NXP Semiconductors
Application Note

Document Number: AN4868
Rev. 2, 03/2017

EEPROM Emulation with Qorivva MPC55xx, MPC56xx, and MPC57xx Microcontrollers

by: David McDaid

Contents

1 Introduction

Electrically Erasable and Programmable Read Only Memory (EEPROM) is often used in applications where runtime-modifiable data needs to be retained after the power has been removed. Although the Qorivva MPC55xx, MPC56xx, and MPC57xx family of microcontrollers do not include EEPROM, the functionality of this memory type can be emulated with the flash memory that is available. This document describes the basic principles behind using Qorivva flash memory for EEPROM emulation, and provides some insight on the functionality of NXP EEPROM emulation drivers.

2 EEPROM and Flash Memory Characteristics

EEPROM and flash memory are both non-volatile memories, meaning they retain their data after power has been removed. They are both composed of an array of memory cells, where each memory cell holds one or more bits of information. In some designs, multiple cells combine to form a single bit.

For the NOR flash memory on Qorivva devices, each memory cell stores one bit of information. In EEPROM and flash memories where each memory cell holds one bit of information, the default, erased state of the memory cell reads

as binary 1. Programming a memory cell changes its value from binary 1 to binary 0, and erasing the memory cell changes its value from 0 to 1. Programming does not work in the reverse direction: only an erase operation changes the value of a memory cell from 0 to 1.

EEPROM is typically programmed and erased by byte or word. Since the program and erase operations both occur on a byte/word boundary, application variables stored in EEPROM can be updated individually without disturbing other values. The values stored in EEPROM may be written frequently during the device lifetime, so EEPROM is usually designed to support a high number of program-erase (PE) cycles.

The flash memory on Qorivva devices supports word or double-word program sizes, with larger program sizes also available. However, as with all flash memories, erasures can only happen by block, where each block is often several kilobytes in size. This restriction on the erase mechanism greatly reduces the silicon area required, allowing much larger flash arrays (in terms of available memory) to be created as compared to EEPROM. Flash memory is typically used for storing constant data such as main application code, and this usage typifies the usual characteristics of flash memory—long data retention time with a low program-erase endurance requirement.

3 EEPROM Emulation Using Flash Memory

There are some challenges involved in using flash memory for the storage of data typically placed in EEPROM:

- Modifying a value stored in flash requires erasing an entire block of memory.
- Applications with frequent data updates must be mindful of the PE cycle specification.

These challenges are overcome by constructing a record-based file system within flash memory. This file system can be thought of as a list of records, where each record is a grouping of one or more variables. Every record also has a tag that identifies the variables contained within the record, the size of the record, and whether the record is valid or not.

A set of flash blocks are designated as the EEPROM emulation blocks. From this set of blocks, a single block is chosen as the active block. Records are programmed into sequential memory locations in the active block. To update a record, the software adds the updated version of the record to the next available location in the active block. Reading a record is accomplished by retrieving the most recently written record (that is, the last record) with a matching tag.

When the active block becomes full, the software copies all the valid records within the active block into one of the other EEPROM emulation blocks. This new block becomes the active block, and the previous active block is then erased. Since this block-swap process purges outdated records, the new active block will have space for further record updates.

3.1 Record structure

Regarding flash memory as a grid of bytes, the layouts of two generic record structures are shown in the figures below (ECC boundaries are explained in ECC and record status). Records are separated into two main segments: a data segment, where the application variables are held, and a metadata segment, which contains all the information needed to manage and retrieve data from the record. The metadata segment is further broken down into additional fields:

- A record status field. This field is used for recovering from errors and for maintaining data coherency.
- An ID field. This field is used to identify the data contained within the record.
- A size field. This field is used to determine the number of bytes of data within the data segment.

The length of the ID and size fields can change, depending on the application. However, the length of the record status field must fit a certain criteria, as explained in the next section. Additional fields can be added to the metadata segment, but the EEPROM emulation scheme detailed in this document only uses the three fields listed above.

