CHAPTER 9

Flash Translation Layer (FTL)

The flash translation layer (FTL) is a key component of the firmware in a NAND-based solid-state drive (SSD). It is responsible for managing the interaction between the host computer and the underlying NAND chips, and it plays a crucial role in the performance and reliability of the SSD.

The FTL is implemented as a layer of software that sits between the host computer and the NAND chips, and it serves several key functions: mapping table, bad block management, wear leveling, and garbage collection. These algorithms and data structures are designed to optimize the performance and reliability of the SSD, and they are constantly updated and refined as the SSD is used.

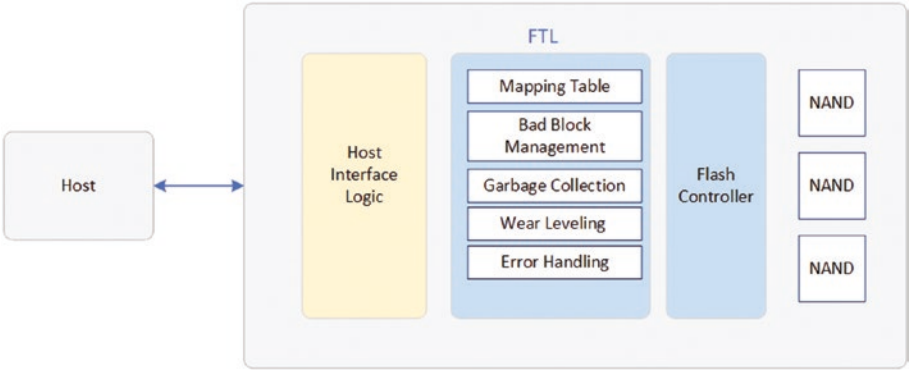

Figure 9-1. *FTL block diagram*

G. Kuppan Thirumalai, *A Beginner's Guide to SSD Firmware*,
https://doi.org/10.1007/978-1-4842-9888-6_9

Mapping Table

The FTL is responsible for mapping logical block addresses (LBAs) used by the host to the physical pages and blocks on the NAND chips (physical block address; PBA). This allows the host to access data on the SSD using logical addresses, rather than having to know the specific physical location of each block on the NAND chips. The data structure can simply be an array, where the index is LBA and its value is PBA. This address translation is necessary to ensure that data is correctly mapped to the physical locations within the NAND flash memory. The FTL (flash translation layer) acts as an intermediary to perform this translation.

Table 9-1. *Basic Mapping Table*

LBA	PBA
0	0x1243
1	0x3953
. .	. .
100	0x9324

This mapping table is stored in the RAM of the SSD for speed of access and is persisted in flash memory in case of power failure. When the SSD powers up, the table is read from the persisted version and reconstructed into the RAM. The simple approach is to use page-level mapping to map any logical page from the host to a physical page. This mapping policy offers a lot of flexibility, but the major drawback is that the mapping table requires a lot of RAM, which can significantly increase the manufacturing costs. A solution to that would store only the part of the table required to service the read request from the host in RAM. The disadvantage of this approach would be needing to read from NAND (on demand) if the host

read does not have a mapping table in RAM. This will have an impact on random read performance.

A logical-to-physical block address table (mapping table) is an essential component of SSD firmware. It is used to translate logical block addresses (LBAs) used by the host system to physical block addresses (PBAs) on the SSD. The mapping table is necessary because the physical blocks on an SSD may wear out or become faulty over time, and the firmware must be able to remap logical blocks to new physical blocks to maintain the integrity of the data.

Size of the Mapping Table

The size of the mapping table depends on the capacity of the SSD and the addressing scheme used. In larger-capacity SSDs, the mapping table can be substantial due to the increased number of LBAs and corresponding PBAs. For example, a mapping table for a multi-terabyte SSD can contain millions of entries.

Size of SSD: 128 GB

Number of clusters: Assuming each cluster is 4 KB (4 kilobytes), let's calculate the number of clusters:

134,217,728 KB (SSD size) / 4 KB (cluster size) = 33,554,432 clusters

Assuming each mapping table entry requires 4 bytes to store the corresponding PBA (physical block address), we can calculate the total RAM size required for the mapping table as follows:

Total RAM size required for mapping table = Number of clusters * Number of bytes required to store the PBA

Total RAM size required = 33,554,432 clusters * 4 bytes = 134,217,728 bytes

Therefore, for an SSD with a size of 128 GB and a cluster size of 4 KB, the mapping table would require approximately 134,217,728 bytes or 134 megabytes of RAM to store the mapping entries.

