Keywords: MAXQ, EEPROM, Commit, Rollback, Protected memory, non-volatile memory Feb 28, 2006

APPLICATION NOTE 3771

Protected EEPROM Operations in MAXQ Environments

Nonvolatile memory is essential for embedded microcontroller applications. This application note explains how to use a transaction-based commit-rollback mechanism to protect the contents of an external EEPROM memory device. While targeted for an external memory device, the principles presented here are equally applicable to the internal EEPROM in many MAXQ microcontrollers.

Introduction

For embedded microcontroller applications, nonvolatile memory is usually essential. Whether maintaining settings that must be retained through loss of power or storing a company's critical transaction records, reliable nonvolatile memory is an essential element in the modern microcontroller landscape.

Very often, non-volatile storage takes the form of external serial memory. Literally billions of these memory components have proven reliable for many years in the field. Now available in densities from a few hundred bytes to one megabyte and more, at least one of these compact, inexpensive devices can be found in almost every piece of equipment that needs to maintain settings.

A problem with any type of nonvolatile storage—from EEPROM to flash to rotating storage—is data loss due to interrupted write cycles. If power fails at the instant when a write cycle is being executed, data corruption can result without any easy mechanism of recovery once power is restored.

This article presents a transaction-based commit-rollback mechanism to protect the contents of an external serial EEPROM memory device. The principles presented here are equally applicable to the internal EEPROM contained in many MAXQ microcontrollers. The files for this application are available for download (ZIP, 20.5kb).

Characteristics of I^{2}C EEPROM

Serial memory devices come with a variety of interfaces, but the most commonly used interface is I^{2}C. This bus has a number of advantages: it is highly standardized; it requires only two wires from the controller to the memory; and it has very flexible timing requirements that allow it to be software driven. One I^{2}C master can drive a number of I^{2}C slave devices, minimizing the pin count on the master device.

In all EEPROM devices write cycles take significantly longer than read cycles. This is because charge is transferred across an insulating barrier through a tunneling mechanism during a write cycle, and this operation takes time. While increasing the voltage can accelerate this process, too much potential will cause dielectric breakdown of the barrier, thus damaging the device. Typical write cycle times for EEPROM devices are on the order of ten milliseconds; read cycles often occur in a few hundred nanoseconds.

Many I^{2}C EEPROM devices attempt to reduce the significance of the write cycle time by using a *page mode*. This mode allows multiple bytes to be transferred into a buffer, which is then written at one time to the array. A typical page size for I^{2}C memory devices is 32 bytes. Thus, one can fill 32 bytes of the EEPROM array with only a single write cycle.

This is important, since serial EEPROM devices have a specified *endurance*: an upper limit on the number of lifetime write cycles that a page can tolerate. Typical endurance for write cycles ranges from 10,000 to as much as 1,000,000. Even with a million write-cycle endurance, however, it is easy to see how software could quickly wear out a memory device. Perform only 100 write cycles per second, and in less than three hours the device write cycle count is

exhausted.

Considering these basic EEPROM characteristics, the designer of a robust nonvolatile storage system for an embedded processor must keep the following points in mind:

- No one page should be the target of repeated writes. Specifically, it is not acceptable to make one page a 'directory' that must be updated every time another page is written.
- If power is interrupted during a write cycle, a mechanism must be provided to: (1) detect the interrupted write: and (2) complete the transaction; or (3) roll back the transaction to the prewrite condition.
- Data integrity must be assured through some type of check-data mechanism (checksum, CRC, message digest).

Design Goals

Although the EEPROM considerations mentioned above are resolved through a number of nonvolatile file systems, such file mechanisms place a heavy burden on a small embedded microcontroller. Many file systems will require more RAM than is available in small microcontrollers, and a complete file system is more than required by most applications.

With that in mind, here are the goals for an EEPROM data protection mechanism:

- **Lightweight**: A protection mechanism should reserve no more than 10% of the EEPROM space for check data. It should require only a small amount of computational overhead.
- **Block size**: The block size for the protected blocks should be the same as the native write pages in the EEPROM. Because the page size for EEPROM devices is always an even power of two, software coding is easier than if one or two bytes were reserved from each block.
- **Endurance**: No single page should be written for each protected cycle.
- **Robust**: Every instance of power failure should be *provably* recoverable.

The protection mechanism presented here has six interface functions: **`read, write, commit, rollback, check, and cleanup.`**

The **`read`** function accepts a block number and a pointer to a 32-byte buffer. If the buffer address and the block number are within valid ranges, the routine reads the designated block into the buffer and checks its validity. It will return the conditions *valid read, invalid read, invalid buffer address or invalid page number or protection failure*.

The **`write`** function accepts a block number and a pointer to a 32-byte buffer previously filled with data to be written. If the buffer address and the block number are within valid ranges, the routine copies the data to a nonvolatile holding buffer and marks the buffer as ready to commit.

The **`commit`** and **`rollback`** functions are complementary operations that can be executed following a **`write`**. The **`commit`** function copies the most recently written buffer to its final position in the memory array, and prepares the buffer structure for the next data set to be written. The **`rollback`** function is essentially an 'undo'. It reverses the effect of the most recent **`write`** operation and prepares the buffer subsystem for the next **`write`**.

The **`check`** function reads every block of the memory device and verifies the validity of the stored data. It also inspects the buffer subsystem to ensure that there are no pending writes. Any invalid block or any pending write causes **`check`** to return an error condition.

The **`cleanup`** function fixes a broken EEPROM. In particular, it attempts to determine what failure occurred and what can be done to resolve the issue.

More details about all these functions are given in **Operational Details** below.

