NAND FLASH management under WinCE 5.0

Kevin Wells
NXP Product Line Microcontroller - Business Line Standard ICs
Systems Design Engineer
Agenda

- Quick NAND FLASH overview
  - NAND vs NOR
- MicroSoft WinCE 5.0 FLASH components
  - FLASH Abstraction Layer (FAL)
  - FLASH Media Driver (FMD)
- FLASH error conditions and error management
  - Error Correcting Codes (ECC), data correction, and marking bad blocks
- NAND FLASH preparation
  - Device formatting
  - Partitioning from eboot
  - Master Boot Record (MBR), data partitions
- Registry setup and the hive
  - Setup example
  - Common problems
NAND FLASH overview
## Quick NAND FLASH overview
### NAND and NOR differences

<table>
<thead>
<tr>
<th></th>
<th>NOR</th>
<th>NAND</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Memory density</strong></td>
<td>Low</td>
<td>Very high</td>
</tr>
<tr>
<td><strong>Reliable data storage</strong></td>
<td>Yes, up to specified program count</td>
<td>No, even new devices may have bad data areas</td>
</tr>
<tr>
<td><strong>Device pin count</strong></td>
<td>High, requires data bus and a full address bus</td>
<td>Low, only needs 1 address line</td>
</tr>
<tr>
<td><strong>Access latency</strong></td>
<td>Access latency is constant across all device locations</td>
<td>High initial access latency for a block</td>
</tr>
<tr>
<td><strong>Bus type</strong></td>
<td>Address and data bus, random access</td>
<td>Multiplexed address/data bus, sequential access</td>
</tr>
<tr>
<td><strong>Erase time</strong></td>
<td>Slow</td>
<td>Fast</td>
</tr>
<tr>
<td><strong>Software programming complexity</strong></td>
<td>Low</td>
<td>High, need to consider error management when writing and reading from the device</td>
</tr>
<tr>
<td><strong>Erase granularity</strong></td>
<td>Individual locations can be erased</td>
<td>Only blocks can be erased</td>
</tr>
<tr>
<td><strong>Supports XIP</strong></td>
<td>Yes</td>
<td>No (some exceptions for first block)</td>
</tr>
<tr>
<td><strong>Cost</strong></td>
<td>High cost per Mbyte</td>
<td>Very low cost per Mbyte</td>
</tr>
</tbody>
</table>
Quick NAND FLASH overview

- Major NAND devices are available in Single Level Cell (SLC) and Multi-Level cell (MLC) architectures
  - SLC (1 bit of data per cell, faster, 100,000 writes/block, slightly more $ than MLC)
  - MLC (2 bits of data per cell, higher density, 10,000 writes/block, ~70% read speed of SLC)

- Consists of multiple blocks, which contain multiple pages per block
  - Small block FLASH is 512 bytes per page, large block FLASH is 2048 bytes per page
  - Block and page count vary per device
  - Example: a large block device with 4096 blocks of 64 pages/block = 4096 * 64 * 2048 bytes = 512MBytes

- Failure conditions of a FLASH device usually occur at the block level, so all pages in that block are effected, usual failure conditions are bits stuck in the wrong state (bit flipping)
  - It's common for new devices to contain bad blocks, the first block is usually guaranteed good

- FLASH devices MUST be erased before being written
  - Erase of a FLASH device is at the block level, while read and write is at the page level
  - Cannot update a single page inside a block without erasing the entire block
    - An exception here is that you can always set a bit to ‘0’ in a page without erasing it, but to reset it to ‘1’, the entire block needs to be erased

- FLASH device pages sizes have an ‘extra/spare’ data area useful for storing additional information about the block/page
  - A 512 bytes device actually has 528 bytes, while 2048 bytes device have 2112 bytes
WinCE 5.0 FLASH components
MicroSoft WinCE 5.0 FLASH components

- FLASH Abstraction Layer (FAL)
  - The FAL provides a block access interface to FLASH via the FMD. Microsoft provides the FAL library (fal.lib) as part of WinCE
    - File systems access NAND via the FAL
  - Needs to be linked with the FMD to create a block driver that can be used with the WinCE file system

- FLASH Media Driver (FMD)
  - Needs to be written by the software developer for a specific FLASH device and/or host interface
  - Handles initialization, read, write, power management
  - Notifies FAL of the FLASH device geometry (pages per block, page size, number of blocks, etc.)
  - Tags blocks and sectors with WinCE information need by the FAL (bad, reserved, read-only)
  - Handles Error Correction and informs FAL of errors
FLASH FAL/FMD operations

- Predefined function list from Microsoft on the required FMD functions
  - Initialization, de-initialization, powerup, and powerdown
  - GetInfo() – returns geometry of FLASH device (blocks, pages, etc.), used by FAL to determine sector sizing, device size, etc.
  - OEMIoControl() – User-defined IOCTL handler
  - GetBlockStatus(), SetBlockStatus() – returns status of the block (reserved, bad, OK, readonly, etc.)
  - EraseBlock() – erases a single block
  - ReadSector() – Reads one or more sectors from a device
  - WriteSector() – Writes one or more sectors to a device