Storing the Mapping Table in RAM

Ideally, it would be advantageous to store the entire mapping table in random access memory (RAM) for fast access. However, due to the limitations of RAM capacity in most SSD designs (due to cost), it is often impractical or impossible to load the complete mapping table into memory. Instead, SSD firmware employs strategies to optimize the storage of the mapping table. For example, a mapping table for a multi-terabyte SSD can contain millions or even billions of entries.

Partial Loading of the Mapping Table

To overcome RAM limitations, the mapping table is typically loaded partially into RAM, focusing on the frequently accessed portions. The FTL prioritizes loading the mapping entries required for active LBAs, ensuring efficient and quick access to frequently accessed data. This partial loading strategy allows the SSD to maintain acceptable performance while conserving valuable RAM resources.

Storage of Non-Loaded Mapping Entries

The mapping entries that are not loaded into RAM reside in the NAND flash memory. These entries are accessed on an as-needed basis. When an LBA that is not in the loaded portion of the mapping table needs to be accessed, the FTL utilizes algorithms to locate the corresponding mapping entry in the NAND flash memory. This retrieval process may introduce some additional latency due to the need to access the slower NAND storage.

Write/Update Operations and the Mapping Table

During write/update operations, the mapping table undergoes modifications to accommodate new LBAs and PBAs that result from data writes, garbage collection, and wear leveling. To optimize the write/update process, SSD firmware employs various techniques, including maintaining a dirty cache buffer of the mapping table in RAM.

Dirty Cache Buffer in RAM

A common approach is to utilize a portion of RAM as a cache buffer for the mapping table. This buffer temporarily holds the mapping table entries, which are modified before they are flushed back to the NAND flash memory. The dirty cache buffer allows for efficient and quick updates without constantly writing to the NAND, which can be time-consuming.

Write/Update Process with Dirty Cache Buffer

When a write/update operation occurs, the SSD firmware first checks the dirty cache buffer in RAM. If the mapping table entry for the specific LBA already exists in the dirty cache buffer, it is updated directly in RAM, avoiding unnecessary writes to the NAND flash memory. This approach reduces latency and improves overall performance.

Flush to NAND

To ensure data durability and to prevent loss in the event of a power failure or system crash, the contents of the dirty cache buffer need to be periodically flushed back to the NAND flash memory. This flushing process involves writing the modified mapping table entries from the dirty

cache buffer to their corresponding locations in the NAND. The frequency of flushing can vary based on factors such as the size of the dirty cache buffer and the SSD firmware's internal policies.

Mapping Table Management and Optimization

As the SSD operates, the mapping table undergoes continuous updates to accommodate new LBAs and PBAs resulting from write operations, garbage collection, and wear leveling. Efficient management of the mapping table involves carefully balancing the usage of RAM resources, the frequency of flush operations, and the optimization of write/update processes. SSD firmware employs various techniques, like buffering, compression, and intelligent mapping algorithms, to optimize mapping-table management, reduce write amplification, and improve overall SSD performance.