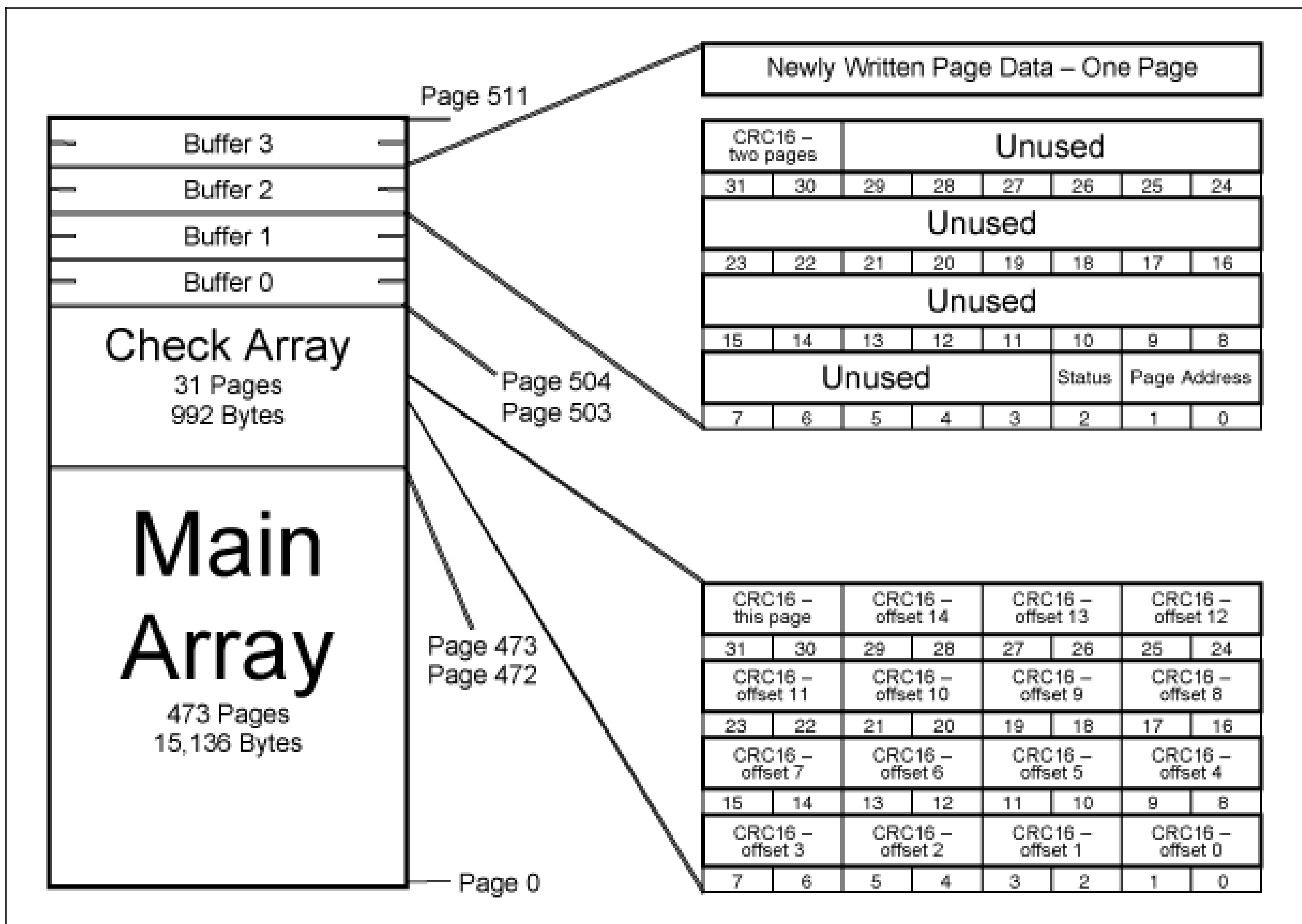

Figure 1. The structure of the EEPROM array. The array is segmented into three areas: the main array that contains actual user data; the check array that contains a CRC for each row in the main array; and a buffer array that contains four buffers to store temporary write data.

EEPROM Structure

Refer to **Figure 1** above for the structure of the EEPROM. The EEPROM contains three major areas:

- **Main Array**: The largest part of the EEPROM is dedicated to data storage. In a 16kB device, there are a total of 512 pages of 32 bytes per page. In such a device, the first 473 pages are dedicated to actual data storage.
- **Check Array**: The second section of the EEPROM checks words for each row in the main array. Each page of the check array contains a set of 15, 16-bit CRC values. The final CRC value in each page checks that page. The check array occupies eleven pages (pages 473 to 503.)
- **Buffer Array**: The final section of the EEPROM contains eight pages that constitute four write buffers. Each buffer consists of four fields: a *data* field that contains the 32-byte data to be written to the main array at the next `commit` directive; an *address* field that identifies the address of the page to which the buffer refers; a *state* field that identifies the state of the buffer (that is, available, occupied, expired); and a 16-bit CRC field that checks the entire write buffer. See Figure 1 above for the buffer structure.

This EEPROM structure fulfils most of the primary goals of the design. First, since each page in the main array is checked in a secondary location, all bits of the page are available for user data. Second, since every page in the main array is checked by a unique word in the check array, there is no single point of failure in the check array and no single page in the entire array that must be updated on every write cycle. Finally, the use of four write buffers distributes the wear for write cycles.

Operational Details

In an unprotected EEPROM, the operational details are simple. A read cycle simply transfers bytes from the selected address to the host; a write cycle transfers bytes from the host to the EEPROM and waits for the operation to complete (a few milliseconds in most devices.) In a protected EEPROM environment, however, reads and writes are more complex operations. In the following section, each operation is dissected to discover exactly what happens when the function is invoked.

