# Platform Independent Non-Volatile Storage Abstractions


| | |
|---|---|
| **TEP**: | 128 |
| **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 a layer of abstraction to reside between a chip-dependent flash chip implementation and platform-independent storage implementations. This abstraction layer should provide methods to perform basic flash operations (read, write, erase, flush, crc), as well as provide information about the physical properties of the flash chip. Efficiency concerns are mitigated by the fact that each platform-independent abstraction is implemented in a platform-dependent manner, allowing it to exist on different types of memory such as NAND flash, NOR flash, and EEPROM. This abstraction layer should allow one implementation of each storage solution to operate across many different platforms.


## 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. A true abstraction layer would exhibit the same set of interfaces and no differences in behavior when implemented across various types of non-volatile memory.

A well defined non-volatile memory abstraction layer should allow core functionality to work on a variety of platforms without modification. Some flash chips may provide extra functionality. If a particular applications on a specific platform wants to take advantage of extra functionality provided by a flash chip, it still has the opportunity to access those features directly with the understanding that its implementation is no longer considered platform-independent.

## 1.1 Platform-dependent volume settings

Differences exist between the TEP 103 storage implementations on the AT45DB, ST M25P80, and PXA27X P30 flash chips.

First, volume information is defined using two distinct methods. As was discussed in TEP 103, an XML file is responsible for the allocation of flash memory into volumes at compile time. Individual storage layers can then mount to the defined volumes, which allows those layers to share the flash memory resource amongst each other.

The AT45DB implementation running on mica* platforms converts the information presented in the application's volumes XML file into information accessible through macros::

```
#ifndef STORAGE_VOLUMES_H
#define STORAGE_VOLUMES_H

enum {
  VOLUME_BLOCKTEST,
};

#endif
#if defined(VS)
VS(VOLUME_BLOCKTEST, 1024)
#undef VS
#endif
#if defined(VB)
VB(VOLUME_BLOCKTEST, 0)
#undef VB
#endif
```

The ST M25P80 implementation running on TelosB/Tmote platforms, on the other hand, converts the information in the volumes XML file into an array of constants::

```
#ifndef __STORAGE_VOLUME_H__
#define __STORAGE_VOLUME_H__

#include "Stm25p.h"

#define VOLUME_BLOCKTEST 0

static const stm25p_volume_info_t STM25P_VMAP[ 1 ] = {
    { base : 0, size : 4 },
};

#endif
```

Furthermore, the two implementations defined incompatible interfaces for accessing information about volumes. For example, the AT45DB interface provides the following::

```
interface At45dbVolume {
  command at45page_t remap(at45page_t volumePage);
  command at45page_t volumeSize();
}
```

The ST M25P80 interface defines a different interface, which allows applications to access the volume settings directly through the stm25p_volume_info_t array::

```
interface Stm25pVolume {
  async event volume_id_t getVolumeId();
}
```

Accessing volume information is very platform-dependent. A single method should be integrated to access volume settings and non-volatile memory properties across any platform.

Another issue exists with the previous concept of volumes. Any storage solution that wishes to retain valid data while erasing invalid data MUST have access to at least two erase units. Circular logging, configuration storage, variable storage, dictionary storage, file systems, and more all require a minimum of two erase units to be implemented effectively. One erase unit can be used to erase all the invalid data while other erase unit(s) retains any valid data.

Therefore, the minimum allowable volume size should be twice the size of a single erase unit to effectively support the majority of storage applications. The XML tools that process and allocate volumes should prevent a user from defining a volume too small::

```
+------------+--------+------------+-----------+------------+
|            | AT45DB |   ST M25P  |   PXA27x  | K9K1G08R0B |
+------------+--------+------------+-----------+------------+
| Min.Volume |  512B  | 512B-128kB | 128-256kB | 16kB-64kB  |
+------------+--------+------------+-----------+------------+
```

## 1.2 Platform-dependent component signatures

The storage components' signatures differ across implementations. For example, the PXA27X P30 flash defines "P30BlockC", "P30ConfigC", and "P30LogC" in place of "BlockStorageC", "ConfigStorageC", and "LogStorageC". Furthermore, the BlockStorageC configuration in the AT45DB implementation takes the following form::

```
generic configuration BlockStorageC(volume_id_t volid) {
  provides {
    interface BlockWrite;
    interface BlockRead;
  }
}
```

while the ST M25P80 implementation adds another interface::

```
generic configuration BlockStorageC( volume_id_t volume_id ) {
  provides interface BlockRead;
  provides interface BlockWrite;
  provides interface StorageMap;
}
```

The StorageMap interface on the M25P80 flash chip simply allows an application to convert a volume-based virtual address into a physical address on flash. Although it is a good idea, it is not consistent with other platforms' defined interfaces.

# 2. DirectStorage

## 3.1 Differences and advantages

The core current BlockStorage, ConfigStorage, and LogStorage layers can all be implemented on a platform-independent abstraction layer. Providing an interface that allows direct, unimpeded access to the memory below while offering information about the properties of that memory is the first step in doing so.

The DirectStorage interface was created to as part of the answer to this issue. DirectStorage resembles the BlockStorage interface in many ways, with two significant exceptions:

1. Erase operation BlockStorage's behavior erases the entire volume at a time, which may consist of multiple erase units. DirectStorage allows erases to occur on per-erase unit basis. Therefore, if only a portion of the volume needs to be erased, it can.