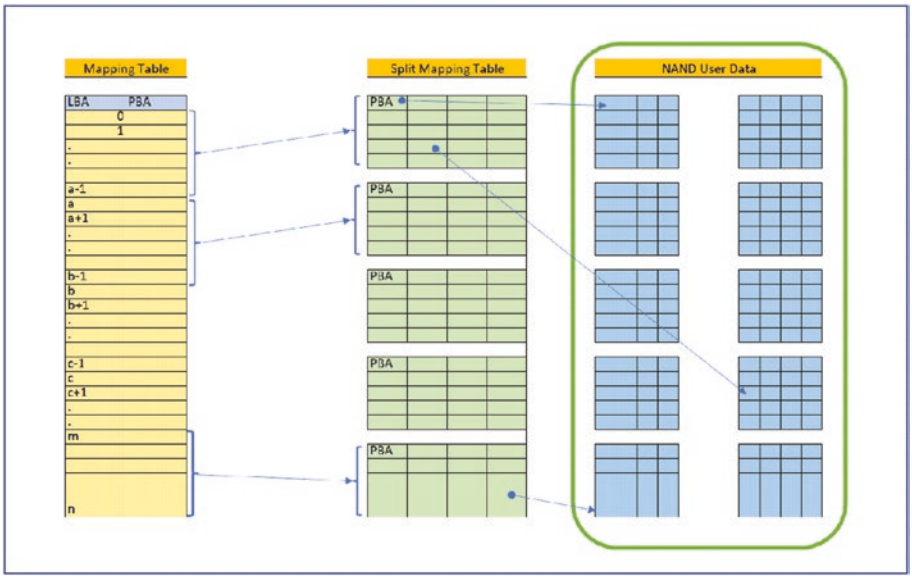

Figure 9-2. *Multi-level mapping table*

The following is a step-by-step guide on how the mapping table is created, accessed, and updated in the erase, read, and write path of an SSD:

Initialization: When the SSD is first initialized, the firmware creates a blank mapping table. This table consists of a series of entries, each of which maps a logical block address to a physical block address. Initially, all of these entries are set to a default value, indicating that the logical block has not yet been mapped to a physical block.

Table 9-2. *Mapping Table: Init*

LBA	PBA
0	0xFFFF
1	0xFFFF
. .	. .
100	0xFFFF

Write: When the host system writes data to the SSD, it sends a write command to the SSD along with the LBA and the data to be written. The firmware receives this command and determines which physical block to write to. Then, it sends the data to be written to that block and updates the mapping table accordingly.

FTL performs a process called *block allocation*, which involves selecting a suitable physical block to store the data and updating the mapping table

to reflect the new mapping. This process takes into account factors such as wear leveling, bad block management, and optimizing data placement to enhance performance and longevity.

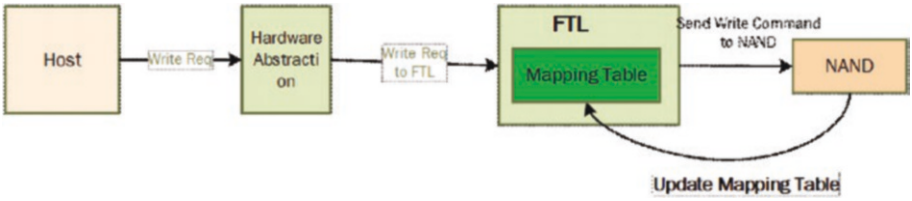

Figure 9-3. *Mapping table update during write path*

Table 9-3. *Mapping Table after Write*

LBA	PBA
0	0x1243
1	0x3953
. .	. .
100	0x9324

Read: When the host system reads data from the SSD, it sends a read command to the SSD along with the LBA of the data to be read. The firmware receives this command and looks up the corresponding entry in the mapping table. If the entry is set to the default value, the firmware returns an error to the host system indicating that the requested data is not present on the SSD (unmapped data). If the entry is set to a physical block, the firmware reads the data from that physical block and returns it to the host system.

Table 9-4. *Mapping Table While Read*

LBA	PBA
0	0x1243
1	0x3953
. .	. .
100	0x9324

Garbage Collection: As physical blocks on the SSD wear out or become faulty, during garbage collection the firmware may need to update the mapping table to remap logical blocks to new physical blocks. Figure 9-4 shows an example of how a physical block is written, unmapped, and moved to a new physical block and the mapping table being updated in parallel.