Read

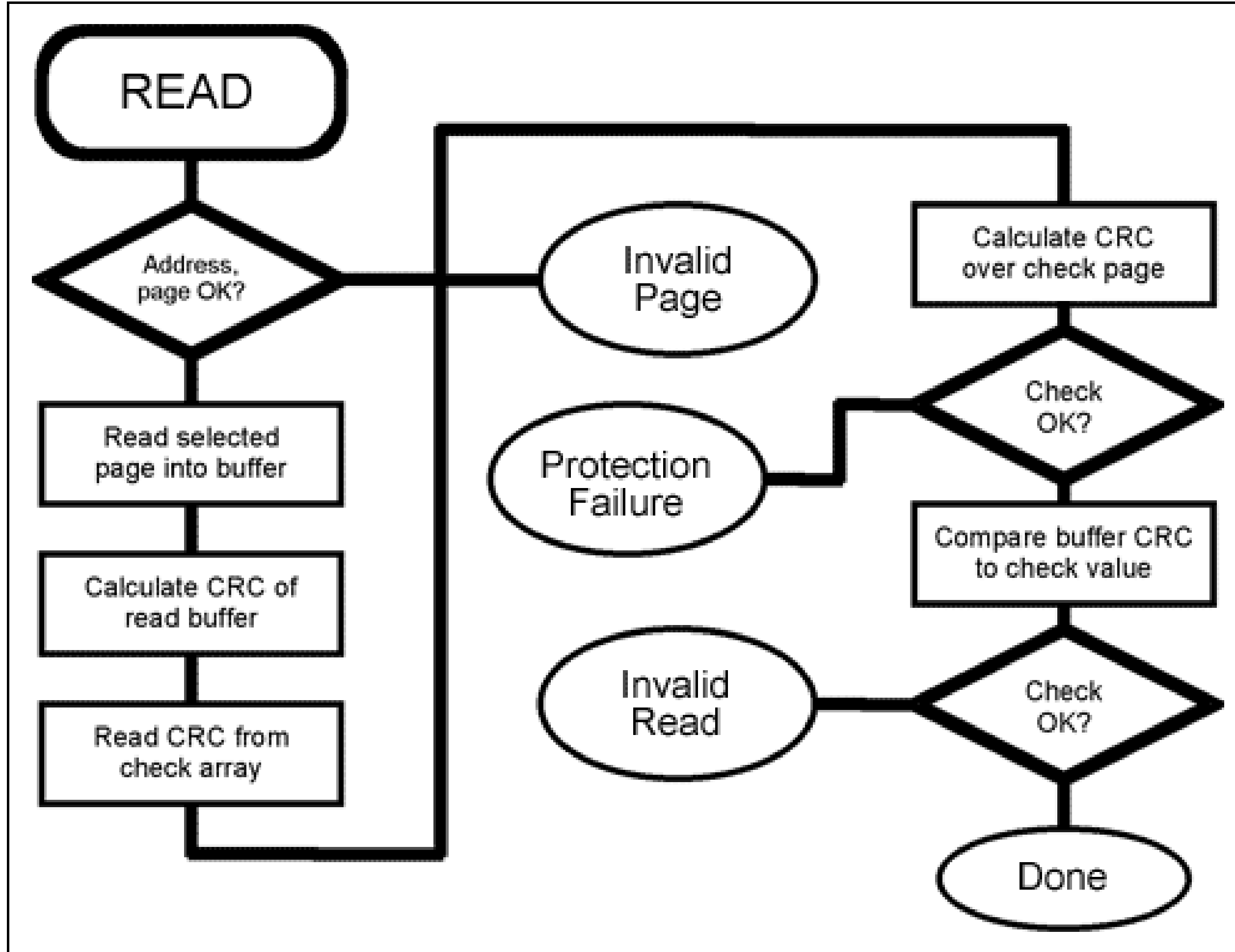

Figure 2. Flowchart for a READ operation.

A read operation, the simplest of the interface functions, is nevertheless rather complex. **Figure 2** illustrates the flow of operations:

- The page address and buffer address are inspected to verify that they are valid. If not, the operation ends here, with the function returning an *invalid buffer address* or *invalid page number* error.
- The selected page is read into the buffer.
- The address of the check page is calculated and the check page is read into a scratch buffer.
- The CRC of the check page is calculated. If it is invalid, a *protection failure* error is returned.
- The CRC is calculated over the data buffer and compared to the CRC stored in the scratch buffer corresponding to the read page. If the CRC matches, the routine returns *valid read*; if the CRC does not match, the routine returns *invalid read*. In any event, the data that was actually read remains in the return buffer for the calling routine to use as appropriate.

