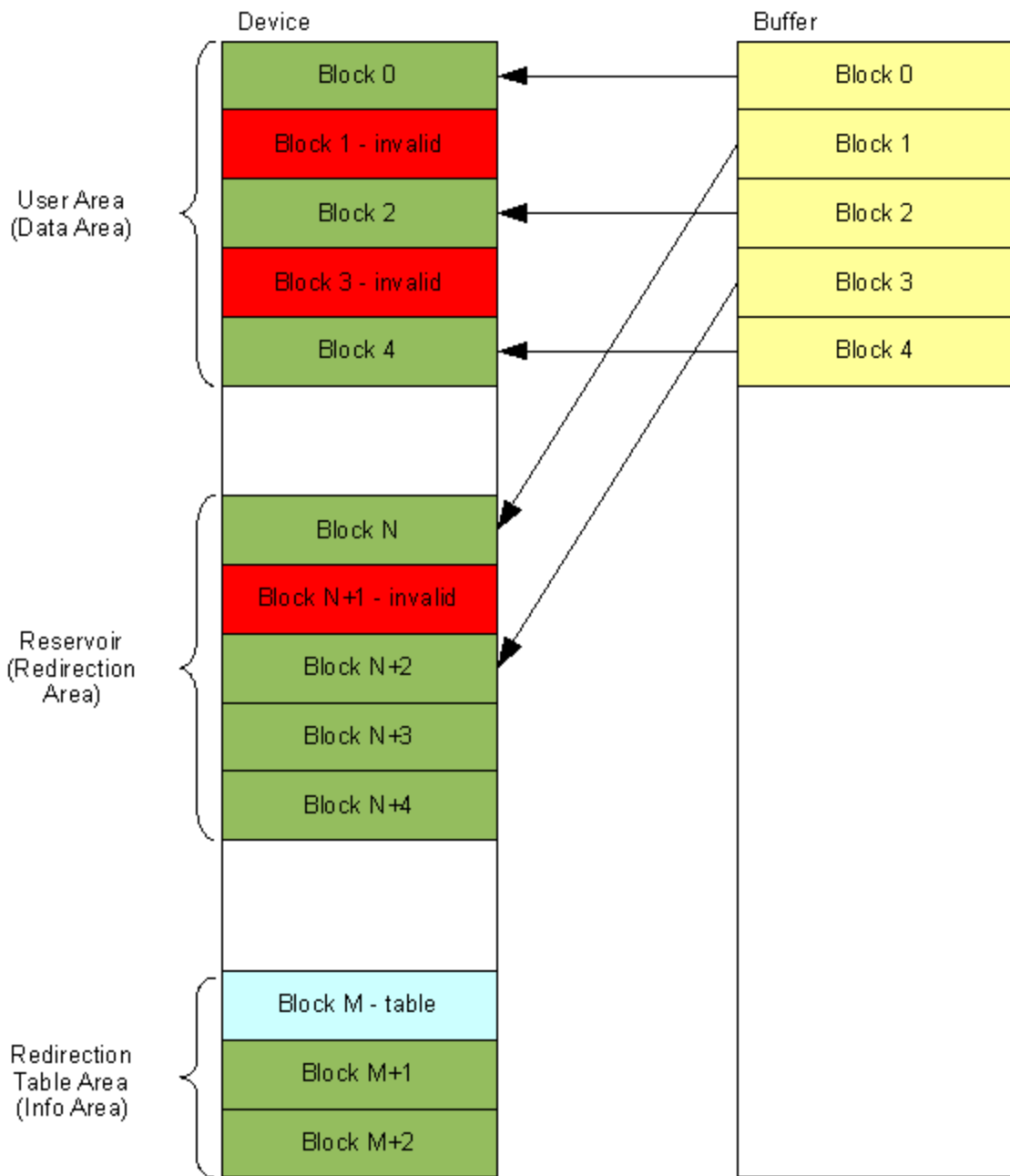


Figure 7. Reserved blocks area scheme

Primarily, the user area is used for data programming. If invalid block is found, the algorithm searches the reservoir for suitable valid block and uses it for replacement. Then, the programming process continues from next block in user area.

After the programming is finished, the information about redirected pairs is compiled and programmed into redirection table area.

What is necessary to specify Reserved blocks area scheme completely:

- User Area start and end, eventually start and size
- Reservoir start and end, eventually start and size
- The way how the blocks are picked-up from reservoir (e.g. from start to end or from end to start)
- The number of redirection table copies
- Redirection Table Area start and end, eventually start and size (there may be also several areas reserved for redirection table - e.g. block #0000 for redirection table and the last device block for table backup copy)
- The way how to select the block for redirection table copies programming (from start, from end, etc...)
- Redirection table formatting (headers, checksums, redirected pairs formatting, etc.)



