Technical Guide

Programming NAND devices

Kelly Hirsch, Director of Advanced Technology, Data I/O Corporation

Recent Design Trends

In the past, embedded system designs have used NAND devices for storing data during run time; therefore, manufacturing was not required to pre-program these NAND devices. Any boot code or Operating System (OS) code was typically stored in NOR flash.

In recent months, design engineers have been reducing the cost of their products by using NAND devices in systems that were "traditionally" NOR based. As memory devices increase in size (due to the application) the cost also increases. The price of NAND flash devices is currently about one-third to one-fourth the cost of binary NOR, and business imperatives put severe pressure on cost reduction, especially in the wireless industry. These events increasingly validate the effort for design teams to add the extra layer of software (and sometimes hardware) required to design in NAND.

As a result, designers are increasingly developing designs that can boot from NAND (and then move the OS into RAM); there is no NOR-type memory required in these new systems. This approach drives new requirements in manufacturing to program the NAND devices with boot code.

In addition to the increased cost, larger memory requires more time to program. Regardless of the type of memory used, programming large devices at In Circuit Test (ICT) or Functional Test (FT) will slow down the production line. This potential bottleneck is driving manufacturing engineers to seek other solutions such as pre-programming the devices with "off-line" or "in-line" programming equipment.

Data I/O's unique solution to this production bottleneck rests on the high-speed FlashCORE architecture for device programming. In the FlashCORE system, all memory devices (both NAND and NOR) are programmed with parallel data paths and in gang mode, greatly increasing the efficiency and lowering total cost of manufacturing products which use large amounts of NAND and NOR flash memory. This paper will examine the differences between NAND and NOR in device architectures and how those differences influence approaches to programming.

NAND versus NOR

With NOR memory, all memory locations are guaranteed to be good and to have the same level of endurance. To improve yields and keep costs down, NAND devices contain randomly located bad blocks in the array. A programming approach for NAND devices therefore must have a scheme to identify and avoid the bad memory cells when programming the device.

Physically, NAND memory devices use a smaller transistor because they don't have to "pull down" a whole bit-line (with relatively high capacitance). 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 NOR and NAND and the approximate size difference.

Since cost is a function of die size, NAND devices are lower cost for the same memory size. Also, for NOR memory, 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. But for NAND, a minimal amount of extra memory is set aside to repair defective regions on the die because it is not necessary to repair the entire array. This further reduces the size (and therefore the cost) of a NAND die relative to a NOR die.

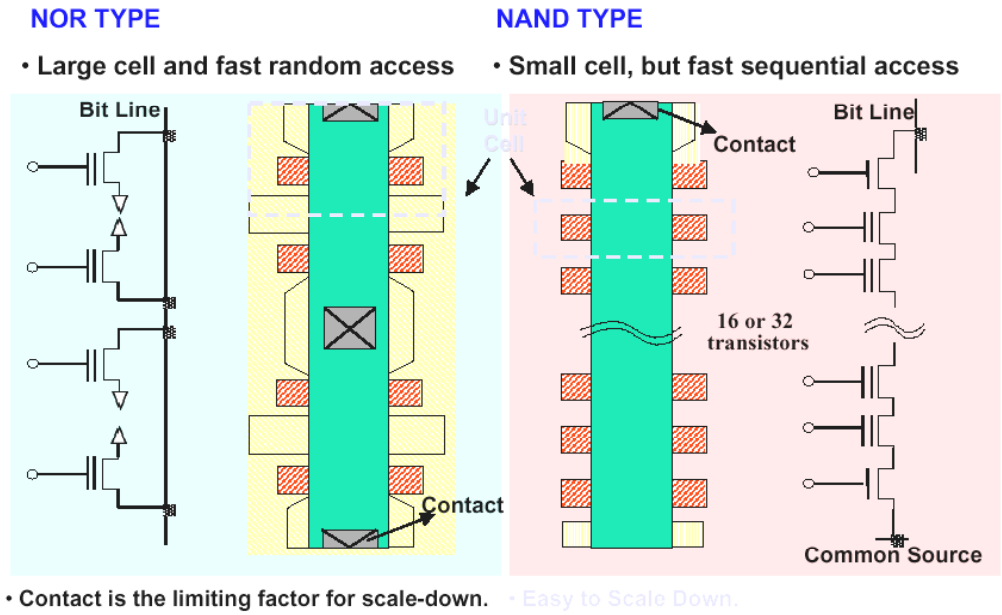


Figure 1: NAND and NOR cell comparison (from Samsung application note).