- The FMD can use the extra data area to store status information about the device
  - WriteSector() and SetBlockStatus() functions may alter this status information
  - Eboot (WINCE boot loader) may initially set the default statuses of blocks and pages
  - Saved information includes block status and ECC information

*Sectors are basically the linear progression of pages inside blocks (ie, block 0/page 0 = sector 0, block 1/page 0 = sector 64, for 64 pages per block FLASH)*
FLASH FAL/FMD operations

- ReadSector() and WriteSector() work at the page level, while EraseBlock() is at the block level
  - EraseBlock() erases all pages inside a block (all bits set to ‘1’)

- If you read, modify, and write data, the data isn’t written to the same physical sector on the NAND FLASH device
  - Because the sector needs to be erased before it can be written to again and because other sectors in the block may have valid data that can’t be immediately erased, the data is written to a new area of FLASH and the sector is flagged as invalid

- FAL compaction thread
  - As sectors become invalid, the compaction thread reclaims them
    - If all the sectors in a block are invalid, the compaction thread erases the block
    - The compaction thread may sometimes move data to reclaim a block that consists of mostly invalid sectors
    - The thread is only kicked off on write operations
FLASH error conditions/management
FLASH error conditions and error management

- **Error types**
  - Only erase/write operations can fail with an hardware error status from the NAND device
    - Blocks that cannot be fully erased are marked as bad blocks by the FAL and are no longer used
    - Data written to a page in a block that fails will be written by the FAL to another block/page
      - The block will be marked as bad
  
- **Error Correcting Codes (ECC)**
  - Data errors can occur without the NAND device generating a status error
    - Data read from a page in a block that fails will attempt to be corrected with ECC logic
      - Uncorrectable data errors return a read failure to the FAL
      - Correctable data errors are fixed and good data is return to the FAL
    - You can write a FMD without ECC, but would be risking your data and wouldn’t know if the read data is bad or good
    - When writing a sector to FLASH, an ECC code should be generated for the data and saved along with the data
      - Can be saved in the FLASH extra data area or any other non-volatile data area
    - When data is read from a FLASH sector, the ECC is re-generated from the read data
      - The generated code is compared against the ECC saved with the data in FLASH
      - A difference between the generated ECC and retrieved ECC indicates a data failure
    - SLC FLASH devices usually exhibit a single bit failure due (1 bit/cell) to a cell failure
      - Algorithms to detect and correct a single bit failure work good for these devices
      - Example algorithm: Hamming codes
    - MLC FLASH devices usually exhibit multiple bit failures (2 bits/cell) due to a cell failure
      - Requires a more complex algorithm than an SLC device
    - A good ECC algorithm can be used to extend the data life of NAND devices
FLASH error conditions and error management

- **Hamming codes**
  - Can detect bit errors in a sector data area from FLASH
  - Can correct a single bit error in a chunk of data
  - Requires 24 bits of ECC for 512 bytes of data, 28 bits for 2048 bytes
  - Code generation algorithm is logic intensive (no math)
    - Generation algorithm uses lots of XOR logic to generate even and odd parity bits for different ranges of bit fields
    - Although the generation algorithm is logic intensive, the correction algorithm is very quick (just a few simple C lines of code)

- ECCs can be generated on sub-sections of the data
  - See FMD block/page strategy example for sectioned ECC layout
  - Requires more processing power (if not using hardware ECC) and more ECC storage area
  - Increases error detection and correction by allowing detection of errors in multiple sub-sections instead of one large section
    - Can still only correct a single bit error in each sub-section

- MLC device may have multiple bit errors inside the same byte
  - Hamming codes would not be an effective correction algorithm
    - BCH algorithm can be used, but has more processing overhead than Hamming code
NAND Block/page FMD strategy example

NAND device (large block)

Block #0  Block #1  .............  Block #n

Page #0  Page #1  Page #2  .............  Page #n

Data area (2048 bytes)  Extra data area (64 bytes)

| Bad block marker (offset 0) (1 byte) | Spare, not used (offset 1) (3 bytes) | SectorInfo (needed by FAL) (offset 4) (8 bytes) | ECC for bytes 0 to 511 (offset 12) (3 bytes) | ECC for bytes 512 to 1023 (offset 15) (3 bytes) | ECC for bytes 1024 to 1536 (offset 18) (3 bytes) | ECC for bytes 1536 to 2047 (offset 21) (3 bytes) | Spare, not used (offset 24) (39 bytes) |
NAND FLASH preparation
NAND FLASH preparation

- **Device formatting**
  - It is common for NAND devices to have random block errors directly from the factory (when new)
    - The factory may already pre-mark bad blocks in the extra data area
    - NAND manufacturers usually specify a minimum number of good blocks in a device
  - Bad blocks may occur and be marked by the FAL during the life and use of the device
  - If formatting from eboot, care must be taken not to erase factory bad block markers
    - Must take care the FMD bad block markers matches factory markers
      - If not, it will need to be matched somewhere
    - Should read every block prior to erasing it
      - Block can be erased if it isn’t a bad block and marked bad if it can’t be erased
  - **Device reformat**
    - Reformat of the FLASH device through the FAL (not eboot) will not erase blocks with bad block markers or reserved markers
NAND FLASH preparation