ELNEC

## ***Dynamic meta-data scheme***

All data, that **MUST** be prepared on-the-fly during the factory pre-programming procedure are called “dynamic meta-data”. They **MUST** be prepared on-the-fly, because - of their nature - it is not possible to build involved fields, tables, etc. at data preparation stage. Typically, these are:

- serial numbers
- MAC addresses
- invalid blocks lists (based on invalid blocks occurrence)
- lists of blocks free for use (valid yet not used blocks)
- mapping tables (based on logical to physical block numbers mapping - see redirection table in previous paragraph)
- file system headers
- and similar

What is necessary to specify Dynamic meta-data scheme completely:

- exact locations and sizes of all dynamic meta-data fields, tables, etc.
- exact specification how the meta-data should be loaded (serial numbers, MAC addresses) or compiled (blocks lists and similar)

## ***Spare area arrangement scheme***

Spare area carries various kinds of system-level information, such as logical block / sector numbers, operating system look-up data, block validity label, block usage / free label, ECC checksums, etc.

If your system uses spare area only for storage of block validity information in the same manner as the memory devices manufacturer's specification says, the spare area can be considered being "not used". In such a case, don't forget to specify that you are not using the spare area.

In all other cases the spare area is used.

Spare area data for static data areas can often be pre-computed and included into input data file. It is recommended to do so, as this way simplifies algorithm implementation and verification (faster implementation time), reduces PC processor load (faster programming time) and simplifies system changes (e.g. if you will decide to change used ECC algorithm, no change in device support will be required). If, despite of all above mentioned advantages, you decide to let the programmer compute the spare area data on-the-fly, don't forget to specify the following:

- each spare area byte exact content assign
- the computational algorithm for each field of spare area (logical sector / block numbers, ECC checksums, etc.)

Spare area data for dynamic data areas must be, similarly the dynamic areas itself, computed on-the-fly during the factory pre-programming process. In such a case it is necessary to proceed like if spare area data for static data are computed on-the-fly, see previous case.

What is necessary to specify Spare area arrangement scheme completely:

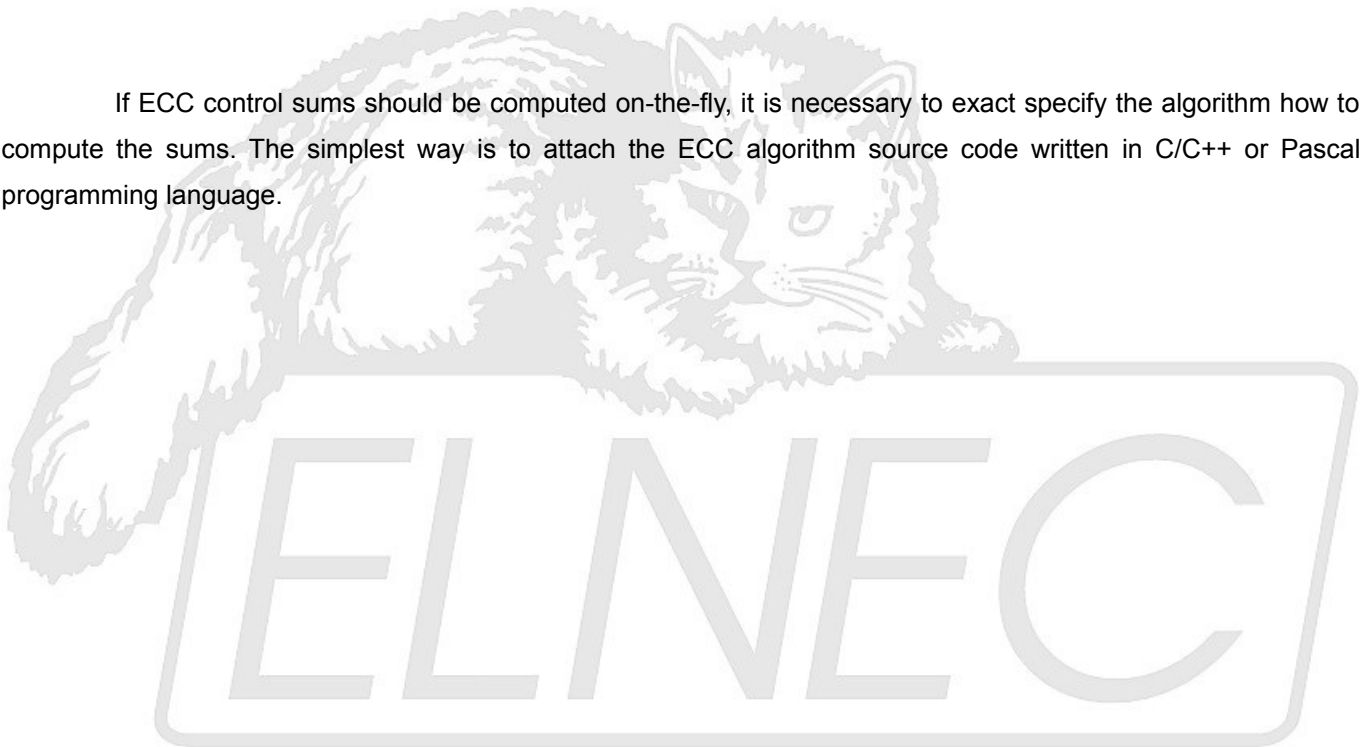
- Is spare area used?
- If yes, does input data file contain also relevant data for spare area?
- If not, what is spare area bytes assignment? How to compute their values?
- If device contains dynamic data areas - do these areas use spare area?
- If yes, what is spare area bytes assignment? How to compute their values?