A block of NAND memory usually consists of 16, 32, or 64 pages. For most NAND devices there are 512 bytes in the "normal" page area and then an extra 16 bytes in the "spare area" for a total of 528 bytes per page. If a memory location is bad, then the entire block is "marked" as a "bad block." Typically, a NAND device starts out with a few bad blocks that are marked by the factory. Usually this is done by writing non-"FFh" data at byte 517 in the first two pages of the block. However, a new bad block can show up whenever it has data written to it. Usually, the block has to be erased and reprogrammed several times before the probability of creating a bad block is significant.

To handle these bad blocks in the embedded system, an extra "layer" of software is required. (This is similar to the software required to manage hard disks that can have bad sectors.) Therefore, to program these devices, all we need to do to handle the bad blocks is to treat them in exactly the same way as they are handled by the user's system. But, as simple as this may sound, there are many ways to do this and there doesn't seem to be any universally accepted standard method.

For example, one common method is to just skip over the bad blocks and place the data in the known good blocks. (We call this the Skip Block Method.) Another common method (by Samsung) is to allocate some of the blocks as a "Reserve Block Area." In this area, one block is used for a "table" that keeps track of the bad blocks and the rest are used for the data that would have been written to the bad blocks. (Both methods are contained in our current base algorithm for NAND devices.)

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 (depending on the ECC algorithm) fix the bad bit. Typically, three bytes of ECC are generated for each page and placed in the "spare area" of the page.

Another important difference between NAND and NOR is the read/write characteristic as shown in Figure 2. NOR is programmed and read by addressing each byte while NAND is programmed and read back (sequentially) page by page.

The programming times are also different. NAND gets charge into the floating gates of the memory cells through a process called Fowler-Nordhiem (FN) tunneling. FN tunneling involves applying an electric field across a dielectric to induce charge to "tunnel through" the dielectric. This process takes about 200 µsec for NAND. For NOR, they typically use the Channel Hot Electron (CHE) effect to get charge on the floating gate. This is faster (about 10µsec) but it takes more current.

Even though it takes longer for the charge to get into NAND cells, because NAND is programmed a page (528 bytes) at a time, the total programming time is much faster with NAND than with NOR memory which only programs 1 byte at a time. Figure 3 is a table that shows the difference between programming processes for a typical NOR flash device and a NAND device using FlashPAK, Data I/O's new gang programmer. (Note that for the NOR erase time, the worst case specification was used; typical erase times for this part are about 125 sec.)

	NAND Flash Memory*	NOR Flash Memory
Read Unit	Page (512 + 16 bytes)	Byte/Word
Access Time (typ.)	7 µs (initial access) 50 ns (serial access)	90 ns (random access)
Write Unit	Page (512 + 16 bytes)	Byte/Word
Write Time (typ.)	200 µs	8 µs/Byte 16 µs/Word (4 ms/528 bytes)
Erase Unit	Block (8K + 256 bytes)	Sector (8K/64K bytes)
Erase Time (typ.)	2 ms	1s

Figure 2: NAND vs. NOR read/write characteristics.