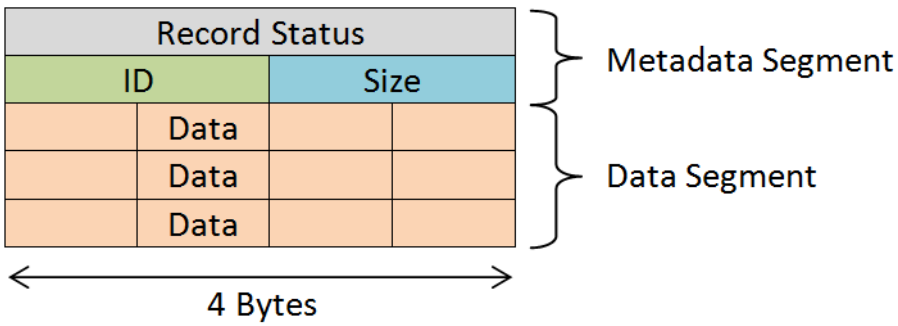

Figure 1. Record structure with 4-byte ECC

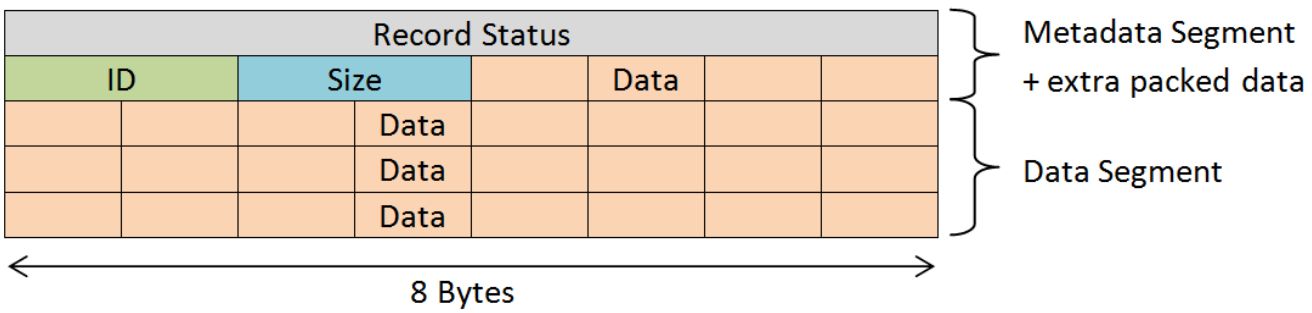

Figure 2. Record structure with 8-byte ECC

3.2 ECC and record status

An Error Correcting Code (ECC) is used alongside memory devices to correct or detect memory corruptions due to radiation or electromagnetic interference. The flash memory on Qorivva devices employs a single-bit correction, double-bit detection ECC algorithm. A single bit error within an ECC boundary is automatically corrected and the MCU notified of the correction; two bit errors within an ECC boundary are uncorrected, but the MCU will be notified of their existence.[1]

ECC works by computing a fixed number of parity bits on the data written into flash memory. At a later time when the data is read from flash memory, the data bits are combined with the parity bits and any errors are handled appropriately. On Qorivva devices, every 4- or 8-byte boundary is designated as a checkbase and has its own set of parity bits (the checkbase size is device specific, as shown in the table below).

In the EEPROM emulation scheme, the record status field is reprogrammed with different values depending on the state of the record. Reprogramming flash memory without an erase is normally not allowed, but this action is permissible on Qorivva devices when changing the record status field. If Qorivva devices did not include ECC, any word in flash memory could be reprogrammed as long as the modification only involves changing binary 1s to 0s (remember, turning a 0 into a 1 involves erasing the entire block). In this way, each bit in the record status field is treated as a write-once flag. For example, a single word could be reprogrammed as follows:

```
0xFFFF_FFFF → 0xFFEF_3FFF → 0x0FEE_1ABC → 0x02AE_1AB0
```

However, Qorivva parts do include ECC, so reprogramming a value also attempts to rewrite the ECC parity bits within the checkbase. Even if the data bits only change from 1 to 0, the new data will likely require changing a parity bit from 0 to 1. Since the parity bits are part of flash memory, the program operation will fail. This normally poses a problem when

1. ECC error events for MPC57xx data flash blocks are not reported to the Memory Error Management Unit (MEMU) / Fault Collection and Control Unit (FCCU). However, the flash Module Configuration Register (MCR) still uses the ECC Event Error (EER) bit to report double-bit errors for these blocks.

attempting to modify the record status; fortunately, the Qorivva ECC algorithm allows certain sequences of data values. Whenever a checkbase is reprogrammed, as long as the modification only clears groups of 8/16 bits, the operation will succeed (the group size is device specific, as shown in the table below). For example, a device with a 8-byte ECC checkbase that allows clearing groups of 16 bits can be reprogrammed with the following sequence of double-words:

```
0xFFFF_FFFF_FFFF_FFFF → 0x0000_FFFF_0000_FFFF → 0x0000_0000_0000_FFFF
```

By taking advantage of this property, a single word/double-word record status field can be updated several times and hold multiple status flags, ultimately reducing the record status field size and allowing more application data to be placed in flash memory.

Table 1. ECC on Qorivva devices

Device	Checkbase size	1/0 bit groupings	Example value
MPC55xx	8 bytes	16 bits	0xFFFF_0000_FFFF_0000
MPC56xx (instruction flash)	8 bytes	16 bits	0xFFFF_0000_FFFF_0000
MPC56xx (data flash)	4 bytes	8 bits	0xFF00_00FF
MPC57xx	8 bytes	8 bits	0xFFFF_0000_FF00_00FF

There is also an important feature to consider within the Qorivva parts, which is the dedicated ‘EEPROM Emulation’ flash blocks that are made available to the programmer. The available blocks are clearly labelled within each device’s Reference Manual. They may often appear in a diagram like the one shown in Figure 3, or listed within the flash segmentation features.

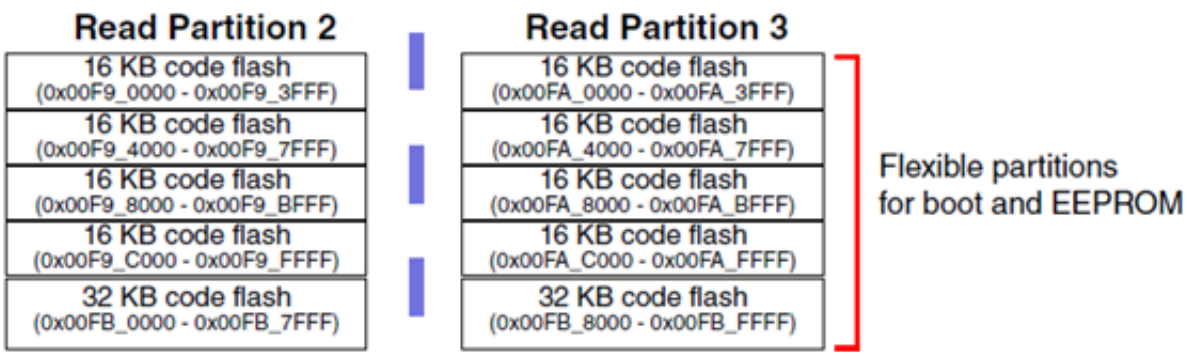

Figure 3. EEPROM flash partitions labelled within MPC5748G Reference Manual

It is strongly recommended that the partitions described as 'EEPROM Emulation' in the Reference Manual are used for EEPROM emulation schemes. This is recommended because the flash controller does not propagate the clock for the ECC on these blocks. This correlates to the following events:

1. It reduces the potential SW overhead by removing the need to account for ECC alignment (see previous paragraphs covering this).
2. It guides the programmer away from the potential to use larger flash blocks which command larger erase times.
3. In certain emergency situations, such as a brown-out there would be no ‘false’ ECC errors reported if an EEPROM operation was unsuccessful.

