Permanent Data Storage (Flash)

TEP:	103
Group:	Core Working Group
Type:	Documentary
Status:	Final
TinyOS-Version:	2.x
Author:	David Gay, Jonathan Hui

Abstract

This memo documents a set of hardware-independent interfaces to non-volatile storage for TinyOS 2.x. It describes some design principles for the HPL and HAL layers of various flash chips.

1. Introduction

Flash chips are a form of EEPROM (electrically-erasable, programmable read-only memory), distinguished by a fast erase capability. However, erases can only be done in large units (from 256B to 128kB depending on the flash chip). Erases are the only way to switch bits from 0 to 1, and programming operations can only switch 1's to 0's. Additionally, some chips require that programming only happen once between each erase, or that it be in relatively large units (e.g., 256B).

In the table below, we summarise these differences by categorising flash chips by their underlying technology (NOR vs NAND). We also include a column for Atmel's AT45DB flash chip family, as it has significantly different tradeoffs than other flash chips:

X	**NOR (ex: ST M25P40, Intel PXA27x)**	**AT45DB**	**NAND (ex: Samsung K9K1G08R0B)**
Erase	Slow (seconds)	Fast (ms)	Fast (ms)

... continued on next page

X	**NOR (ex: ST M25P40, Intel PXA27x)**	**AT45DB**	**NAND (ex: Samsung K9K1G08R0B)**
Erase unit	Large (64KB-128KB)	Small (256B)	Medium (8KB-32KB)
Writes	Slow (100s kB/s)	Slow (60kB/s)	Fast (MBs/s)
Write unit	1 bit	256B	100's of bytes
Bit-errors	Low	Low	High (requires ECC, bad-block mapping)
Read	Bus limited[*]	Slow+Bus limited	Bus limited
Erase cycles	10^4 - 10^5	10^4[†]	10^5 - 10^7
Intended use	Code storage	Data storage	Data storage
Energy/byte	1uJ	1uJ	.01uJ

The energy/byte is the per-byte cost of erasing plus programming. It is derived from the timing and power consumption of erase and write operations (for NOR flash, values are for the STMicroelectronics M25P family, for NAND flash, values are from the Samsung datasheet). Energy/byte for reads appears to depend mostly on how long the read takes (the power consumptions are comparable), i.e., on the efficiency of the bus + processor.

Early TinyOS platforms all used a flash chip from the AT45DB family. In TinyOS 1.x, this chip could be accessed through three different components:

- Using a low-level interface (`PageEEPROMC`) which gave direct access to per-page read, write and erase operations.
- Using a high-level memory-like interface (`ByteEEPROMC`) with read, write and logging operations.
- Using a simple file system (`Matchbox`) with sequential-only files [1].

Some more recent platforms use different flash chips: the ST M25P family (Telos rev. B, eyes) and the Intel Strataflash (Intel Mote2). None of the three components listed above are supported on these chips:

- The `PageEEPROMC` component is (and was intended to be) AT45DB-specific
- `ByteEEPROMC` allows arbitrary rewrites of sections of the flash. This is not readily implementable on a flash chip with large erase units.
- The `Matchbox` implementation was AT45DB-specific. It was not reimplemented for these other chips, in part because it does not support some applications (e.g., network reprogramming) very well.

[*] M25P40 reads are limited by the use of a 25MHz SPI bus. The PXA27x flash is memory mapped (reads are very fast and can directly execute code).

[†] Or infinite? Data sheet just says that every page within a sector must be written every 10^4 writes within that sector

2. Storage in TinyOS 2.x

One approach to hiding the differences between different flash chips is to provide a disk-like, block interface (with, e.g., 512B blocks). This is the approach taken by compact flash cards. However, in the context of TinyOS, this approach has several drawbacks:

- This approach is protected by patents, making it difficult to provide in a free, open-source operating system.
- To support arbitrary block writes where blocks are smaller than the erase unit, and to deal with the limited number of erase cycles/block requires remapping blocks. We believe that maintaining this remapping table is too expensive on many mote-class devices.