Device	Blank Check	Erase	Program	Verify
Intel 128 Mbit StrataFlash (NOR)	2 sec	250 sec max Typ. 125 sec	122 sec	9 sec
Samsung 128 Mbit NAND	8 sec	2 sec	10 sec	13 sec

Figure 3: Table of programming times on FlashPAK

File Systems

Some embedded products use NAND devices to store "files" as well as the boot code or OS code mentioned earlier. Therefore, additional software is required to manage the NAND memory used as a file system. This may include a format function, wear leveling, and the ability to de-fragment the memory area. Most of the NAND manufacturers have software that manages these higher level functions and are willing to license the software to an end user.

For example, SanDisk has a software product called "Stingray" that is essentially a NAND management system, and Samsung has a custom ASIC that manages ECC and hardware protection.

Another example is M-Systems DiskOnChip® devices. M-Systems manufactures memory devices that are based on NAND but have a "thin controller" layer that makes integration much easier. M-Systems will license their *TrueFFS*® file system software in order to access and manage their DiskOnChip® devices. Data I/O has ported *TrueFFS*® to its FlashCORE programmers so that M-Systems' customers can now use Data I/O's most advanced programming technology.

The characteristics of the file system being used are a key consideration in the approach to device programming. To optimize programming accuracy and speed, Data I/O's FlashCORE system needs access to the file system details to support pre-programming with files.

Bad Block Schemes

Data I/O's popular TaskLink for Windows™ software is used to create "jobs" for devices programmed on FlashCORE programmers. Typically, the user creates the job, associates it with the programming data file, and stores the information on a PCMCIA card. This card is then inserted into the programmer to begin programming. No computer is need to run the programmer; only to create the job.

Currently, we have two methods available in our basic NAND algorithm – the "skip-block" method and Samsung's "Reserve Block" method. The user can also use the device without any bad block scheme. (Note that some NAND devices are available with all blocks guaranteed to be good.)

Figure 4 shows the dialog box that is produced by TaskLink when a NAND device is selected from the device list.

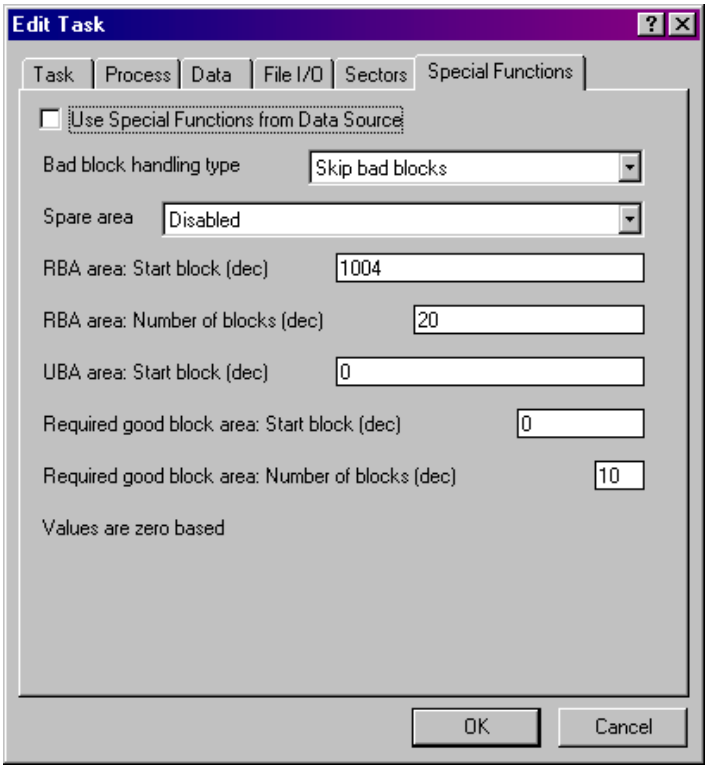


Figure 4: Special Functions for NAND devices.