3.3 Adding and updating records

Before adding a new record to the active block, the EEPROM emulation software locates the first erased memory location in the block. Records are written into the active block sequentially, so the software knows that the remaining memory locations are also erased. From this location, the software then determines if there is enough space available to hold the record. This

free-space check must factor in the size of the metadata segment. If sufficient space is available, the software writes the record into flash. If there is not enough free space, the software first executes a block swap (described in Swapping blocks) before writing the record.

The actual write process is broken down into the following steps:

1. The software writes the record status as $invalid.
2. The software writes the ID and size fields, as well as any other data packed into the same ECC checkbase.
3. The software writes the remaining data.
4. The software writes the record status as $valid.

This multistage operation is designed to ensure that partially programmed, corrupt data is never treated as good data if the programming process is interrupted. The $valid and $invalid values are chosen such that they follow the ECC rules as explained in the previous section. The table below gives suggested values for the status field. Note that before the $invalid value is written, the record status is considered to be $empty, which is the value of the erased memory.

Table 2. Record status field constants

Status	4-Byte ECC boundary	8-Byte ECC boundary
$empty	0xFFFF_FFFF	0xFFFF_FFFF_FFFF_FFFF
$invalid	0x0000_FFFF	0x0000_0000_FFFF_FFFF
$valid	0x0000_0000	0x0000_0000_0000_0000

To update a record already written into flash memory, the software simply adds the updated record to the end of the record list. When searching for a record with a particular ID, the software knows that the last record with a matching ID is the most up-to-date record.

The figure below shows an example of the active block memory contents after several records have been written. In this example, the device was reset while adding the second record. Since the record addition process did not complete, the second record has a status of $invalid. The flash block status field at the start of the flash block is described in Swapping blocks.

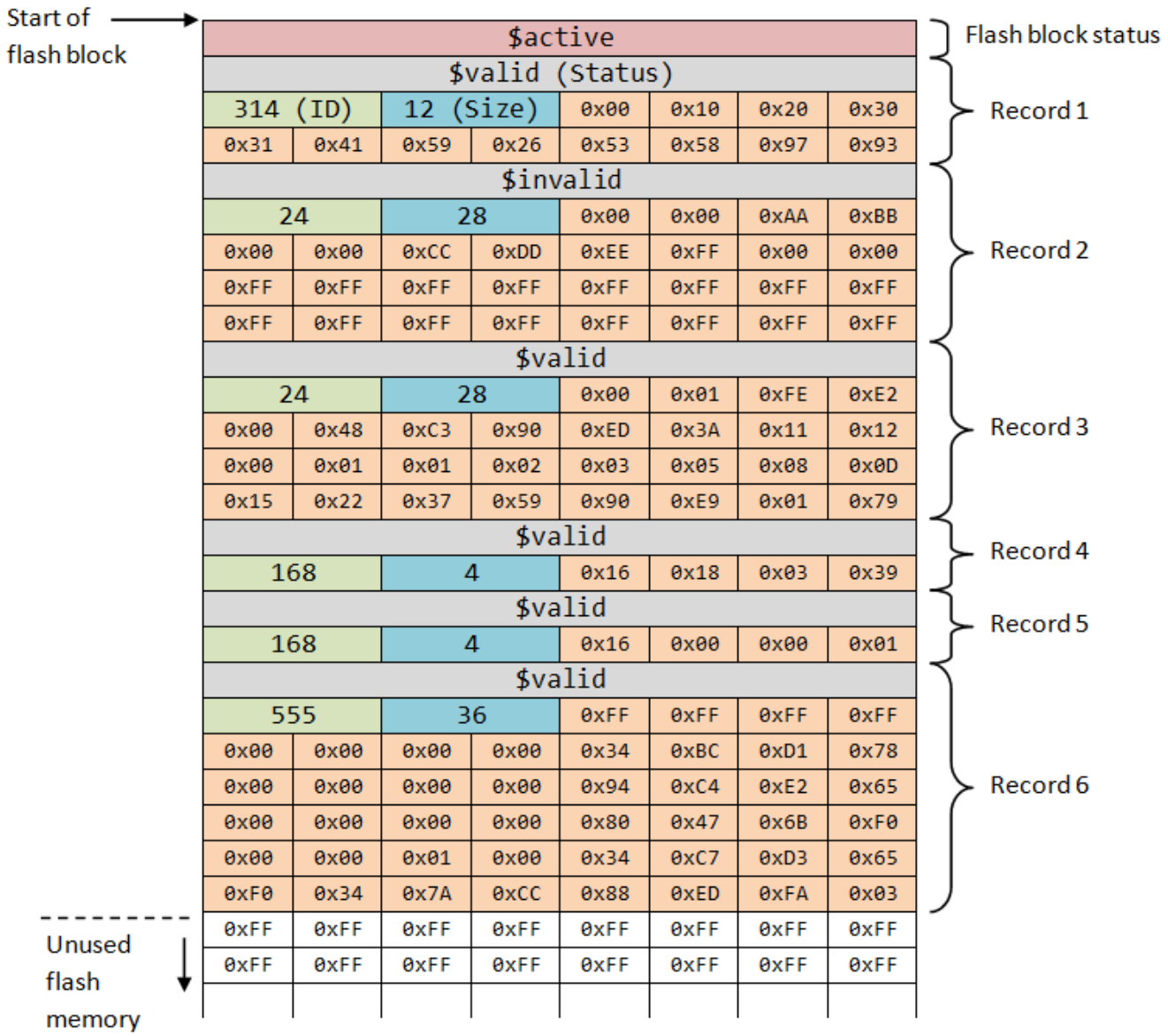

Figure 4. Example record list

3.4 Reading records

When the main application software requests to read a record, the EEPROM emulation software must locate the record. The main application software supplies the record ID, and the EEPROM emulation software walks through the list of records, comparing each record ID along the way. The EEPROM emulation software returns the address of the last valid record with a matching ID.

