Basic Platform Independent Non-Volatile Storage Layers

TEP:	129
Group:	Storage Working Group
Type:	Documentary
Status:	DRAFT
TinyOS-Version:	2.x
Author:	David Moss
Author:	Junzhao Du
Author:	Prabal Dutta
Author:	Deepak Ganesan
Author:	Kevin Klues
Author:	Manju
Author:	Ajay Martin
Author:	and Gaurav Mathur

Abstract

The storage abstractions proposed by TEP 103 are implemented on a platform-dependent basis. A version of BlockStorage, ConfigStorage, and LogStorage were created from the ground up for both the AT45DB and ST M25P80 flash chips. Looking forward into the further growth and development of hardware, rebuilding each of these storage layers for every new flash chip will be time consuming and cause compatibility issues.

We propose versions of BlockStorage, ConfigStorage, and LogStorage be built on top of a platform-independent interface. This would allow one version of each to exist on multiple platforms. Platform-independent implementation concepts are discussed along with recommended solutions, and changes are proposed to the interfaces defined by TEP 103.

1. Introduction

The implementations of the BlockStorage, ConfigStorage, and LogStorage layers described in TEP 103[1] are platform-dependent. Platform- dependent implementations can cause behavioral and usage differences as well as compiling problems when attempting to port an application written on one platform to another.

Building upon the DirectStorage, DirectModify, and VolumeSettings abstraction layers defined in TEP128[2], the three basic storage solutions can be implemented in a platform-independent manner. This requires combining all properties of various memory types, which aids in the creation of

platform-independent storage solutions. Behavioral differences are minimized, and applications using the platform-independent storage layers can expect to work the same way on different types of non-volatile memory.

2. Implementing a platform-independent BlockStorage

The DirectStorage interface initially stemmed from the BlockStorage interface with differences in interfaces, organization, and erase behavior, as well as the additional VolumeSettings interface. To implement BlockStorage on DirectStorage, the erase behavior must be extended to erase the entire volume instead of an individual erase unit.

VolumeSettings can first be accessed to determine the total number of erase units in the currently mounted volume. Looping through these erase units, DirectStorage is accessed to erase() each one. At the end of the erase operation, the entire volume is set back to fill bytes (0xFF).

2.1 Improved BlockStorage interface

Previous BlockStorage interfaces were divided into BlockRead and BlockWrite. This was found to be cumbersome because applications typically required access to both interfaces. The getSize() is unnecessary due to the addition of the VolumeSettings interface. All other BlockStorage commands can simply pass through to their respective DirectStorage functions. This TEP proposes the following unified BlockStorage interface::

```
interface BlockStorage {

  command error_t read(uint32_t addr, void *buf, uint32_t len);

  command error_t write(uint32_t addr, void *buf, uint32_t len);

  command error_t erase();

  command error_t flush();

  command error_t crc(uint32_t addr, uint32_t len, uint16_t baseCrc);



  event void readDone(uint32_t addr, void *buf, uint32_t len, error_t er-
ror);

  event void writeDone(uint32_t addr, void *buf, uint32_t len, error_t er-
ror);

  event void eraseDone(error_t error);

  event void flushDone(error_t error);

  event void crcDone(uint16_t calculated-
Crc, uint32_t addr, uint32_t len, error_t error);

}
```

`read(uint32_t addr, void *buf, uint32_t len);` • Read 'len' bytes into `*buf` from the given address

- Returns FAIL if the request cannot be handled
- Signals readDone(...) when complete.

`write(uint32_t addr, void *buf, uint32_t len);` • Write 'len' bytes from `*buf` starting at the given address
- Returns FAIL if the request cannot be handled
- Signals writeDone(...) when complete.

erase(); • Erase the entire volume
- Returns FAIL if the request cannot be handled
- Signals eraseDone(...) when complete.

flush() • All data that has been previously written and is not yet located on non-volatile memory should be immediately stored to non-volatile memory.
- Returns FAIL if the operation cannot be completed at this time
- Signals flushDone(...) when complete.

crc(uint32_t addr, uint32_t len, uint16_t baseCrc); • Calculate the CRC of 'len' bytes starting at the given address, using the given baseCrc as a seed.
- Returns FAIL if the request cannot be handled
- Signals crcDone(...) when complete.

3. Implementing a platform-independent LogStorage

As described in TEP 103, logging can be implemented using two different methods: linear and circular. A linear log fills up its volume and stops when it comes to the end. A circular log allows at least half of its volume to remain valid while continuing to write the other half. As previously described, this requires at least two erase units to be effective.

Both logging behaviors can be implemented using the same code. A flag for linear log behavior prevents the logger from freeing up an erase unit in which to continue writing.