Write

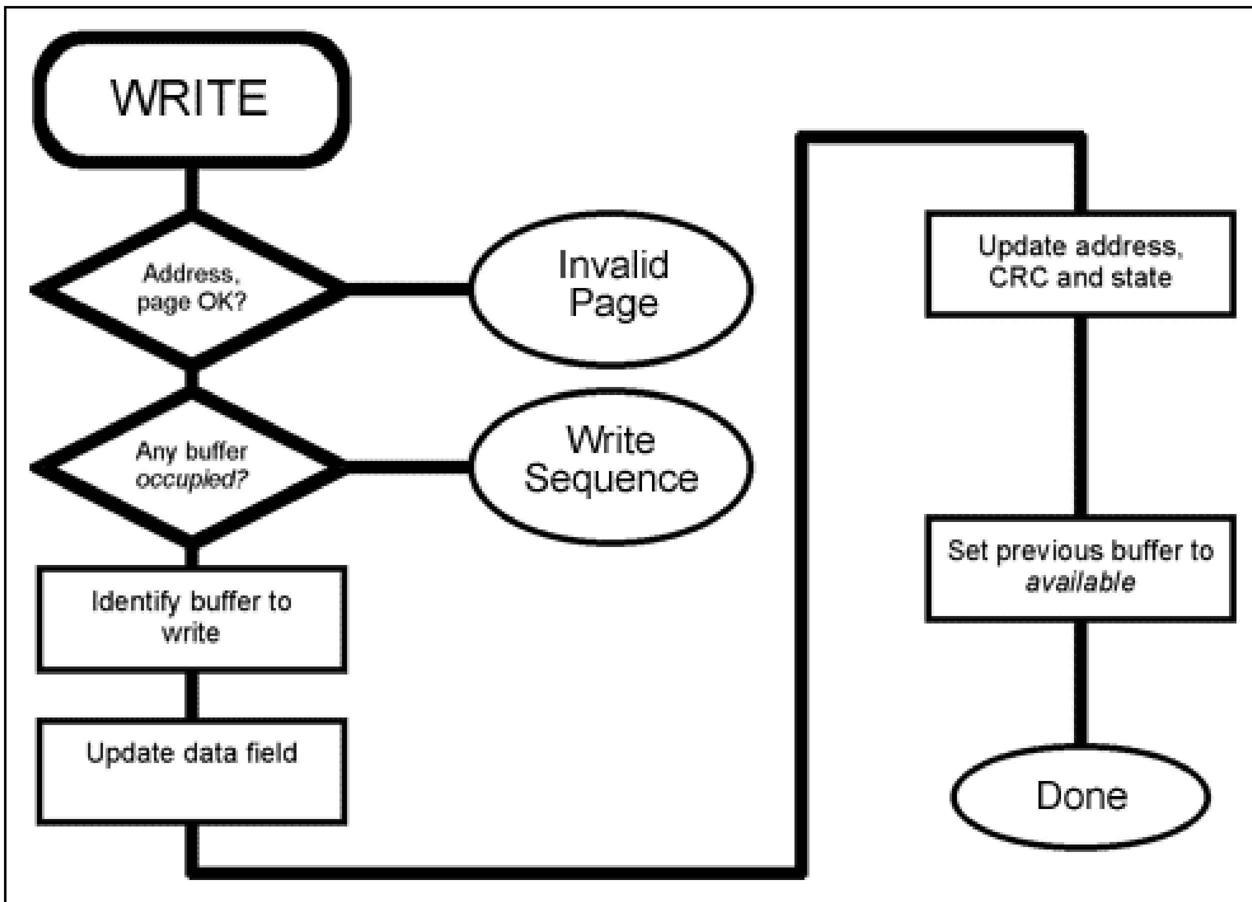

Figure 3. Flowchart of the WRITE operation.

As has been described above, the **`write`** operation does not actually **`write`** to the main array. Instead, the write operation stores its data in one of the four buffers. In this way, the previous data in the main array is preserved until the validity of the write process is ensured. The flowchart in **Figure 3** shows:

- The page address and buffer address are inspected to verify that they are valid. If not, the operation ends here, with the function returning an *invalid buffer address* or *invalid page number* error.
- The *state* field of each write buffer is read. If any buffer has the state *occupied*, the operation fails with a *write sequence* error.
- One of the four write buffers should be in the *expired* state. If so, the *next* buffer in the sequence is activated.
- Data is copied to the *data* field of the write buffer.
- The page address is written to the *address* field. A CRC is calculated and written to the *CRC* field. The state is updated to *occupied*. The *previous* buffer is set to the *available* state (that is, updated from *expired*.)

Note that, at this point, a **`read`** operation will return *the old value* for a newly written page. The new value will not be returned until the **`commit`** operation is complete.