Walking through the list of records uses the following steps, beginning with the first record in the block:

1. The EEPROM emulation software checks the record status. If the record status is $empty, the software is at the end of the list and returns the address of the most recent matching record (if any).
2. The software checks the ID of the record. If the ID matches the ID of the requested record, and the record status is $valid, the software makes note of this record. Because multiple records with the same ID may exist in the active block at any one time, the software must continue walking through the list until it reaches the end.

3. The software reads the size of the record and uses it to calculate the address of the next record in the list.
4. The software moves to the next record, starting the process over again at step 1.

During a block swap operation or when adding records, the EEPROM emulation software walks through the record list in a similar manner. In the case of a block swap, the record ID is ignored since the software is only concerned with finding valid records. In the case of adding records, the software walks through the record list only to find the end of the list.

3.5 Recovering from an interrupted record addition

Certain events, such as removing power from the device, can interrupt the record addition process at any point in time. Proper use of the record status field allows the application to recover gracefully, even if volatile context information is lost. Whenever the EEPROM emulation software walks through the record list, it skips over records that are marked as $invalid. Records that are marked as $invalid are purged from the record list during the block swap process, as they are not written into the new flash block. The table below pairs each stage of the record addition process with the state of the record after each stage completes.

Table 3. Record state during the addition process

Stage	Operation	Record status	ID/Size field
A	State of the record before any writes take place.	$empty	blank
B	Software writes the record status as $invalid.	$invalid	blank
C	Software writes the ID and size fields, as well as any data packed into the same checkbase.	$invalid	written
D	Software programs the remaining record data. This requires multiple writes, depending on the size of the data.	$invalid	written
E	Software writes the record status as $valid	$valid	written

The $invalid record status serves two purposes: to mark the presence of a corrupt record, and to protect against marginal write errors. Marginal writes occur whenever the system is interrupted [2] at the exact point in time when the system programs a value into flash memory. In some cases, this memory location reads back as erased (the bits are binary 1), even though the true state is somewhere between programmed and erased. The system can either reprogram this memory location with the same value as the previous attempt, or erase the flash block and start over.

Unfortunately, if a memory location reads back as erased, the system does not know if this is the expected value or if this memory location is marginally written. If this location is marginally written, and the system attempts to program this location with a different value, the operation will likely fail. The EEPROM emulation software combats marginal writes by using the same $invalid value whenever it adds a new record. If the software is interrupted before stage B completes (described in the table above), the $invalid value is marginally written. Since this same exact value is written the next time the software attempts to add a record, the operation will complete successfully.

When skipping $invalid blocks, the EEPROM emulation software looks at the record size to know where the next record sits in memory. If the software is interrupted before stage C completes, the record ID and size are not available. For this reason, the software treats any record with a blank size field as a record holding zero bytes of data. Since none of the record data has been written, the software uses the word/double-word (depending on the checkbase size) immediately after the ID and size fields as the start of the next record. After stage C completes, the software recovers from any interruptions by skipping over the incomplete, $invalid record.

2. Interruptions occur primarily by a reset.

3.6 Swapping blocks

After a number of record additions and updates, there will not be sufficient erased memory in the active block to perform further updates. At this stage, the EEPROM emulation software performs a block swap by copying the valid records from the active block to an alternate block.

Whenever the software erases a block, the software subsequently verifies that the block has been erased by checking that every memory location in the block reads back as all 1s. Any block which is verified as erased may be used as the alternate block in a block swap. Verifying erased blocks prevents the software from using a block that is not fully erased[3] as an alternate block. Attempting to program a block that is not fully erased will likely result in an error.

When the alternate block is selected, the software reads through the contents of the active block and copies the most recent record for each ID into the alternate block. The alternate block is then marked as the active block for any subsequent access, and the software erases and verifies the previous active block in preparation for future use.

Each block reserves a word/double-word at the start of the block (as shown in the figure in Adding and updating records) to hold the block status. The block status field is used in the same way as the record status field, and cycles sequentially through four possible states:

1. $erased - default value of an erased block (all 1s)
2. $verified - block has been verified as erased
3. $copy - block is being used as the alternate block in a block swap
4. $active - block is used as the active block

As with the record status field, the immediate values $verified, $copy, and $active must be specifically chosen so that the ECC rules are obeyed. The table below provides some suggested values.

Table 4. Suggested block status field constants

Status	4-Byte checkbase, groups of 8 bits	8-Byte checkbase, groups of 16 bits
$erased	0xFFFF_FFFF	0xFFFF_FFFF_FFFF_FFFF
$verified	0x00FF_FFFF	0x0000_FFFF_FFFF_FFFF
$copy	0x0000_FFFF	0x0000_0000_FFFF_FFFF
$active	0x0000_00FF	0x0000_0000_0000_FFFF

The following table shows the different stages of the block swap operation, along with the status of each block involved in the process. Stage A represents the normal operating condition where flash block A is the active block. The other stages are transient stages that only exist during the block swap process. If the block swap process is interrupted at any point, the EEPROM emulation software is able to recover and identify which block to use for subsequent accesses and updates.

Table 5. Flash block status during block swap operation

Stage	Block A status	Block B status	Operation
A	$active	$verified	Initial status with block A active and block B verified erased.
B	$active	$verified → $copy	Software sets the status field of block B to $copy, indicating the start of a block swap operation
C	$active	$copy	Software copies the active records from block A into block B.
D	$active	$copy → $active	Software finishes copying the active records from block A into block B. Software sets block B as the new active block.
E	$active → $erased	$active	The software erases block A.
F	$erased → $verified	$active	Software verifies that block A has been erased and sets the status of block A as $verified.