## ***Error control and correction scheme***

Various error recovery mechanisms are used to minimize device failure rate during the read operation. Single-bit errors (so-called flip-flops) in NAND flash technology occur naturally. All NAND flash memories manufacturers recommend to use an error control and correction (ECC) algorithms to extend the memory devices lifetime. Typically, 2 bits detecting / 1 bit correcting algorithm is recommended for single-level cell devices. The situation is rather worse within multi-level cell device - typically 4 up to 12 bits correcting algorithms are required.

ECC control sums are typically stored in spare area. Their byte position assignment should be specified in Spare area arrangement scheme. Also, their including into input data file is recommended, similarly to other spare area data.

If ECC control sums should be computed on-the-fly, it is necessary to exact specify the algorithm how to compute the sums. The simplest way is to attach the ECC algorithm source code written in C/C++ or Pascal programming language.



## **Unused blocks formatting scheme**

Sometimes, it is necessary to pre-format also the blocks that are not used (e.g. some operating system specific data in spare area). There can be two kinds of such blocks:

### **Padding blocks**

The formatting data can be included in input data file. Using this way, however, the blocks will be not considered unused by programmer. It is necessary to specify that omitting of excessive blocks that cannot be programmed into specified user data area (partition) is allowed.

Other way is to specify the pre-formatting scheme and left corresponding area in input data file blank. After programming the area corresponding to partition size (see Partitioning scheme paragraph) the remaining blocks until the partition end will be processed using the specified scheme.

### **Invalid blocks**

Remember, that inherent (device original) invalid blocks are present already at time of memory device shipment. They are labelled using some labelling scheme specified by device manufacturer. It is highly recommended to adhere to this scheme and don't violate it. Also, it is recommended to use the same scheme for acquired (raised on use) invalid blocks.

Sometimes, it may be necessary to use some specific features for acquired invalid blocks labelling. E.g. a value of 0x0F may be used instead of common 0x00. Or all locations inside of acquired invalid blocks may be programmed to 0x00.

Since new invalid blocks can arise also during factory pre-programming operation, it is necessary to specify and implement invalid blocks (of both types) labelling scheme already for device pre-programming support. It may be specified as a part of Invalid blocks replacement scheme.

## ***Input data file scheme***

Ideally, input data file should contain a “mirror” of device content. As NAND flash memory devices capacities are often really huge (several gigabytes), such input data files became rather uncomfortable. Therefore, there are various techniques used how to “pack” unused (blank) areas. If any such technique is used, the exact specification must be provided.

Sometimes, there can be some safety features used to disable the usage of incorrect file. E.g. the file compiled for pre-programming algorithm of revision 1 cannot be used in combination with pre-programming algorithm of revision 2. All such features must also be exactly specified.



## Conclusion

There isn't any general template available so that we might simply allow you to fill-in and sent to our technical support department. Please, use this application note as the guideline and build your own specification in easy readable form. We can accept the specifications in DOC, XLS, PPT, ODT, ODT, ODP, PDF, raw text and compatible formats.

Please, write your specification using English language. Avoid using national fonts, as they may became not-readable on our side.

If you have any doubts in terminology, try to explain the task in your own words. You can use also some kind of easy understandable programming quasi-language (C-like). We will contact you an discuss eventual problems.

Don't forget to attach exact algorithms for all on-the-fly computational tasks, such as ECC and similar. The most useful form is the algorithm implementation source code in C/C++ or Pascal programming language (may be, the part of the project code for your target application memory controller - this is the best way to ensure the consistence with your project).

Please, attach also sample input data file for implementation verification purposes on our side.

Please, assume, that we may ask you for sample memory devices (1 to 4 units). They can be sent back to you after successful implementation, however, it is recommended to keep at least one sample on our side for possible post-release support in case of any problems or modifications.

Do not hesitate to contact our technical support department in any doubts, even during the preparation of the scheme. You can use our Algorithms On Request service available from our web-site: <http://www.elnec.com/support/algor-service/>.

## Version history

Version 1.0 – February 2009

– initial release