2. Organization BlockStorage defines two different interfaces for interacting with the flash: BlockRead and BlockWrite. These two interfaces are combined into one interface. The getSize() command provided by the BlockRead interface is removed and replaced with VolumeSettings, which will be discussed later. Also, sync()/syncDone() is replaced with flush/flushDone(), which is responsible for writing any data that has not already been written to non-volatile memory. Although the crc() command can technically exist above BlockStorage as well as DirectStorage, it remains in DirectStorage for its ease of use.

Finally, layers should have the ability to be added beneath DirectStorage to further optimize and enable memory operation. For example, the ST M25P80 flash does not have any on-board RAM buffers, so it is up to the microcontroller to buffer and flush out data to write units. This functionality may not be desirable on all applications because it uses valuable microcontroller resources; therefore, it should be removable as layers can be added and removed from radio stack architecture.

Other memory types may require extra support in and underneath the hood of DirectStorage as well. NAND flash, for example, requires bad block management and error correction. This functionality can be implemented without changing the behavior of the DirectStorage interface above.

## 3.2 DirectStorage Interface

The DirectStorage interface is described below. Each "addr" variable is a virtual address, with 0x0 relative to the base address of the volume. This base address may actually be physically located somewhere else on the non-volatile memory::

```
interface DirectStorage {

  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(uint16_t eraseUnitIndex);

  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(uint16_t eraseUnitIndex, 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 volume is already in use
- 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 volume is already in use
- Signals writeDone(...) when complete.

`erase(uint16_t eraseUnitIndex);` • Erase a single 0-indexed erase unit
- Returns FAIL if the volume is already in use
- 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 volume is already in use
- Signals crcDone(...) when complete.

### 3.3 DirectModify Interface

Some memory types have the ability to modify their contents without destroying surrounding data.

The AT45DB NOR-flash, for example, is able to do this because it has built in RAM buffers coupled with small erase unit sizes. The physical RAM buffers perform a read-modify-write operation to effectively change the contents of flash, allowing it to emulate the behavior of an EEPROM with the speed and efficiency of NOR-flash.

The ATmega128 microcontroller has 4kB of internal EEPROM memory which can be directly modified. Also, the MSP430 has 256 bytes of internal NOR-flash memory which is divided into two segments of 128 bytes each. When implemented properly, this NOR-flash memory can be modified in a fault-tolerant manner.

The ST M25P80 NOR-flash cannot support modification without sacrificing significant overhead. It has 16 erase units that are 64kB each, which is too large to effectively modify bytes.

While not all memories support modification, a unified interface should exist to interact with memories that do. This interface should be access with the understanding that applications built on top may not be portable to all memory types. Also, DirectStorage and DirectModify are mounted to their own individual volumes, so DirectModify cannot share its allocated memory resources with a DirectStorage interface::

```
interface DirectModify {

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

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

  command error_t erase(uint16_t eraseUnitIndex);

  command error_t flush();

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

  command bool isSupported();


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

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

  event void eraseDone(uint16_t eraseUnitIndex, 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);

}
```

`modify(uint32_t addr, void *buf, uint32_t len)` • Modify 'len' bytes located on non-volatile memory at the given address, replacing them with data from the given buffer

- Returns FAIL if the volume is already in use
- Signals modified(...) when the operation is complete

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

- Same as DirectStorage.read(...)
- Returns FAIL if the volume is already in use
- Signals readDone(...) when complete.

`erase(uint16_t eraseUnitIndex);` • Erase a single 0-indexed erase unit

- Returns FAIL if the volume is already in use
- 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.

- Same behavior as flush() methods found in Java
- 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 volume is already in use
- Signals crcDone(...) when complete.

`isSupported()` • Returns TRUE if DirectModify is available on the current memory type

## 3.4 VolumeSettings Interface

As was shown in Section 1.1, finding information about the current volume required platform-dependent methods of access. VolumeSettings provides a unified method of accessing information about the underlying memory chip and volume settings.

VolumeSettings MUST be implemented separately for DirectStorage and DirectModify, not only because those abstractions will exist on separate volumes, but also because the DirectModify interface may change the available size of the volume to support certain memory types such as NAND- and NOR-flash::

```
interface VolumeSettings {

  command uint32_t getVolumeSize();

  command uint32_t getTotalEraseUnits();

  command uint32_t getEraseUnitSize();

  command uint32_t getTotalWriteUnits();

  command uint32_t getWriteUnitSize();

  command uint8_t getFillByte();

  command uint8_t getEraseUnitSizeLog2();

  command uint8_t getWriteUnitSizeLog2();

}
```

**getVolumeSize()** • Returns the size of the volume the DirectStorage layer is mounted to, in bytes

**getTotalEraseUnits()** • Returns the total number of erase units on the mounted volume

**getEraseUnitSize()** • Returns the size of an individual erase unit, in bytes

**getTotalWriteUnits()** • Returns the total number of write units on the mounted volume

**getWriteUnitSize()** • Returns the size of an individual write unit, in bytes

**getFillByte()** • Returns the default byte value found on the memory after an erase, which is typically 0xFF

**getEraseUnitSizeLog2()** • Returns the size of an erase unit in Log2 format for ease of calculations

**getWriteUnitSizeLog2()** • Returns the size of a write unit in Log2 format for ease of calculations

## 4. 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 -

## 5. Citations

[1] TEP 103: Permanent Data Storage (Flash). http://tinyos.cvs.sourceforge.net/*checkout*/tinyos/tinyos-2.x/doc/html/tep103.html

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

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

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