3. This can occur, for example, if the device resets while the block is actively being erased.

3.7 Recovering from an interrupted block swap

As with record additions, it is possible to recover from a situation where the block swap operation has been interrupted and all volatile context information is lost. The software treats any block whose status is not $verified or $active as being in a faulty state, and any block in this condition should be erased and verified blank. Refer to the previous table for a description of the stages used in this section.

If the block swap process is interrupted at either stage B or C, flash block A remains marked as the active block, allowing records to still be accessed correctly. Flash block B, marked as $copy, contains potentially corrupt or partially programmed data records. To return block B back to a known, useful state, the EEPROM emulation software erases block B and verifies that it is blank. The software then restarts the block swap process.

If the block swap process is interrupted at stage D, both blocks are marked as $active. The software identifies which block has the largest amount of unused (that is, erased) flash memory and uses that block as the active block. The other block is completely or nearly completely full, being the original reason the block swap process was required. The software completes the block swap process by erasing the full block and verifying that it is blank.

If the block swap process is interrupted at stage E or F, block A is in a partly erased state. This state can be detected by illegal block/record status values, or by the ECC errors that are likely to occur. Recovery consists of re-erasing block A and verifying that it is blank.

4 Read-While-Write

The system is unable to read certain parts of flash memory whenever a program or erase operation is in progress. The EEPROM emulation software needs to be aware that adding a record or performing a block swap limits the range of accessible memory locations. Qorivva devices include two mechanisms that allow the system to fetch flash-resident application code during an ongoing program or erase operation: hardware read-while-write (RWW) partitions, and the capability to suspend and resume a flash operation.

4.1 Hardware read-while-write partitions

Qorivva flash memory supports true RWW operations between flash memory partitions. A flash memory partition is a collection of flash blocks that share the same control circuitry. When one block in a flash memory partition is being erased or programmed, blocks from other partitions can be simultaneously read. The flash blocks within the same partition cannot be read while the program or erase operation is in progress. The main application software is typically placed in a different flash memory partition than the EEPROM emulation blocks so that fetching the main application software is not interrupted when updating records or doing a block swap.

The flash blocks used for EEPROM emulation can also span multiple partitions. Alternating the active block partition is especially useful for reducing the read latency during a block swap. While records are written into the new flash block, the system can still retrieve records from the full, active block. Once all the valid records are written to the new flash block, software can then read records from the new block while the old block is being erased.

The flash memory partition boundaries are device specific, and the layout for each device is contained in the device reference manual. The figure below shows an example flash block arrangement. In this scenario, the 16 KB or 48 KB blocks in partition 0 could be used for EEPROM emulation while the 128 KB blocks would hold the main application code.