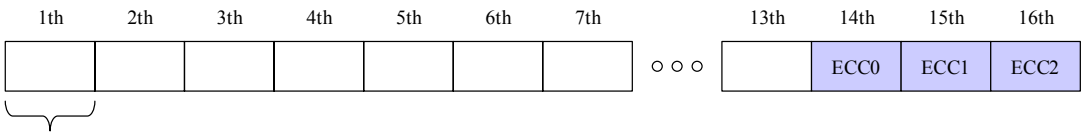
Skip Block Method

This method is very straightforward as mentioned above. The algorithm starts by reading the entire spare area (only) of the entire memory. The addresses of the factory-marked bad blocks are then collected in the programmer RAM. Next, the image is sequentially programmed (page by page) into the target device. When the target address corresponds to a bad block address, these pages are stored in the next good block, skipping the bad block. Because this bad block is skipped, the original factory programmed (non-FFh) data in the spare area indicating the presence of the bad block is still there. Therefore, the user's system can also build a table of bad block addresses by reading the spare area of all the blocks at boot time.

Note that the user can choose to use the spare area to store user data/code or leave it blank. If the spare area is used, the user must modify the linear image so that "FFh" is written to byte 517 (in the spare area). In this way, the system can still generate a bad block table after the device has been programmed by Data I/O equipment. In other words, there won't be any confusion between good blocks that contain data in the spare area, and bad blocks that were marked with non-FFh data at the factory.

The choices for the drop-down list titled *Bad Block handling type* are "None", "Skip bad blocks" (as shown above) and "Samsung RBA style." For the *Spare area* drop-down list, the choices are "Disabled," "Enabled," and "ECC."

If "ECC" is selected for the "Skip bad block" method, the algorithm will calculate three bytes of ECC data corresponding to the 512 bytes of data in each page. The ECC bytes are placed in the 14th, 15th, and 16th byte location of the spare area (byte 526, 527, and 528 of each page) as shown below.



Note that if "ECC" is selected from the "Spare area" drop down list, the rest of the spare area is left blank (FFh); the spare area cannot be used for both ECC and data storage. For details on the method used to calculated the ECC value, see the code example on Samsung's web site at http://www.samsungusa.com/download/semiconductors/flash/eccalgo.pdf.

Also shown in Figure 4 are the parameters for specifying a "known good" block area. This is typically used by systems that have to boot from NAND. The boot code must reside in good blocks because most systems can't handle bad blocks at boot – the code hasn't been loaded yet. The user can specify both the starting block for the known good area and the size.

During device programming, if a bad block is encountered in this range of blocks, the device is rejected (fails programming). The probability of failure increases when a larger number of blocks are used for a "known good area." We recommend that the minimum number of known good blocks be set aside for the boot code.

Note that most NAND manufacturers guarantee that the first block is good. Therefore, if your boot code can fit into a single block (and you specify a known good range of 1 block with a starting address of 0), you will not have any failures due to bad "boot blocks" during programming.

The "known good" feature applies to both the Skip Block method and the RBA method discussed in the next section.

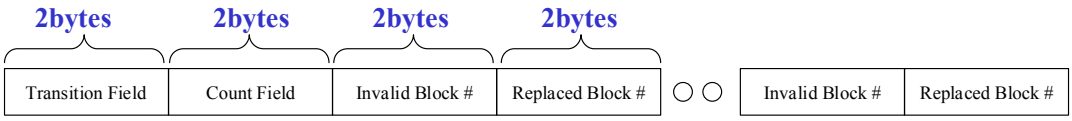
Samsung RBA Method

The Samsung RBA method is based on the "Reserved Block Area" concept. In the user's system the bad blocks are replaced with good blocks by "re-directing" the system to a known good block.

The programming algorithm works by first determining which blocks will be used as the User Block Area or UBA and which blocks will be used as the Reserved Block Area (RBA). The starting address and size of the UBA are determined by the parameters entered in the Special Function dialog as shown in Figure 4. Also, the starting block of the RBA is defined by the user as well.

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 in the RBA are used for this table.

The map table contains information on how to substitute bad blocks in the UBA with good blocks in the RBA. The data fields in the map table are shown below.



The *Transition Field* is always FDFEh. The *Count Field* is incremented by one for each page of the map table. For the current algorithm, there are only two pages used so the field will contain 0001h for the page in the first block and 0002h for the page in the second block of the map table.