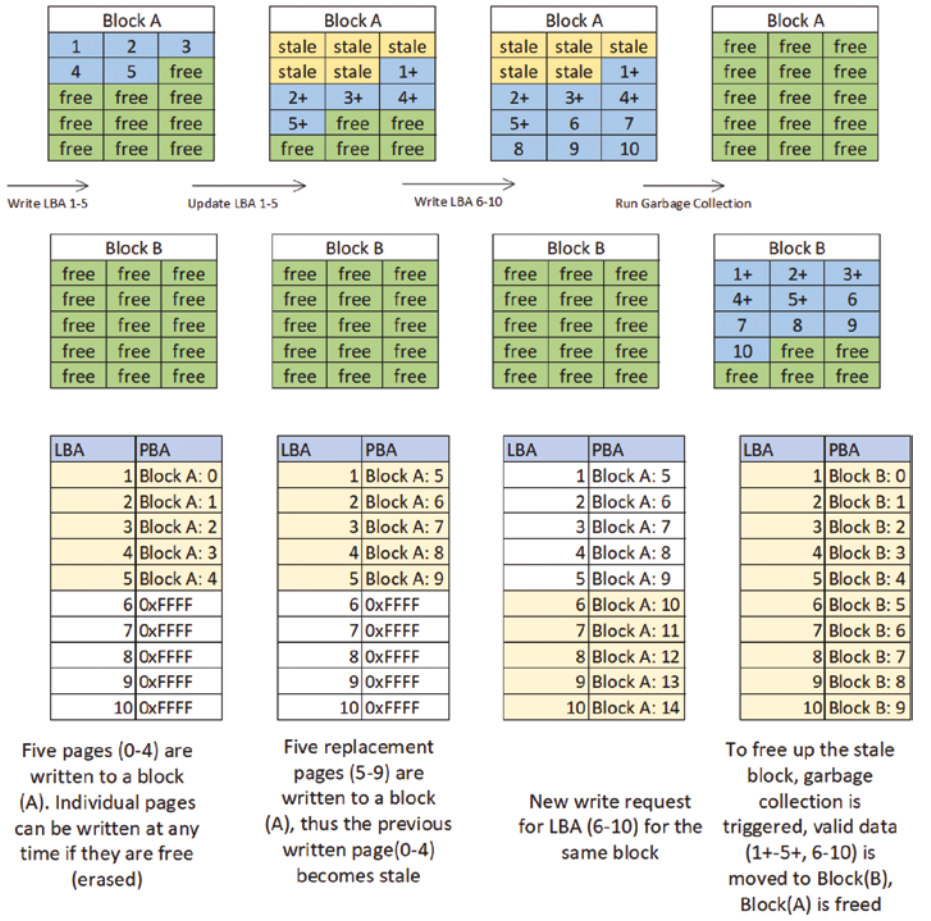

Figure 9-4. *Garbage collection for two SSD storage blocks—Block A and Block B—as they progress through the data update mapping table process*

Physical Erase/Sanitize/Format: When the SSD firmware receives an erase command, it selects the physical block specified in the command and erases it by setting all the bits in the block to 1. This allows the block to be overwritten with new data.

The firmware also updates the corresponding entry in the mapping table to reflect the fact that the logical block address is now mapped to an erased physical block.

Table 9-5. *Mapping Table after Physical Erase/Sanitize*

LBA	PBA
0	0xFFFF
1	0xFFFF
. .	. .
100	0xFFFF

Trim: When the SSD firmware receives a trim command, it marks the specified logical block address as no longer in use. This may involve updating the corresponding entries in the mapping table to set them to the default value, indicating that the logical blocks are not currently mapped to any physical blocks. The trim operation does not actually erase the physical blocks associated with the logical blocks; rather, it simply informs the SSD that these blocks are no longer needed and should be erased at a later time.

This can improve the performance of writing data to SSDs and help extend the lifespan of the SSD. `TRIM` is available for SSDs that support the Serial ATA (SATA) interface, while the `UNMAP` command serves a similar purpose for Small Computer System Interface (SCSI)

SSDs, and the DEALLOCATE operation performs a similar function in the nonvolatile memory express (NVMe) command set for Peripheral Component Interconnect Express SSDs.

The TRIM command works by enabling the operating system to proactively notify the SSD which data pages in a particular block can be erased. This allows the SSD's controller to manage the available storage space more efficiently for data. TRIM eliminates any unnecessary copying of discarded or invalid data pages during the garbage-collection process, which is an internal SSD housekeeping operation that manages and maintains available storage space by moving valid data pages to another block on the SSD so that the original block containing invalid data pages can be erased. By reducing the number of data pages that need to be moved during garbage collection, TRIM can reduce the number of program/erase cycles (P/E cycles) to the NAND flash media and extend the endurance of the SSD.