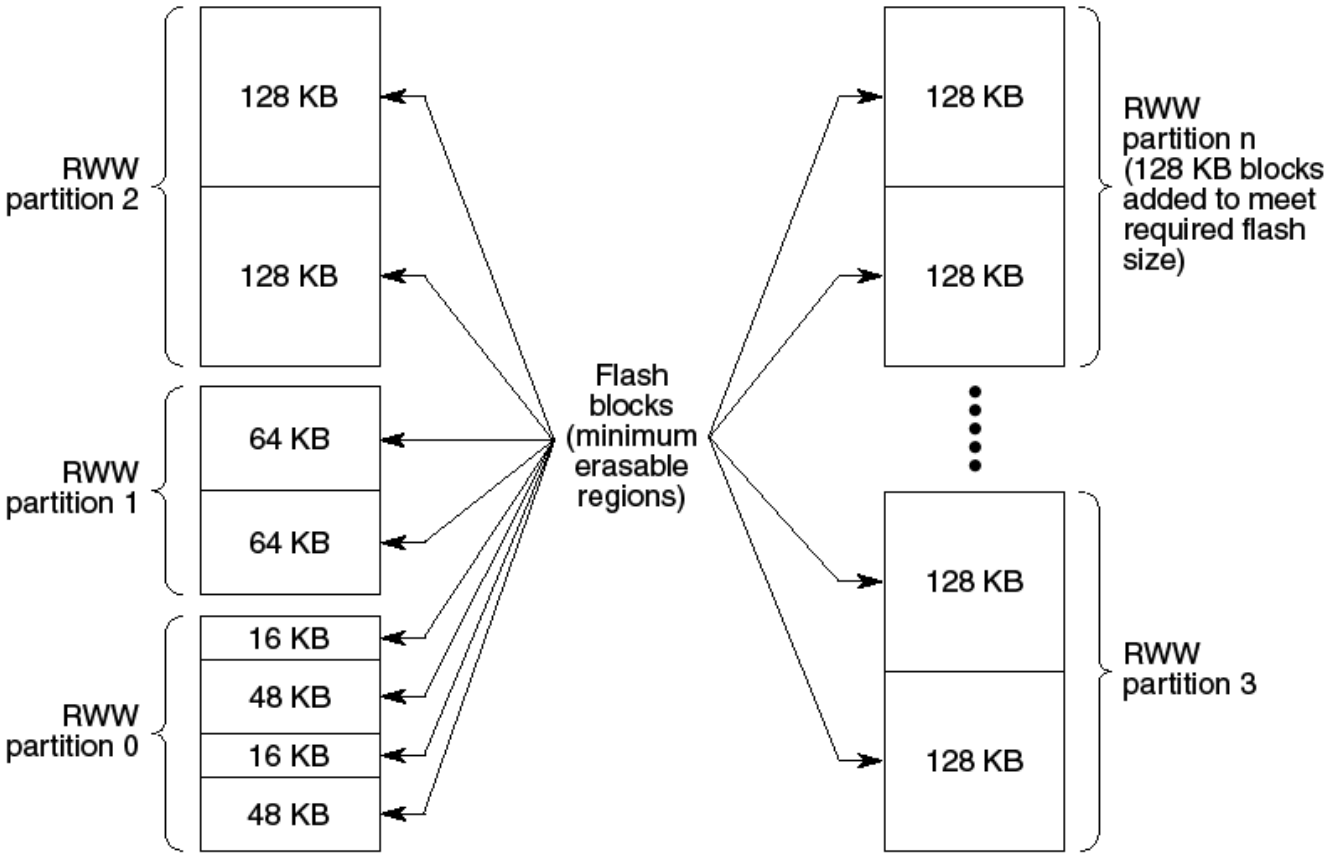

Figure 5. Example flash read-while-write organization

4.2 Suspend and resume

The flash controller on Qorivva devices also has built-in suspend and resume operations. Ongoing program and erase operations can be suspended, allowing read access to the entire flash memory array except for the block undergoing the program/erase operation (reads to the block being programmed/erased will return indeterminate data). Since reads are not truly concurrent to the program or erase operation, and since suspending a program or erase operation will impart a delay before the flash array can be read, it is recommended to take advantage of the hardware RWW partitions whenever possible.

It is recommended that suspending an erase should not be utilized frequently. This mechanism should only be used in situations which cannot wait for any on-going operation to complete; for example, during an unexpected event when the system is shut down with only a small-window available to store critical data. Performing frequent erase-suspend operations will put unnecessary stress on the Flash blocks.

It is also suggested that in applications where emergency writes cannot be fully avoided, then the user leaves a single Flash EEPROM block free for the purpose of dealing with these immediate write operations.

5 Software Implementation and Variations

The previous sections in this document provide an overview on how to use a record-based file system in flash memory to emulate EEPROM. The implementation provided in this document is used as an example to illustrate several key techniques when writing EEPROM emulation software:

- Grouping data into a list of tagged records
- Packing several records into a block to reduce the number of block erasures
- Proper handling of ECC when overwriting status fields
- Block swapping when the active block becomes full
- Manipulating records in a way which allows for recovery from unexpected interruptions

New applications can follow the example implementation outlined in this document or take advantage of the EEPROM emulation drivers provided by NXP. These drivers can be found through the NXP website by searching for the following keywords:

Table 6. NXP EEPROM emulation drivers

Device	Driver
MPC55xx	MPC55XX_EEPROM_EMU_DRIVER
MPC56xx	MPC56XX_C90LC_EE_DRV and C90FL_JDP_EEE_DRV
MPC57xx	MPC5700_C55_EEE_DEVD

NXP's drivers provide a thorough, complete set of features, and the driver documentation and source code can be referenced for the actual implementation. The following subsections highlight some implementation variations that can simplify or improve the performance of new EEPROM emulation software. Some of these variations are included in the drivers NXP provides.

5.1 Record cache

One possible feature to add to the EEPROM emulation software is the ability to cache records into RAM. This feature is included in NXP's drivers. Reserving an area in RAM to hold several of the most frequently used (or the most recently used) records offers several advantages:

- Reads and writes to RAM happen faster than the respective flash memory access.
- Record updates can happen in place.
- Moving more record manipulations into RAM means fewer block swap operations, which reduces the number of block erasures.

Adding this feature increases the complexity of the EEPROM emulation software as the software must now handle moving records between flash memory and RAM. Additionally, there must be enough notice of an imminent device shutdown so that the software can move any updated, cached records back into flash memory. Records that are cached in RAM should retain a copy in flash so that data loss is minimized if an error prevents moving updated, cached records back into flash memory.

Instead of holding the entire record in RAM, the software can construct a lookup table that holds the memory location of a record given the record ID. This reduces the time needed to search for a record (especially for blocks that are nearly full) since the software only needs to refer to the lookup table instead of traversing the entire record list.

5.2 Fixed record length

In some cases, the flexibility of the variable record length scheme is not required and a fixed record length scheme is sufficient. If every record in an application is the same length, or if a number of mixed-sized variables always combine to form a fixed length record, the record length field carries redundant information and can be removed. The software complexity involved in searching through fixed-length records is reduced since the records are known to always start at fixed offsets in memory. It is no longer necessary to read the size field of each record to compute the address of the next record.

5.3 Single flash block

Some applications may only have (or want) a single block available for EEPROM emulation. With only a single block available, the EEPROM emulation software must use a different block swap process. When the single block runs out of free space, the software copies the active, valid records into RAM before proceeding with the erase procedure. After the block is erased and verified, the software writes the records back into flash memory.

Using only a single flash block for EEPROM emulation poses a great amount of risk for data loss. During the block swap procedure, if a critical error occurs or if power is removed from the device before the software can write every record back into flash memory, the data in these records will be lost. For most applications, a minimum of two flash blocks is recommended so that a non-volatile copy of the data is always available to recover from such issues.

5.4 Cyclic redundancy check

In addition to hardware ECC, the EEPROM emulation software can include a cyclic redundancy check (CRC) to detect record errors. CRCs work by calculating a check value with a fixed number of bits over any data of interest. The next time this data is accessed, the check value is recalculated and compared to the previous value. The data is known to have changed if the two check values do not match. Many CRC specifications exist, with different specifications providing different levels of protection. In the simplest case, even parity is regarded as a CRC with a 1-bit check value which can detect if an odd number of bits change in a given segment of data.

As an example implementation, suppose the EEPROM emulation software uses the CRC-32-IEEE standard. Every time the software writes a record, the software uses the record ID, size, and data to calculate a 32-bit check value. The software then writes this check value into flash memory immediately after the record (while properly handling the ECC checkbase size), effectively appending the check value to the record. Whenever the software reads a record, it recalculates the check value. If the recalculated check value does not match the previous check value, then the software knows that the record ID, size, data, or check value has changed.