It should also be noted that the use of a circular log mandates the use of at least two erase units on the volume. As discussed in TEP128[2], forcing volumes to contain at least two erase units solves this issue.

3.1 LogStorage Boot Behavior

In the previous LogStorage implementations, reboots cause data to be lost or overwritten because the beginning and ends of the log were never located. Preventing previously stored data from being lost or overwritten after reboot is critical for the successful use and integration of logging storage components within a practical, deployable system.

A method is required on boot to locate the first memory location to read from as well as the next available memory location to write to. Although one method is to use microcontroller user memory to store the information, the goal is to avoid relying on external support due to cross-platform compatibility reasons. Luckily, storing and updating this information on the volume itself is easier than it seems.

Flash cannot overwrite areas of memory it has already written without performing a read-modify-write operation, and this operation is not supported on many flash types. Regardless of whether the memory type can support modifications, all types of memory - including EEPROM - should take wear-leveling into account. Combining these properties, it is possible to design a method of maintaining and updating logging start and stop information in a cross-platform compatible manner.

The method of locating logging properties on boot is simplified by making entries aligned to erase unit boundaries, never allowing a single entry to bridge erase units. This also prevents invalid entries from being created as a result of erasing an erase unit.

To find the first available write address to add new log entries, the first header entry on each erase unit is evaluated to find the greatest 32-bit "cookie" value that is not fill-bytes (0xFFFFFFFF). The erase unit with the largest value contains the newest data. Next, each entry in that erase unit can be iterated through by reading each header and skipping the length of the header + data, until a header with the value 0xFFFFFFFF is located. The address of this location is the first available address to write.

Finding the first available address for reading involves the same process. The first header entry on each erase unit is evaluated to find the lowest 32-bit "cookie" value. The entry with the lowest value is the beginning of the log.

The first entry to read from and last address to write to MUST be located on platform boot.

3.2 Appending log entries

The previous M25P80 log storage implementation is a good place to start. In it, each write consists of a 32-bit header "cookie" and the data to be appended to the log. Locating the beginning of the log is therefore a matter of finding the lowest header cookie value. If this were to be implemented so entries align with erase unit boundaries, only the first header of each erase unit needs to be checked for the lowest value.

32-bits leaves plenty of space to increment log entry values for. If the log were to append one chunk of data every second, it would take 136.1 years before the 32-bit header recycles to 0 and causes an issue in properly locating the first and last log entries. This is well beyond the expected lifetime of a deployed system.

Each header entry can provide additional support for every data entry by allowing it to track the amount of appended data as well as an optional 8-bit CRC to verify the data is valid::

```
typedef struct log_header_t {
  uint32_t cookie;
  uint8_t length;
  uint8_t crc;
} log_header_t;
```

When the logger appends to the next erase unit boundary, it can first erase it to ensure future appends are not corrupted by existing bytes. At the point where it reaches the end of its volume, the 'circular' logging flag can be used to determine if the logger should go back to the beginning of the volume and continue writing. Again, this is performed in conjunction with the VolumeStorage interface to determine erase unit properties.

3.3 Reading log entries

After the first log entry is located, entries are extracted by first reading the header of a single entry, and using the information from the header to pull out the subsequent log information. After each read, the read pointer is updated to point to the read location of the next header.

If the header ID is fill bytes (0xFFFFFFFF), then the entry is invalid and the read process has reached the end of the log. As with the ST M25P80 implementation, entries may be randomly seeked by providing the 32-bit "cookie" identifier to locate.

3.5 Logging conclusions

This proposed logging storage solution will provide the ability to maintain and locate previously logged data as well as support truly circular logs by mandating more than one erase unit per volume. Behavioral differences between flash chip implementations are eliminated so one application can access logging

storage across all platforms. Furthermore, reboots will not cause logged information to be lost or overwritten.

Existing LogRead and LogWrite interfaces defined in TEP 103[2] are sufficient to implement cross-platform logging abilities.

4. Implementing a platform-independent ConfigStorage

The previous interface to ConfigStorage looks very similar to that of BlockStorage. The ConfigStorage interface allows reads and writes to arbitrary addresses, which is not optimal. One critical concept behind the storage of configuration data is the ability to modify and overwrite existing parameters while preventing surrounding data from being corrupted. The former ConfigStorage definition did not support this, so a new solution should be explored and developed.

This new solution should prevent an application from specifying addresses to read and write to on the volume. Instead, it should dictate addresses underneath, not allowing applications to see those addresses. In essence, the solution should handle the allocation of memory and take the burden of determining valid addresses off of the application layer. This will allow it to support multiple components written by different authors on the same system.