Using TRIM can provide benefits in terms of performance and drive longevity. It can speed up the write performance of the drive by avoiding unnecessary copying of invalid data and extend the lifespan of the drive by reducing the number of erase cycles.

Table 9-6. *Mapping Table after Trim*

LBA	PBA
0	0xFFFF
1	0x3953
. .	. .
100	0x9324

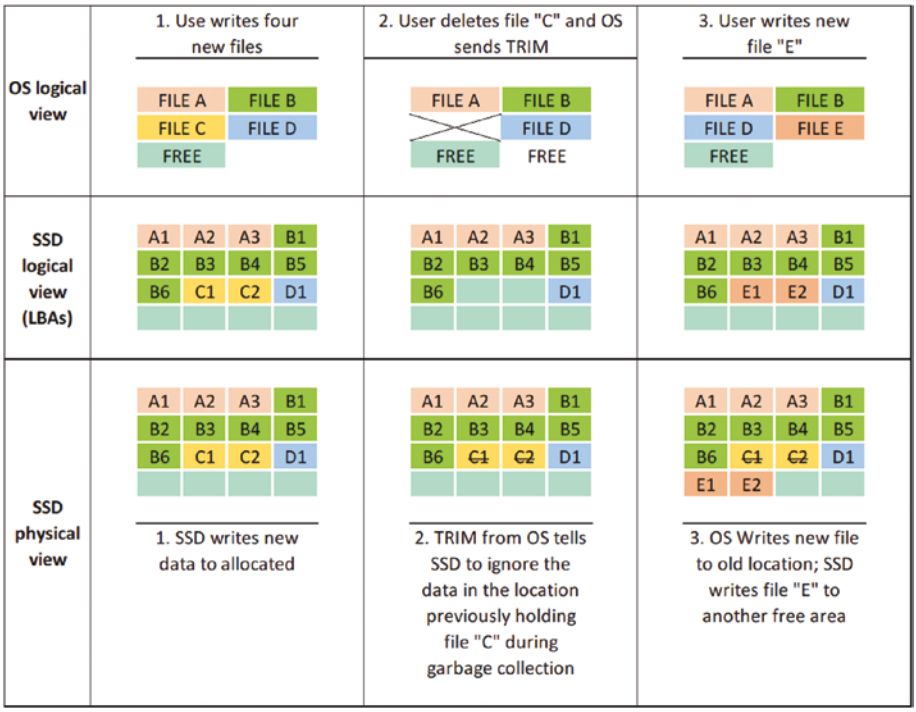

Figure 9-5. *Trim execution flow from host*

Bad Block Management

Bad blocks on an SSD can be a major problem, as they can prevent the device from functioning properly and may result in data loss. To address this issue, the firmware on an SSD includes a feature called bad block management, which is responsible for identifying and remapping bad blocks on the NAND chips, which are blocks that can no longer be reliably written to or read from due to physical defects or damage.

There are three types of bad blocks that the firmware may encounter:

1. ***Factory-marked bad blocks:*** Bad blocks (or initial bad blocks), that is, blocks that do not meet the manufacturer's standards or have been tested by the manufacturer and fail to meet the manufacturer's published standards, and have been identified as bad blocks by the manufacturer when they leave the factory.
2. ***Used bad blocks:*** Those that have become defective due to wear and tear during use, or that have reached the end of their lifespan.
3. ***False bad blocks:*** Those that are misjudged by the controller due to abnormal power failures or other issues.

Factory Bad Block Assessment

When a specific physical block in the NAND flash memory is detected as defective (bad block), the firmware must perform two fundamental activities: record the flash address of the bad block and update the bad block bitmap table.

Bad Block Flash Address

A bad block flash address contains essential information about the physical block that is considered defective. The exact format/content of this address depends on the NAND flash manufacturer. The firmware needs this information to translate the flash address information into meaningful data and to manage logical block mappings accurately.

Recording Bad Block Flash Address

The firmware must promptly record the flash address of the detected bad block. This information will be crucial in managing and avoiding future access to the defective block during normal read and write operations. The firmware should include protective measures to prevent any write or erase commands from targeting these identified defective blocks. Attempting to perform erase or program operations on such defective blocks will yield unpredictable and indeterminate results.