The data pair *Invalid Block / Replaced Block* show the address of the bad block and the address of the replacement block respectively. The rest of the page consists of these 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 bad blocks is typically less that 1% of the total number of blocks for new devices being programmed. Of course, new bad 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 page of the map table is used to duplicate the information in the first page in case one of these pages becomes corrupt during usage. All fields use "little endian" protocol; the low byte is first.

For details of how to implement this method in your embedded systems design, please consult with Samsung Field Application Engineers in your area.

Programming NAND on FlashCORE Programmers

When you use TaskLink for Windows to build a programming job, you can select the operations shown in the *Edit Task* dialog box (Figure 5).

For the case shown in Figure 5, *Erase Device* is not checked because this job is for a new device which does not need to be erased. However, it's always a good idea to do a *Blank Check* to make sure the device was not previously programmed.

Note that with Data I/O equipment, if a NAND device is erased, the factory programmed bad block information is left intact. However, if you have selected "None" for the "Bad Block handling type" and "Enabled" from the "Spare area" drop-down list (see Figure 4), then the entire device, including the spare area, will be erased. This should be done only for NAND devices that are guaranteed to have no bad blocks by the manufacturer.

Verification of the image in the device is accomplished by reading the entire device and performing a binary compare with the original data (in programmer RAM). If there is not an exact match, the programmer will report a verification error for that device.

As mentioned above, it is possible to generate a new bad block whenever NAND devices are being programmed. This is the main reason "wear leveling" is used in NAND management systems. A wear leveling routine will make sure the same block is not erased and reprogrammed repeatedly. This is

accomplished by changing the physical to logical "map" continuously so that, over the course of time, all blocks are utilized the same amount.

Since the probability of generating a bad block is proportional to the number of times each page in the block is programmed, it is very rare to induce a new bad block when these devices are being programmed for the first time. Therefore, Data I/O equipment does not allow newly generated bad blocks during programming. If verification fails (due to a new bad block showing up or any other reason) the device is not passed. For our automated equipment, the device is automatically placed in a reject bin or tray. A failure of this type is indicated as a "verify failure" in the log file and a red LED is turned on next to the device that failed.