Commit

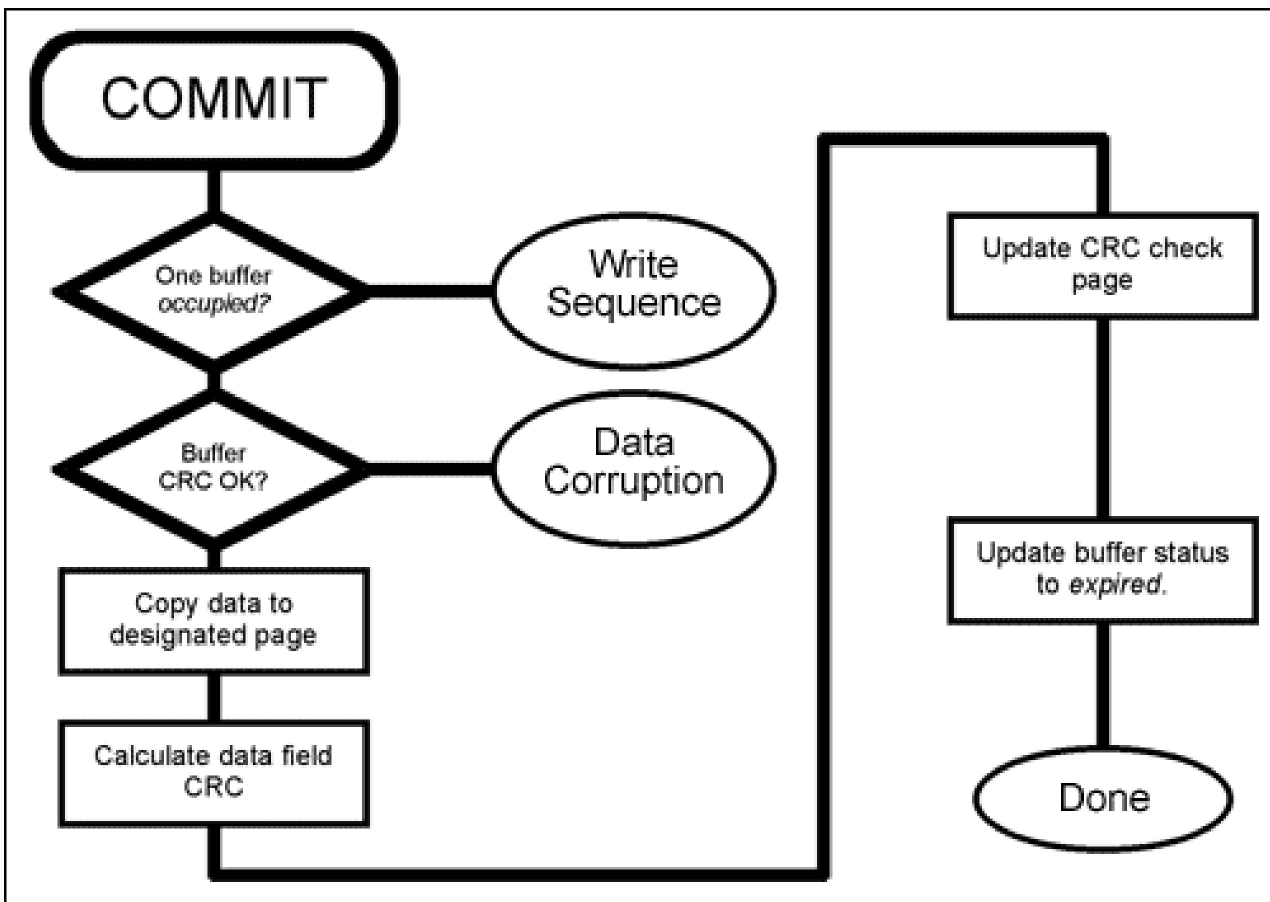

Figure 4. Flowchart of the COMMIT operation.

The **`commit`** function needs no parameters. Its only job is to faithfully transfer the data from the write buffer to the main array, and then to mark the write buffer expired. The **`commit`** function operates as illustrated in **Figure 4**:

- The state field of each write buffer is read. Exactly one buffer should be marked occupied. If this is not so, the function ends here with a write sequence error.
- The occupied buffer is checked with a CRC. If there is no match, a data corruption error is thrown.
- The address is extracted, and the data is written to the designated page in the main array.
- A CRC is calculated across the data portion of the buffer. That value is held in a temporary register.
- The check page is located and read for the selected main page.
- The check page is updated with the previously calculated CRC, and a new CRC is calculated for the check page.
- The check page is written back to the check array.
- The write buffer is updated with an expired status.

Rollback

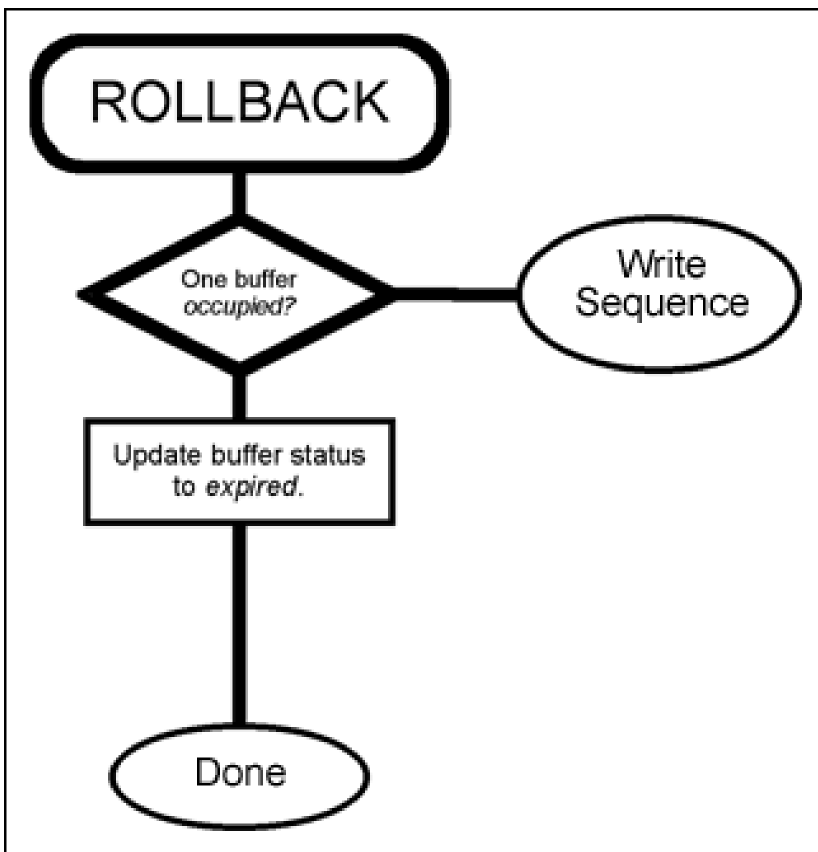

Figure 5. Flowchart of the ROLLBACK operation.

The **`rollback`** function, shown in **Figure 5**, is one of the simplest. Since the main array is not updated following a **`write`** operation but only on completion of a **`commit`** operation, a **`rollback`** needs only to invalidate the write buffer.

- The *state* field of each write buffer is read. Exactly one buffer should be marked *occupied*. If this is not so, the function ends here with a *write sequence* error.
- The *state* field of the selected write buffer is given the value *expired*.