4.1 Hash-based configuration storage

Several practical deployments have demonstrated the effectiveness of a hash-based configuration storage solution. In it, any module in a system can store and update any type of data of any size without destroying other modules' information.

The implementation is similar to that of ConfigStorage in that each entry contains a header followed by a variable amount of data. The header is different than ConfigStorage headers in that instead of containing a 32-bit cookie, it contains a 32-bit hash key and a magic number to keep track of state::

```
typedef struct config_header_t {
  uint32_t hashId;
  uint8_t magic;
  uint8_t length;
  uint8_t crc;
} config_header_t;
```

The magic number allows Configuration storage to determine if the entry is valid or has been deleted. Because non-volatile memory typically writes from 1's to 0's, the magic number can take on a finite number of values before it is filled with 0's. For example::

```
enum {
  ENTRY_EMPTY = 0xFF,
  ENTRY_VALID = 0xEE,
  ENTRY_INVALID = 0xDD,
};
```

Configuration data can be stored and retrieved by indexing it with a hash key. Like AM types in the radio stack[3], each key is uniquely defined by the developer.

Each new update to the configuration storage should create an entirely new entry while invalidating any previous entry containing the same hash key. Optionally, a small cache in RAM can be used to maintain information about where existing hash ID's are located in memory, so non-volatile memory does not need to be traversed each time.

When space runs out on one erase unit, the next erase unit can be used to copy in all valid data from the first. The first erase unit can then be erased. This allows parameters and configuration data to be infinitely updated so long as the amount of valid data plus supporting headers is less than half the volume's total erase unit size.

4.2 Improved ConfigStorage interface

The interface to access the configuration storage mechanism is proposed as follows, allowing the application layer to continually update previously stored parameters while preventing it from accessing memory addresses directly::

```
interface ConfigStorage {

  command error_t getTotalKeys();

  command error_t insert(uint32_t key, void *value, uint16_t valueSize);

  command error_t retrieve(uint32_t key, void *valueHolder, uint16_t max-
ValueSize);

  command error_t remove(uint32_t key);

  command error_t getFirstKey();

  command uint32_t getLastKey();

  command error_t getNextKey(uint32_t presentKey);


  event void inserted(uint32_t key, void *value, uint16_t valueSize, er-
ror_t error);

  event void retrieved(uint32_t key, void *valueHolder, uint16_t value-
Size, error_t error);

  event void removed(uint32_t key, error_t error);

  event void nextKey(uint32_t nextKey, error_t error);

  event void totalKeys(uint16_t totalKeys);

}
```

getTotalKeys() • Determine the total number of valid keys stored on non-volatile memory
- Signals totalKeys(...) when complete

`insert(uint32_t key, void *value, uint16_t valueSize)` • Insert some data into the configuration storage associated with the given key
- Signals inserted(...) when complete

`retrieve(uint32_t key, void *valueHolder, uint16_t maxValueSize)` • Retrieve the value associated with the given key. The maximum value size is the maximum amount of data that can be loaded into the `*valueHolder` location, to avoid overflow
- Signals retrieved(...) when complete

remove(uint32_t key) • Removes the given key and its associated values from non-volatile memory
- Signals removed(...) when complete

getFirstKey() • Determines the first key available for reading on non-volatile memory
- Signals nextKey(...) when complete

getNextKey(uint32_t presentKey) • Obtain the next available key on non-volatile memory based on the current key. Allows the application to traverse through all stored keys.
- Signals nextKey(...) when complete

getLastKey() • Returns last key available for reading on non-volatile memory.
- This value is assumed to be located in cache, so it can return immediately

5. Author's Address

David Moss
Rincon Research Corporation
101 N. Wilmot, Suite 101
Tucson, AZ 85750

phone - +1 520 519 3138
phone - +1 520 519 3146
email ? dmm@rincon.com

Junzhao Du
Contact -

Prabal Dutta
Contact -

Deepak Ganesan
Contact -

Kevin Klues
Contact -

Manju
Contact -

Ajay Martin
Contact -

Gaurav Mathur
Contact -

6. Citations

[1] TEP 103: Permanent Data Storage (Flash).

[2] TEP 128: Platform independent Non-Volatile Storage Abstraction Layers

[3] TEP 116: Packet Protocols

[4] Atmel AT45DB041B datasheet. http://www.atmel.com/dyn/resources/prod_documents/DOC1432.PDF

[5] ST M25P80 datasheet. http://www.st.com/stonline/products/literature/ds/8495/m25p80.pdf

[6] K9K1G08R0B datasheet. http://www.samsung.com/Products/Semiconductor/NANDFlash/SLC_SmallBlock/1Gbit/K