5.5 Persistent read access

The system can be designed to allow guaranteed, full-time read access to the records stored in flash memory. The basic operation of this scheme is detailed in the following figure. For every record, a copy of the record exists in a block from a different partition. Whenever a record needs to be updated, the EEPROM emulation software writes the update into both blocks. While the software is writing the update to one of the blocks, the record copy from the other block can still be read. Persistent read access does have a trade-off, though: since each record update requires adding two records, the update process takes longer to complete.

Two blocks from each partition (four blocks total) are used to ensure that a non-volatile copy of every record exists at all times. A block from each partition is selected as the active block, and the other block from each partition is used for the block swap process. With reference to the figure, suppose an additional block C exists in the same partition as block A, and a block D exists in the same partition as block B. Once blocks A and B become full, they are swapped with blocks C and D, respectively. The block swaps happen sequentially, so that while block A is swapped with block C, the records from block B can still be read. Similarly, once the block A to block C swap completes, the records from block C can be read while block B is swapped with block D.

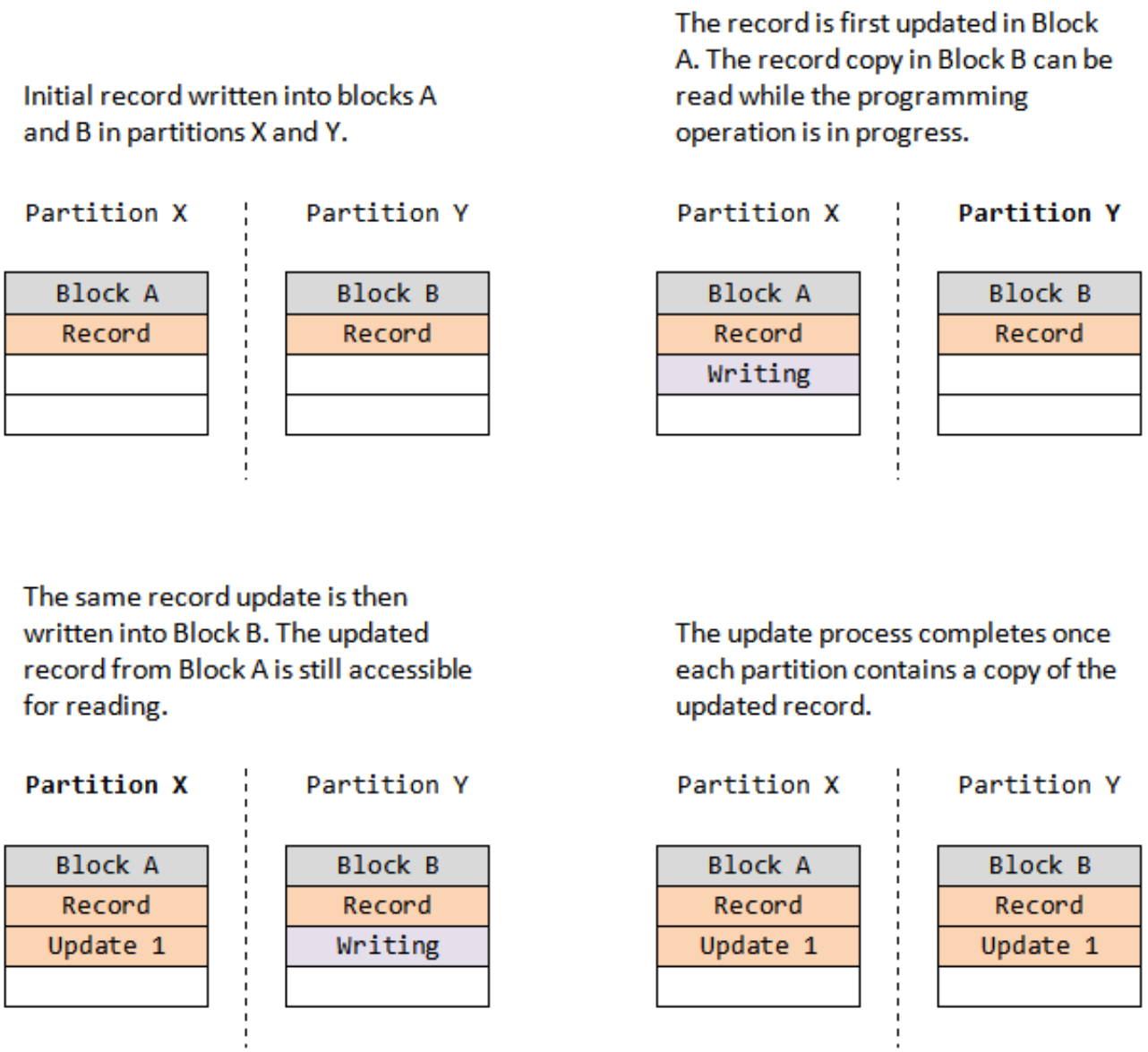

Figure 6. Continuously readable records

6 Program-Erase Endurance

The amount of EEPROM that can be emulated depends on the flash block size and the expected number of data accesses. As larger amounts of flash are used to implement a set amount of emulated EEPROM, the number of program/erase (PE) cycles seen by the flash memory is reduced. This allows EEPROM to be emulated with a higher PE endurance than the native flash specification.