Check

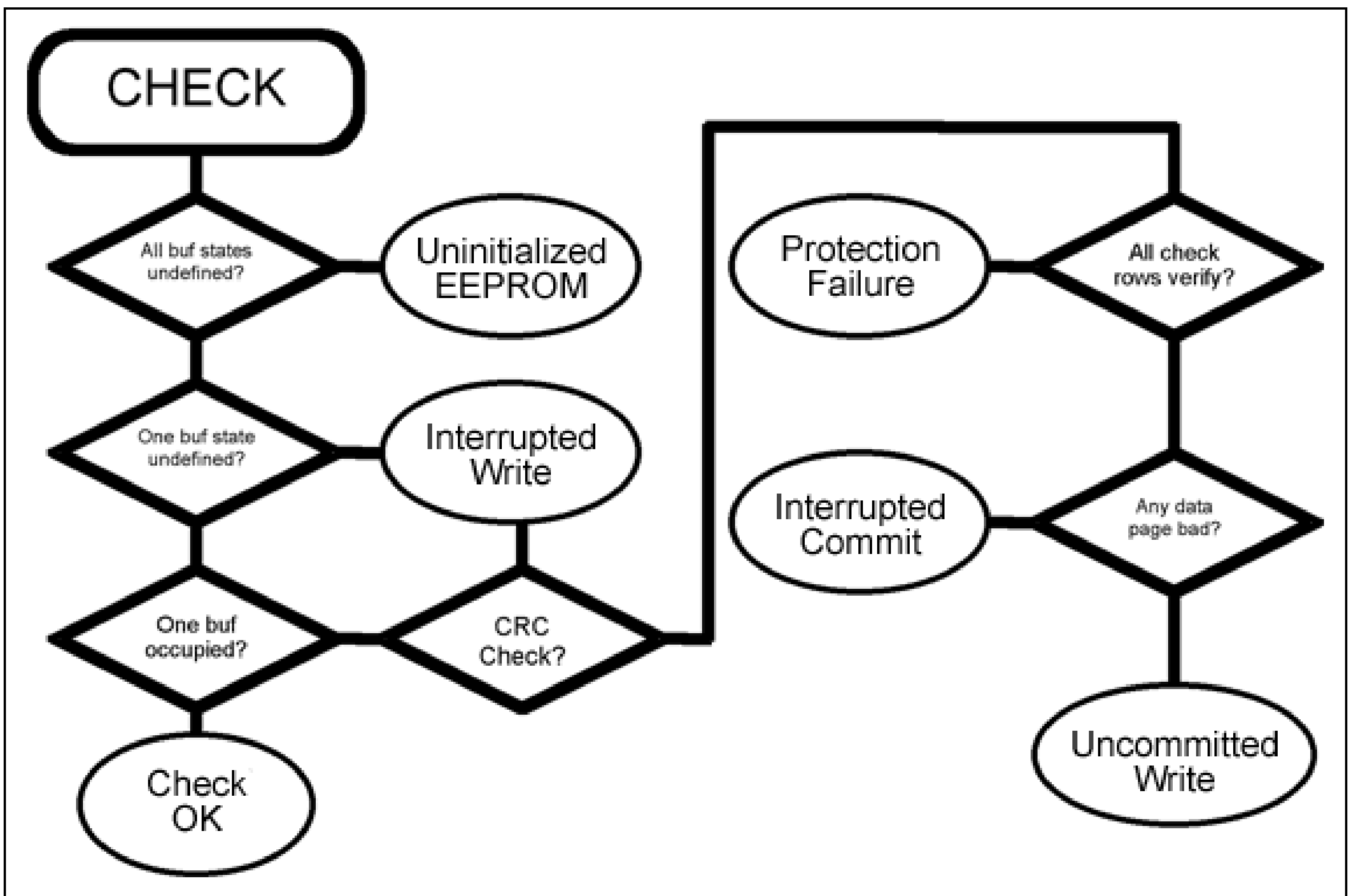

Figure 6. Flowchart for the CHECK operation.

On any power-on event, the `check` function should be called to verify that the EEPROM is ready to accept data. The `check` function verifies the health of the storage system and reports any errors that are encountered. It performs the checks shown in **Figure 6**:

- Read each write buffer. Verify that only one buffer is not in an *available* state. If exactly one buffer contains an undefined state code, return an *interrupted write* error. If all buffers contain undefined state codes, return an *uninitialized EEPROM* error.
- If only one buffer contains an *occupied* state code, CRC that buffer. If the CRC fails, return an *interrupted write* error.
- Check each page of the check array. If any row fails its CRC check, return a *protection failure* error.
- Finally, check each page of the main array against its stored CRC. If any single page fails its CRC, flag an *interrupted commit* error.