A second approach is to provide a generic low-level interface providing operations (read, write, erase) corresponding to the basic flash operations, along with information describing the flash chip's layout (minimum write and erase unit, timing information, etc). However, we believe that the large differences between NAND and NOR flash (see the table above), in particular the differences in reliability, write and erase units, make the design of a useful generic low-level interface tricky at best.

We thus believe it is best, for now at least, to define high-level storage abstractions that are useful for sensor network applications, and leave their implementation up to each flash chip - such abstractions will be necessary anyway. We leave open the possibility that a future TEP may define portable lower-level flash interfaces (either for all flash chips, or, e.g., for NOR-family flashes). Such low-level interfaces would allow implementations of the storage abstractions defined in this TEP to be used for multiple flash chips.

This TEP describes three high-level storage abstractions: large objects written in a single session, small objects with arbitrary reads and writes, and logs. TinyOS 2.x, divides flash chips into separate volumes (with sizes fixed at compile-time). Each volume provides a single storage abstraction (the abstraction defines the format).

We prefer the use of single abstractions over fixed-size volumes over the use of a more general filing system (like Matchbox) for several reasons:

- TinyOS is currently targeted at running a single application, and many applications know their storage needs in advance: for instance, a little space for configuration data, and everything else for a log of all sampled data. In such cases, the flexibility offered by a filing system (e.g., arbitrary numbers of files) is overkill,
- Each abstraction is relatively easy to implement on a new flash chip, and has relatively little overhead.
- The problem of dealing with the limited number of erase cycles/block is simplified: it is unlikely that user applications will need to rewrite the same small object 100'000 times, or cycle 100'000 times through their log. Thus the abstractions can mostly ignore the need for "wear levelling" (ensuring that each block of the flash is erased the same number of time, to maximise flash chip lifetime).

New abstractions (including a filing system) can easily be added to this framework.

The rest of this TEP covers some principles for the organisation of flash chips (Section 3), then describes the flash volumes and storage abstractions in detail (Section 4).

3. HPL/HAL/HIL Architecture

The flash chip architecture follows the three-layer Hardware Abstraction Architecture (HAA), with each chip providing a presentation layer (HPL, Section 3.1), adaptation layer (HAL, Section 3.2) and platform-independent interface layer (HIL, Section 3.3) [2]. The implementation of these layers SHOULD be found in the `tos/chips/CHIPNAME` directory. If a flash chip is part of a larger family with a similar interface, the HAA SHOULD support all family members by relying, e.g., on platform-provided configuration information.

Appendix A shows example HPL and HAL specifications for the AT45DB and ST M25P chip families.

3.1 Hardware Presentation Layer (HPL)

The flash HPL has a chip-dependent, platform-independent interface. The implementation of this HPL is platform-dependent. The flash HPL SHOULD be stateless.

To remain platform independent, a flash chip's HPL SHOULD connect to platform-specific components providing access to the flash chip; these components SHOULD be placed in the `tos/platforms/PLATFORM/chips/CHIPNAME` directory. If the flash chip implementation supports a family of flash chips, this directory MAY also contain a file describing the particular flash chip found on the platform.

3.2 Hardware Adaptation Layer (HAL)

The flash HAL has a chip-dependent, platform-independent interface and implementation. Flash families with a common HPL SHOULD have a common HAL. Flash HAL's SHOULD expose a `Resource` interface and automatically power-manage the underlying flash chip. Finally, the flash HAL MUST provide a way to access the volume information specified by the programmer (see Section 3). This allows users to build new flash abstractions that interact cleanly with the rest of the flash system.