For example, suppose the 16 KB flash blocks on a device are specified for a minimum 100,000 native program/erase cycles.[4] This means that a block is not guaranteed to work after being programmed then erased 100,000 times. However, this does not mean that only 100,000 record manipulations can occur. If the EEPROM emulated data only amounts to a single 256 byte record (including overheads such as the record status, ID, and size fields), this record can be updated in the 16 KB block 63

4. Refer to the device data sheet for the actual PE endurance specification.

times before the block needs to be erased (16 KB / 256 bytes = 64, but the block status field also consumes some memory). If the system uses two 16 KB blocks to hold EEPROM emulated data, this 256 byte record can be updated 12,600,000 times before the blocks reach 100,000 PE cycles. Using two 64 KB blocks specified for 100,000 PE cycles can sustain four times as many record updates.

As the required amount of emulated EEPROM data grows, the frequency of required block swap operations rapidly increases. The end user must estimate the maximum amount of emulated EEPROM based on inputs such as frequency of record updates and the PE endurance specification of the flash memory. Applications using the NXP EEPROM emulation drivers can use the NXP EEPROM emulation endurance calculators to estimate the flash block lifetime:

Table 7. NXP EEPROM endurance calculators

Device	Calculator
MPC55xx	MPC55XX_EEE_CALC
MPC56xx	MPC56XX_EEPROM_CALCULATOR
MPC57xx	C55_EEECALC

Applications can further improve the flash lifetime by using as many blocks as possible for EEPROM emulation. By cycling through several blocks in a round-robin manner, as shown in the following figure, the erasures on each block are further reduced.

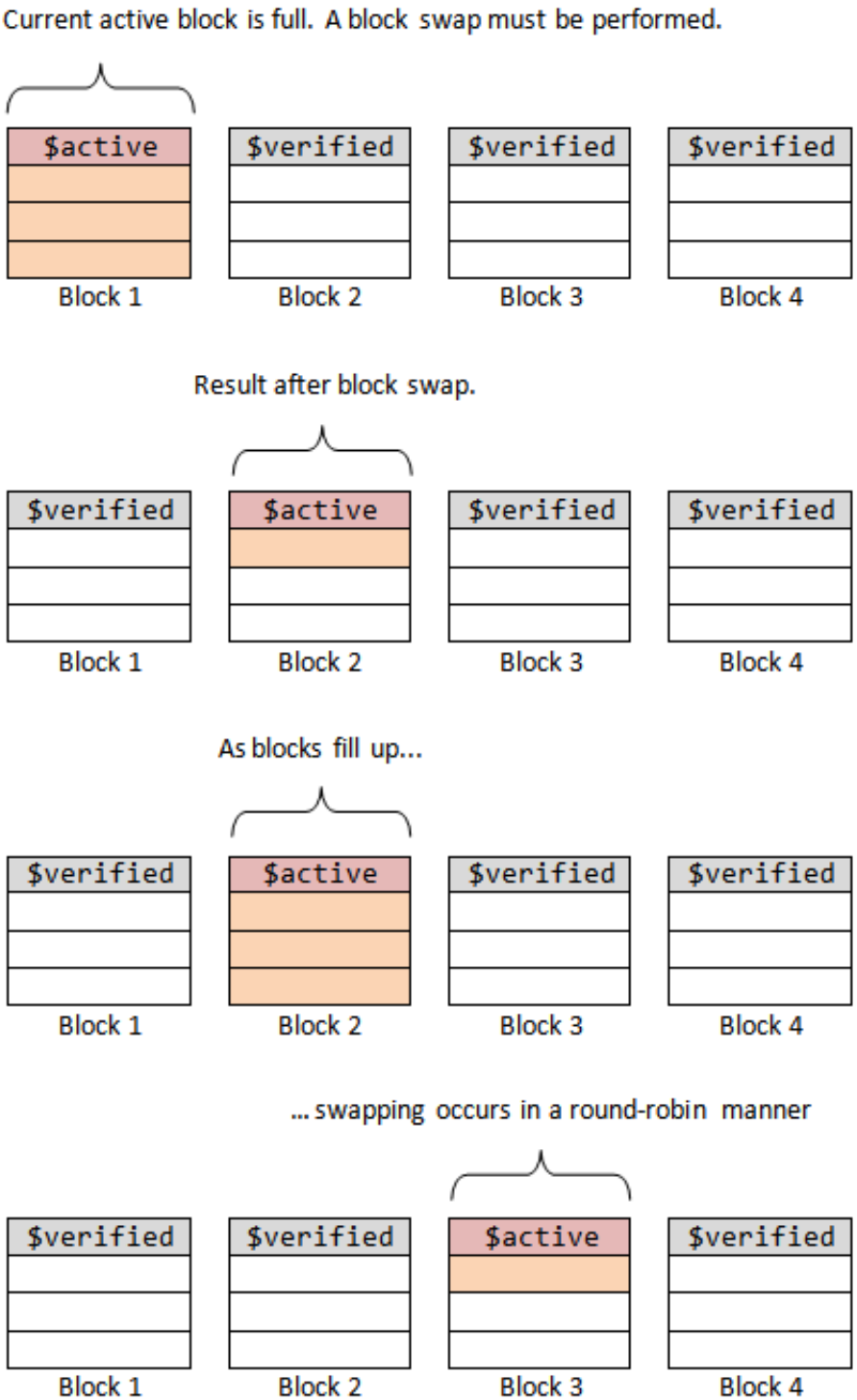

Figure 7. Cycling through multiple blocks

6.1 Reliability

Qorivva devices come with multiple EEPROM Flash blocks. Efficient EEPROM emulation engines will make use of more than 2 blocks when data rates or write-erase counts are high. This is recommended because, as EEPROM write-erase cycling increases, excessive cycling can place strain on the flash blocks and program/erase times may go beyond specification.

For the purpose of tackling any problems that occur, it is recommended to use an EEPROM algorithm that achieves redundancy by permitting the EEPROM engine to continue operation with n-1 blocks. If a user takes care to consider this redundancy aspect then no data loss should ever occur.

How to Reach Us:

Home Page:
nxp.com

Web Support:
nxp.com/support

Document Number AN4868
Revision 2, 03/2017