Initial Bad Block Handling Flow

When an SSD is powered up and mounted for the first time, the firmware performs the initial bad block handling to identify and manage any factory-marked defective physical blocks in the NAND flash memory. The goal is to ensure that these bad blocks are appropriately marked and avoided during subsequent read and write operations to maintain data integrity and optimize SSD performance.

Step 1: Power-Up and Mounting

The SSD is powered up, and the firmware initializes the device.

During the mounting process, the firmware initializes the bad block management mechanism, including the bad block bitmap table.

Step 2: Reading the NAND Flash

As part of the initialization process, the firmware reads each block in the NAND flash memory. The firmware checks for any errors or anomalies during the read operation.

Step 3: Identifying Bad Blocks

If a read operation encounters a defective physical block (bad block), the firmware identifies it as a bad block and records the flash address of the bad block in a bad block bitmap.

Step 4: Updating Bad Block Bitmap

After identifying a bad block, the firmware updates the corresponding entry in the bitmap table, indicating that the block is defective.

Step 5: Skipping Bad Blocks

During subsequent read and write operations, the firmware checks the bad block bitmap table. When accessing data, the firmware will skip any blocks marked as bad in the bitmap table, effectively avoiding the defective physical blocks.

Step 6: Error Handling (Optional)

If the bad block causes any data corruption or errors during the read operation, the firmware may implement error correction techniques or take appropriate measures to ensure data integrity.

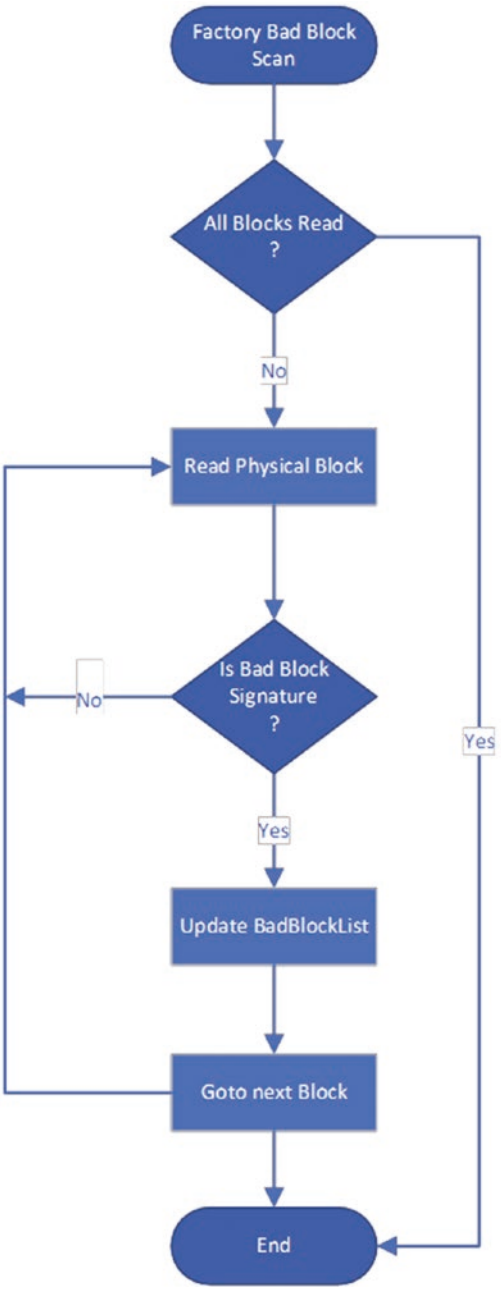

Figure 9-6. *Initial bad block scan flow*

Used Bad Block Assessment

Used bad blocks are those that have become defective due to wear and tear or that have reached the end of their lifespan. The firmware on an SSD is responsible for identifying used bad blocks and managing them to maintain the reliability and performance of the device. During program or erase actions, if the status register of the operation fails, the SSD controller will list this block as a bad block. Examples are as follows:

- An error occurred while executing the erase command.
- An error occurred while executing the write command.
- When the read command is executed, an error occurs; when the read command is executed, if the number of bit errors exceeds the error-correction capability of the ECC, the block will be judged as a bad block.