Cleanup

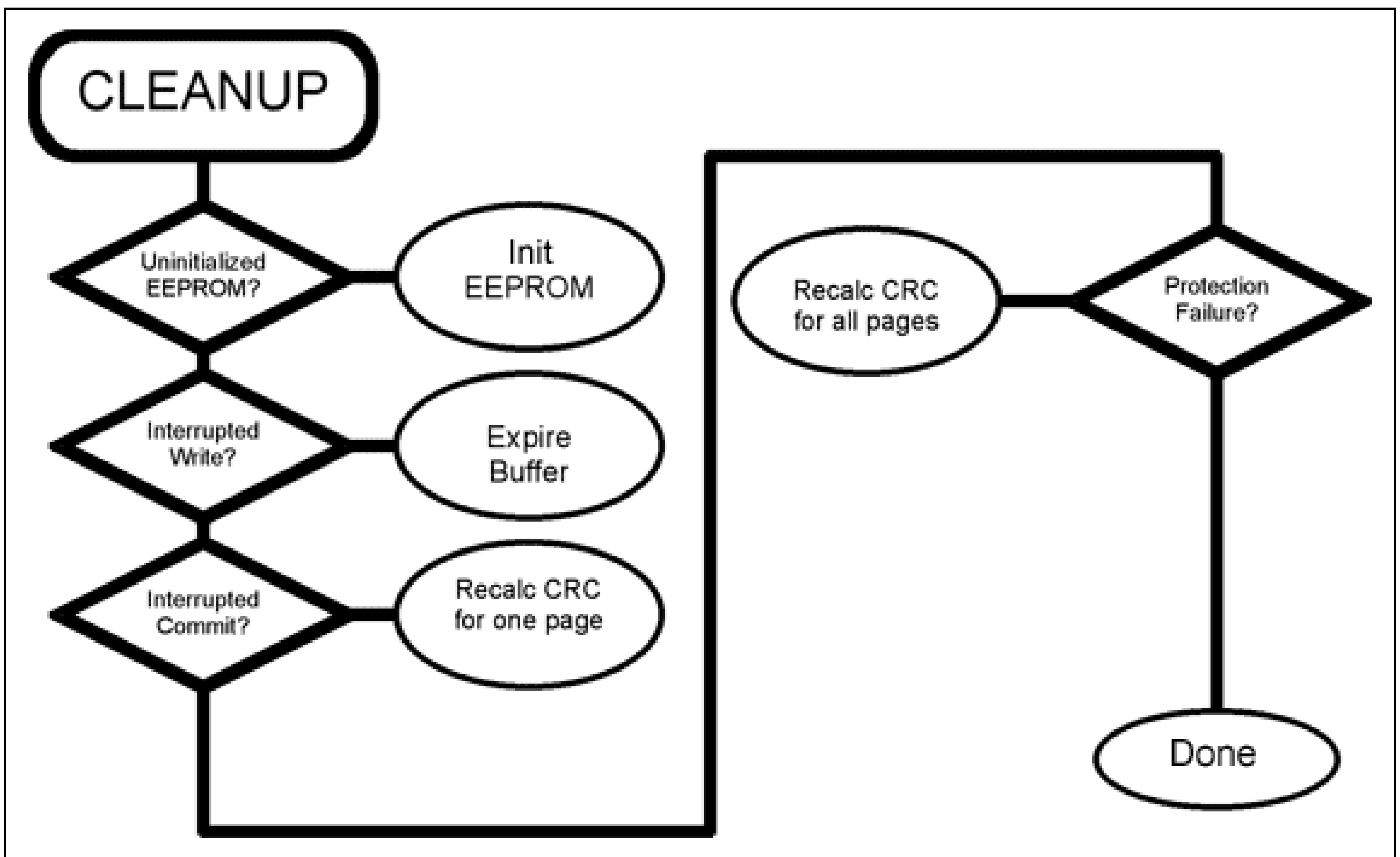

Figure 7. Flowchart for the CLEANUP operation.

The **`cleanup`** function resolves any problems that exist with the EEPROM system. When **`cleanup`** exits, the EEPROM subsystem should be ready to use regardless of the state in which it was found. All uncommitted writes will be rolled back, and failed **`commit`** operations will be completed.

Figure 7 shows how **`cleanup`** works:

- If **`check`** returns an *uninitialized* EEPROM error, the EEPROM is initialized. All data pages are cleared, and all check pages are initialized. All write buffers are cleared and written with *available* status—except the last write buffer, which is initialized with an *expired* state.
- If **`check`** returns an *interrupted* write error, find the one write buffer that has a status other than *available*. Change that status to expired.
- If **`check`** returns an *interrupted commit* error, find the main page for which the CRC does not match. Calculate its CRC and update the associated check page.
- If **`check`** returns *protection failure*, then updating the check page following a **`commit`** operation was interrupted. Read all main pages associated with the defective check page, and refresh the check page.

Proof of Security

The proof of the system's security centers on identifying the vulnerable moments during a write transaction. (A read transaction is inherently safe. No page of the EEPROM is written during a read operation, so data cannot be corrupted.) Having identified those vulnerable moments, one need only identify a recovery process. If all identified vulnerabilities are covered by a recovery mechanism, and if we assume that a check/cleanup cycle will be the first event after any operation that could corrupt an EEPROM write cycle (such as power-up), then the system is provably secure.

Typically in most serial EEPROM devices, a write operation first sets every bit in the affected page to a known value, and then changes the bits that must be changed to write the requested value. Consequently when power fails, it is most likely that *all* bytes of a page corrupt during the interrupted write operation. It is generally possible to recover from this failure event by writing new data to the corrupted page. Nonetheless, the previous data is lost.

Vulnerable moments during a **write** operation are (in chronological order):

- **During write operation to the *data* field**: If a power failure occurs at this time, the check operation will not detect an error. The write buffer being written still has the *available* status, and available buffers do not contain valid CRC values.
- **Writing status for the current write buffer**: This operation changes the *status* field to *occupied*, sets the CRC, and fills in the page address for the write operation. If this process is interrupted, one of the following will be true: (1) the *status* will be invalid, resulting in an *interrupted write* error; (2) the *status* will be valid, but the CRC will fail, again resulting in an *interrupted write* error; or (3) the *status* and the CRC fields will be valid. In this final case, the system has an uncommitted write pending. This condition can be detected because one buffer will be *occupied* and another buffer will be *expired*. If the rest of the subsystem checks out, user code could proceed by issuing a **commit** or **rollback** operation. In any event, the main array and the check array are safe.
- **Clearing previous buffer status to *available***: A buffer will have a corrupted status or CRC, and the *next* buffer will be *occupied*. This means that the operation to clear status on that buffer was interrupted, and either **commit** or **rollback** is possible.
- **Between write and commit operations**: Only a single write buffer will have *occupied* status and its CRC will verify. User code can request a **commit** or a **rollback**. The write buffer array, the check array, and the main array are safe.