3.3 Hardware Interface Layer (HIL)

Each flash chip MUST support at least one of the storage abstractions described in Section 4. These abstractions SHOULD be presented in components named `ChipAbstractionC`, e.g., `At45dbLogStorageC`. Additionally, a flash chip implementation MAY support platforms with multiple instances of the same storage chip. The way in which this is achieved is not specified further in this TEP.

Each platform MUST have `AbstractionC` components (e.g., `LogStorageC`) implementing the storage abstractions of Section 4 supported by its flash chip(s). On platforms with multiple storage chips SHOULD redirect uses of `AbstractionC` to the appropriate storage chip, based on the requested volume.

4. Non-Volatile Storage Abstractions

The HIL implementations are platform-independent, but chip (family) dependent. They implement the three storage abstractions and volume structure discussed in the introduction.

4.1. Volumes

The division of the flash chip into fixed-size volumes is specified by an XML file that is placed in the application's directory (where one types 'make'). The XML file specifies the allocation as follows:

```
<volume_table>
  <volume name="DELUGE0" size="65536" />
  <volume name="CONFIGLOG" size="65536" />
  <volume name="DATALOG" size="131072" />
  <volume name="GOLDENIMAGE" size="65536" base="983040" />
</volume_table>
```

The name and size parameters are required, while base is optional. The name is a string containing one or more characters in [a-zA-Z0-9_], while size and base are in bytes. Each storage chip MUST provide a compile-time tool that translates the allocation specification to chip-specific nesC code. There is no constraint on how this is done or what code is produced, except that the specification to physical allocation MUST be one-to-one (i.e. a given specification should always have the same resulting physical allocation on a given chip) and the result MUST be placed in the build directory. When not specified, the tool picks a suitable physical location for a volume. If there is any reason that the physical allocation cannot be satisfied, an error should be given at compile time. The tool SHOULD be named `tos-storage-CHIPNAME` and be distributed with the other tools supporting a platform. The XML file SHOULD be named `volumes-CHIPNAME.xml`.

The compile-time tool MUST prepend 'VOLUME_' to each volume name in the XML file and '#define' each resulting name to map to a unique integer.

The storage abstractions are accessed by instantiating generic components that take the volume macro as argument:

```
components new BlockStorageC(VOLUME_DELUGE0);
```

If the named volume is not in the specification, nesC will give a compile-time error since the symbol will be undefined.

A volume MUST NOT be used with more than one storage abstraction instance.

4.2 Large objects

The motivating example for large objects is the transmission or long-term storage of large pieces of data. For instance, programs in a network-reprogramming system, or large data-packets in a reliable data-transmission system. Such objects have an interesting characteristic: each byte in the object is written at most once.

This leads to the definition of the `BlockStorageC` abstraction for storing large objects or other "write-once" objects:

- A large object ranges from a few kilobytes upwards.
- A large object is erased before the first write.
- A sync ensures that a large object survives a reboot or crash
- Reads are unrestricted
- Each byte can only be written once between two erases

Large objects are accessed by instantiating a BlockStorageC component which takes a volume id argument:

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

The `BlockRead` and `BlockWrite` interfaces (briefly presented in Appendix B) contain the following operations (all split-phase, except `BlockRead.getSize`):

- `BlockWrite.erase`: erase the volume. After a reboot or a commit, a volume MUST be erased before it can be written to.
- `BlockWrite.write`: write some bytes starting at a given offset. Each byte MUST NOT be written more than once between two erases.
- `BlockWrite.sync`: ensure all previous writes are present on a given volume. Sync MUST be called to ensure written data survives a reboot or crash.
- `BlockRead.read`: read some bytes starting at a given offset.
- `BlockRead.computeCrc`: compute the CRC of some bytes starting at a given offset.
- `BlockRead.getSize`: return bytes available for large object storage in volume.

For full details on arguments and other considerations, see the comments in the interface definitions.

Note that these interfaces contain no direct support for verifying the integrity of the BlockStorage data, but such support can easily be built by using the `computeCrc` command and storing the result in a well-defined location, and checking this CRC when desired.