To keep track of bad blocks, SSDs have a feature called a bad block able (BBT), which is typically stored in a separate area of the NAND memory. The BBT is read after each power-up to make it more efficient, and it may also be backed up to protect against damage to the NAND memory. The number of copies of the BBT that are backed up may vary depending on the specific design strategy, with some SSDs backing up with as many as eight copies. Figures 9-7 and 9-8 show basic (not the only way) handling for used bad blocks.

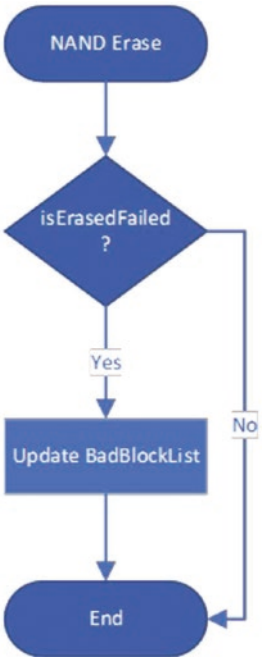

Figure 9-7. *Handling bad block during erase operation*

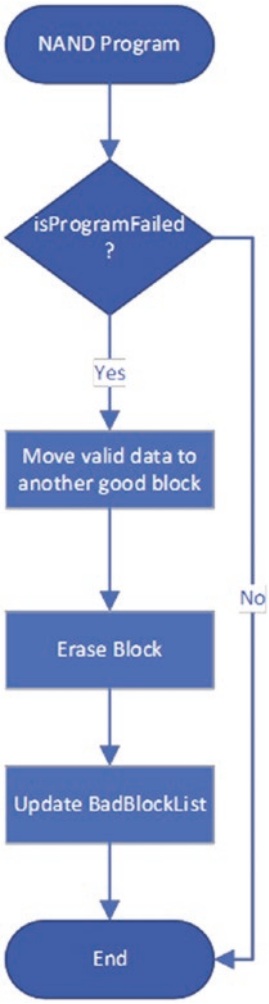

Figure 9-8. *Handling bad block during NAND program operation*

There are generally two approaches to managing bad blocks: the bad block skip strategy and the bad block replacement strategy. The bad block skip strategy involves simply skipping over any bad blocks and not using them, while the bad block replacement strategy involves replacing bad blocks with good ones. Both approaches have their own benefits and drawbacks, and the choice of which to use may depend on the specific requirements of the SSD.

Bad Block Skipping Strategy

1. For the initial bad block, the bad block skip will skip the corresponding bad block through BBT and directly store the data in the next good block.
2. For the new bad block, update the bad block to the BBT, transfer the valid data in the bad block to the next good block, and skip this bad block every time you do the corresponding read, program, or erase in the future.

Bad Block Replacement Strategy

In general, the OP (over provision)-area free block is used to replace the new block during use. Take garbage collection as an example. When the garbage-collection mechanism is running, the valid page data in the block that needs to be reclaimed is first moved to the free block, and then the erase operation is performed on the block. It is assumed that the erase status register is fed back at this time. When the erase fails, the bad block management mechanism will update the block address to the new bad block list, and at the same time write the valid data pages in the bad block to the free block in the OP area. It will update the bad block management table, and next time when writing data, it will skip the bad block and go directly to the next available block.

The OP size varies from manufacturer to manufacturer; there are different application scenarios, different reliability requirements, and different OP sizes. There is a trade-off between OP and stability. The larger the OP, the larger the available space for garbage collection in the process of continuous writing, the more stable the performance, and the smoother the performance curve. Conversely, the smaller the OP, the worse the performance stability, the larger the available space for users, and the lower the cost.

Generally speaking, OP can be set to 5 percent to 50 percent. An OP of 7 percent is a common ratio. Unlike the 2 percent fixed block suggested by the manufacturer, 7 percent is not a fixed block for OP. Instead, it is dynamically distributed among all blocks, which is more conducive to the wear-leveling strategy.

Summary

In summary, the FTL is a critical component of the firmware in a NAND-based SSD, and it plays a vital role in managing the interaction between the host and the NAND chips. It is responsible for ensuring that data is stored and retrieved efficiently, and it helps to maintain the performance and reliability of the SSD over time.