The vulnerable moments for the **commit** operation are:

- **Copy *data* field to main array**: If the write operation is interrupted, one page of the main array can be corrupted. The **check** function will identify the condition of (1) a valid, occupied write buffer, and (2) a corrupted main array page as an *interrupted commit*. The write buffer array and the check array will be safe. In this case, **cleanup** will complete the commit and return a cleaned system. Even if the write completed, note that the **check** operation will fail, because the CRC in the check array will differ from the calculated CRC.
- **Update CRC in check array**: If the write operation is interrupted to a page in the check array, the entire page will probably be corrupt. That means that 15 pages in the main array will have invalid CRC values. But **check** can discover this because each page in the check array has its own checksum, and this will fail following an interrupted write. In this case, **check** will return *protection failure*. The fix is, first, to recalculate CRC values for all 15 affected pages. Then write those values to the page in the check array, along with a valid CRC value for the page itself.
- **Update *state* in write buffer array**: If a write cycle is interrupted when the *state* variable is changed from *occupied* to *expired*, the entire row will probably be corrupt. The check array and the main array, however, will remain safe. *Check* will find one page corrupt and will throw an *interrupted write* error. When *cleanup* runs, it will reset the write buffer subsystem, completing the **commit** operation.

Finally, the vulnerable moments during a **rollback** are:

- **Update *state* in write buffer array**: Similar to the final state of a *commit* cycle, this simply resets the *occupied* state in a write buffer to the *expired* state. If this is interrupted, the **check** routine will return *interrupted write* and **cleanup** will reinitialize the entire write buffer array. Once again, the check array and the main array are safe.

Thus, it can be seen that *no matter when power fails or the processor is reset, the storage subsystem maintains its integrity*. Following a power failure, the storage subsystem will be returned to a state in which it is ready to write or read. If a **commit** operation was interrupted, the subsystem will be returned to a state in which a **commit** or **rollback** will succeed.

Designing from Here

The EEPROM storage system for MAXQ microcontrollers is complete. Enhancement of the system is at the discretion of the individual system integrator. Nonetheless, a few ideas come to mind:

- **C Wrappers**: In most C dialects there is a standard way to transfer data to and from assembly-language

subroutines. For example, in the IAR environment, parameters are passed in and out in low-numbered accumulators. Creating a C wrapper for these routines would be as simple as writing function prototypes, since the parameters are already passed in A[0] and A[1]. In other C environments, where parameters are passed on a data stack, a simple wrapper subroutine is necessary.

- **Concurrent Transactions**: Guaranteeing the integrity of a write cycle is fundamental, and providing a mechanism by which this integrity can be achieved is crucial to the overall success of the platform. But many applications require a mechanism by which a series of write cycles can be queued and then executed as a unit, with the assurance that either all will execute or none will execute. The mechanism presented here does *not* work that way. If a system is maintaining records that span multiple pages, it is possible to interrupt a write so that, following recovery, a record contains a page with part of a new entry and a page with part of an old entry. One way of avoiding this problem is to permit multiple **`write`** operations before a **`commit`** is executed. This approach is not quite as simple as it sounds, since a partially committed transaction can contain a mixture of new record fragments, old record fragments, and corrupted pages.
- **Wear Leveling**: A feature of flash file systems, wear leveling refers to the practice of virtualizing the page addresses so that frequently written pages can physically appear anywhere in the array. The best methods for this practice are not obvious. This is because the most obvious solution (a directory of the movable sectors that resides in a fixed location and which is updated once each write) would cause the page at which the directory is stored to quickly wear out. Instead, the directory itself has to be virtualized and distributed, just as the data pages are.
- **Alternative Geometries**: The system presented here assumes a 16kB part with 32-byte pages. If the selected part has larger pages (64 bytes or 128 bytes) the functions will still work, but with some additional write wear. (Updating one 32-byte segment of a 128 byte page will execute a write to the entire 128-byte page.) But these functions will *not* work with devices that incorporate smaller pages. A system could be built that determines the characteristics of the particular EEPROM device on-the-fly, and configure the system parameters accordingly.
- **Enhanced Security**: This system protects against one type of fault: interrupted EEPROM operations due to power failure or unexpected system reset. But EEPROM devices can, and occasionally do, fail in other ways. An example of this is soft failure due to circuit noise or ionizing radiation. Another example is hard-failure due to wear-out of one or more cells.

  One way to deal with these issues is to calculate and maintain *syndromes* rather than simple CRC check words. A syndrome is similar to a check word, but contains enough information to correct simple bit errors. The simplest syndrome system can check n data bits with $\log_2 n + 1$ check bits. Thus, with a page size of 32 bytes (256 bits) a syndrome word of only nine bits could correct any single-bit error. More complex systems can be brought to bear on the problem as requirements for data integrity become more stringent.

Conclusion

The external serial EEPROM provides a reliable means of storing nonvolatile data in a microcontroller environment. Using the techniques presented here, the serial EEPROM can be made reliable even in the face of interrupted write cycles. Designers should consider these techniques whenever data integrity is critical to the application.

Application Note 3771: http://www.maxim-ic.com/an3771

More Information
For technical questions and support: http://www.maxim-ic.com/support
For samples: http://www.maxim-ic.com/samples
Other questions and comments: http://www.maxim-ic.com/contact

AN3771, AN 3771, APP3771, Appnote3771, Appnote 3771
Copyright © 2005 by Maxim Integrated Products
Additional legal notices: http://www.maxim-ic.com/legal