- Device partitioning
  - The NAND FLASH device can be divided into multiple partitions
    - For example, a BINFS partition might want to be used to store the nk.bin image in it
  - The BOOTPART library can be used with the FMD and eboot to partition and setup FLASH prior to WinCE using it
    - Blocks can be marked as reserved to prevent the FAL from using them
    - The BP_LowLevelFormat() function can be used to initially format the device and create a Master Boot Record (MBR) on the first block (or first non-reserved block)
      - The MBR contains the partition locations, types, and sizes
    - The BP_OpenPartition() function is used to create partitions on the FLASH device
      - Created partitions have entries added into the partition table in the MBR
  - A FAT/TFAT or other partition type should be created
    - This just creates the partition in the MBR, but doesn’t format the partition (setting up volume record, directories, and FAT entries)
      - BP_LowLevelFormat() cannot be used to setup a partition

- If you don’t create an partition, Storage Manager may attempt to use any blocks that are not marked as reserved when setting up FLASH storage
  - But only if autopart is enabled in the registry
NAND partitioning example

- The first block (block #0) is dedicated to the MBR and partition table
  - Contains entries for BINFS and FAT partitions

- Block #1 to block #b is dedicated to the BinFS partition
  - This partition may be a little bigger than the required size of the data to allow for possible bad blocks
  - This partition is marked as Reserved and/or read-only so the FAL will not attempt to erase it

- Block #(b+1) to block #n is dedicated to the FAL
  - For this example, it is tagged as a FAT partition in the MBR
  - This partition can be examined, changed, and formatted by the control panel Storage Manager applet
Registry setup and the hive
Registry setup and the hive

- The hive is a persistent copy of the registry
  - The hive is maintained across power off/on cycles
    - NAND FLASH is an ideal location for the hive
  - When WinCE boots up the first time, the hive registry is created
    - Initial format of the FLASH partition with Storage Manager is needed before the hive accesses it

- There are a few important settings in the registry needed to make the FAL/FMD device work
  - See the “Registry setup example” page for important entries
  - The comments “; HIVE BOOT SECTION” and “; END HIVE BOOT SECTION” are important and not optional

- Simplest way to get the hive working without NAND support in eboot
  - Setup the registry to support autopart and autoformat for the NAND device
Registry setup example

; HIVE BOOT SECTION
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\fireflash]
"Profile"="fireflash"
"Index"=dword:1
"Order"=dword:0
"FriendlyName"="Firefly FLASH driver"
"Prefix"="DSK"
"Dll"="fireflash.dll"
"IClass"={A4E7EDDA-E575-4252-9D6B-4195D48BB865}"

[HKEY_LOCAL_MACHINE\System\StorageManager\Profiles\fireflash]
"Name"="Firefly Flash Disk"
"Folder"="Firefly Storage"
"BootPhase"=dword:1
"MountPermanent"=dword:1
"DefaultFileSystem"="FATFS"
"PartitionDriver"="mspart.dll"
"AutoMount"=dword:1

"AutoPart"=dword:1
"AutoFormat"=dword:1
"MountAsRoot"=dword:0
"MountAsBootable"=dword:1

[HKEY_LOCAL_MACHINE\init\BootVars]
"SYSTEMHIVE"="Documents and Settings\system.hv"
"PROFILEDIR"="Documents and Settings"
"Flags"=dword:3
"DefaultUser"="fflyuser"

; END HIVE BOOT SECTION

Items in bold are important!
Some common problems

- WinCE formats FLASH every time it is powered up
  - Likely cause is different status formats in the set and get status functions in the FMD, check to make sure the SectorInfo data is the same layout for read, write, and erase functions
    - Make sure ECC isn’t overwriting FAL statuses in WriteSector()

- WinCE boot seems to lock up with the hive registry enabled
  - Partition where hive is stored is not formatted, try enabling auto-format in the registry
    - For an extended partition, auto-format will not work until the Storage Manager applet is used to define and format the partitions; this can be a problem if WinCE can’t be brought up to start the applet due to the hive
  - NAND device is not partitioned
    - Either partition the device in eboot using the BOOTPART library or make sure the autopart registry entry is enabled
More information

- NAND FLASH software
  - Micron’s website has a wealth of NAND software related to ECC, drivers, and hardware (www.micron.com)

- WinCE NAND FMD reference driver
  - A reference FMD driver and ECC code is included with WinCE 5.0
    - “WINCE500\PUBLIC\COMMON\OAK\DRIVERS\BLOCK\MSFLASHFMD”
  - The NXP WinCE BSP for the LH7A404 SDK board includes a NAND FMD driver with ECC support for the LH7A404 MCU, tested with Micron and Samsung devices

- 3rd party developers of NAND products and robust file systems
  - CMX (no WinCE drivers) (www.cmx.com)
  - Datalight (www.datalight.com)