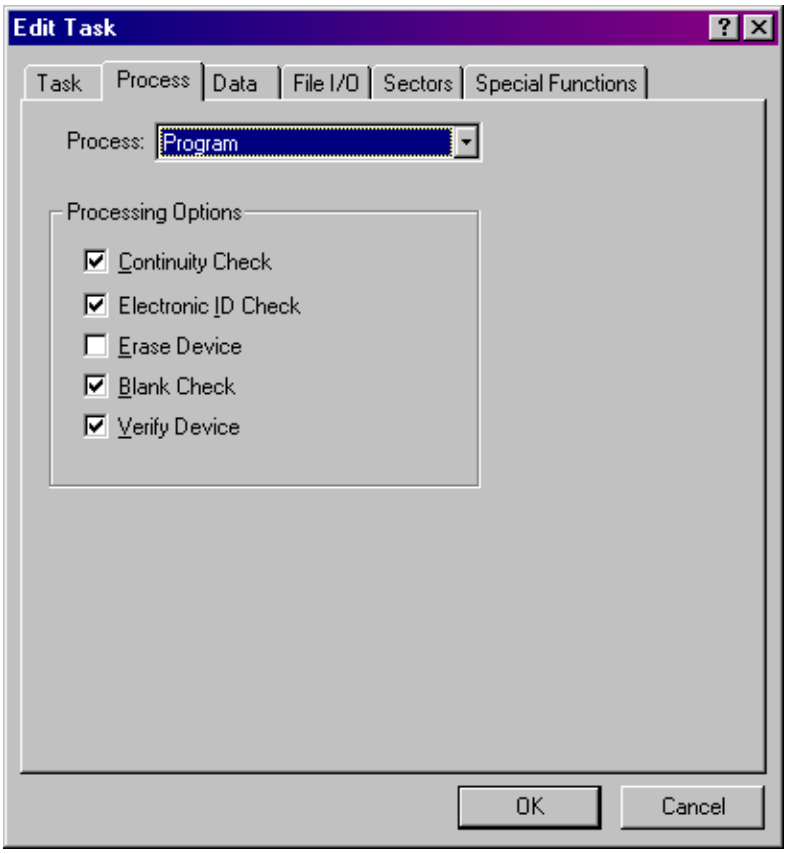


Figure 5: Edit Task dialog box from TaskLink for Windows.

Designer's Checklist

Most NAND devices can easily be supported on Data I/O's FlashCORE programmers. The following set of questions is intended to provide guidance and assure compatibility with our existing NAND algorithm.

1) Do I need bad block management?

   a) Yes, if you are using any NAND device that can potentially have bad blocks. (The exception to this is M-Systems DiskOnChip® devices where the bad block management is handled by the controller that is integrated onto the NAND memory array.)

2) Will my system be able to identify the bad blocks after programming?

   a) Data I/O programmers do not erase the "bad block" indicator that is placed in the spare area. Therefore, the system can always check specific bytes (of the first page) in each block. If the value is non-FFh (for x8 devices) or non-FFFFh for x16 devices then the factory has marked the block "bad." (Be sure to consult the manufacturer to verify the location they use to mark bad blocks.)

   b) For the RBA method, a map table is created by the programmer that contains all the bad block information. You can specify the starting address of this map table when you program the device.

3) Do I need ECC?

   a) For boot code applications this is usually not necessary because the data is only programmed once. (Remember, bad blocks only show up during programming.)

   b) For data/parameter storage where a block is erased and reprogrammed (or over programmed) during the application, you will most likely need limited ECC. Data I/O FlashCORE programmers can generate three-bytes of ECC per 512 bytes of data. This ECC method allows for detecting 1-bit and 2-bit errors. The ECC bytes are placed at the end of the spare area for each page.

   c) For applications where the NAND device is being used as a file system, the file system software usually handles ECC. The only file system supported on FlashCORE products at this time is M-Systems' TrueFFS. ®

4) I have a special proprietary ECC algorithm for my application. Instead of having Data I/O duplicate this algorithm in my equipment, can I just modify my binary file with the ECC data instead?

   a) Yes. If you choose to run your raw image through some kind of utility that places the ECC data in a certain location, then Data I/O FlashCORE programmers can just program the device without ECC selected. In this case we recommend that you place the ECC bytes in the spare area (and enable usage of the spare area when you create the job).

5) Do I need to port my file system to Data I/O's programmer?

a) If you use a file system (like Microsoft's BinFS) but do not need any files placed into the NAND device during manufacturing, then you do not need to port the file system to Data I/O's programmer. This is true if, for example, you have your operating system in NAND (programmed at manufacturing) but you only create files after the OS is transferred to SRAM and you launch your application.

b) If you use a file system and you expect some files to be available at boot, then those files need to be programmed with the same file system you are reading them with. So in this case, contact Data I/O about the possibility of porting the pertinent portion of your file system to Data I/O's programming platform in order to support your product.

6) I am using a Multi Chip Package (MCP) with NOR flash, NAND flash, and SRAM in a single package. Can I use Data I/O equipment to program both the NOR and the NAND at the same time?

a) Yes. Our FlashCORE programmer has the capability to program multiple devices in one package. Since each MCP device may have different pin-outs, the first and most important thing to do is work with the manufacturer to make sure an "open top" test socket is available.

Conclusion

Data I/O FlashCORE™ programming systems are flexible and powerful, allowing designers and manufacturers to unleash the full potential offered by NAND flash technology. Visit http://www.datio.com/nand/nandflash.asp to view a list of supported NAND devices. If your NAND design and implementation challenges include a need for a customized approach to bad block handling, or a solution for volume programming to commercialize your application, we invite you to contact us.