4.3 Logging

Event and result logging is a common requirement in sensor networks. Such logging should be reliable (a mote crash should not lose data). It should also be easy to extract data from the log, either partially or fully. Some logs are *linear* (stop logging when the volume is full), others are *circular* (the oldest data is overwritten when the volume is full).

The `LogStorageC` abstraction supports these requirements. The log is record based: each call to `LogWrite.append` (see below) creates a new record. On failure (crash or reboot), the log MUST only lose whole records from the end of the log. Additionally, once a circular log wraps around, calls to `LogWrite.append` MUST only lose whole records from the beginning of the log.

Logs are accessed by instantiating a LogStorageC component which takes a volume id and a boolean argument:

```
generic configuration LogStorageC(volume_id_t volid, bool circular) {
  provides {
      interface LogWrite;
      interface LogRead;
  }
} ...
```

If the `circular` argument is TRUE, the log is circular; otherwise it is linear.

The `LogRead` and `LogWrite` interfaces (briefly presented in Appendix B) contain the following operations (all split-phase except `LogWrite.currentOffset`, `LogRead.currentOffset` and `LogRead.getSize`):

- `LogWrite.erase`: erase the log. A log MUST be erased (possibly in some previous session) before any other operation can be used.
- `LogWrite.append`: append some bytes to the log. In a circular log, this may overwrite the current read position. In this case, the read position MUST be advanced to the log's current beginning (i.e., as if `LogRead.seek` had been called with `SEEK_BEGINNING`). Additionally, the `LogWrite.appendDone` event reports whenever, in a circular log, an append MAY have erased old records.

  Each append creates a separate record. Log implementations may have a maximum record size; all implementations MUST support records of up to 255 bytes.
- `LogWrite.sync`: guarantee that data written so far will not be lost to a crash or reboot (it can still be overwritten when a circular log wraps around). Using `sync` MAY waste some space in the log.
- `LogWrite.currentOffset`: return cookie representing current append position (for use with `LogRead.seek`).
- `LogRead.read`: read some bytes from the current read position in the log and advance the read position.

  `LogStorageC` implementations MUST include error detection codes to increase the likelihood of detection of corrupted or invalid log data. Data returned by a successful read MUST have passed this error detection check. The behaviour on failure of this check is unspecified (e.g., the at45db believes as if the end of the log has been reached; other implementations may behave differently).
- `LogRead.currentOffset`: return cookie representing current read position (for use with `LogRead.seek`).
- `LogRead.seek`: set the read position to a value returned by a prior call to `LogWrite.currentOffset` or `LogRead.currentOffset`, or to the special `SEEK_BEGINNING` value. In a circular log, if the specified position has been overwritten, behave as if `SEEK_BEGINNING` was requested.

  `SEEK_BEGINNING` positions the read position at the beginning of the oldest record still present in the log.

  After reboot, the current read position is `SEEK_BEGINNING`.
- `LogRead.getSize`: return an approximation of the log's capacity in bytes. Uses of `sync` and other overhead may reduce this number.

For full details on arguments, etc, see the comments in the interface definitions.

Note that while each call to `append` logically creates a separate record, the `LogStorageC` API does not report record boundaries. Additionally, crashes, reboots, and appends after wrap-around in a circular log can cause the loss of multiple consecutive records. Taken together, these restrictions mean that a `LogStorageC` implementation MAY internally aggregate several consecutive appends into a single record. However, the guarantee that only whole records are lost is sufficient to ensure that applications do not to have worry about incomplete or inconsistent log entries.

4.4 Small objects:

Sensor network applications need to store configuration data, e.g., mote identity, radio frequency, sample rates, etc. Such data is not large, but losing it may lead to a mote misbehaving or losing contact with the network.

The `ConfigStorageC` abstraction stores a single small object in a volume. It:

- Assumes that configuration data is relatively small (a few hundred bytes).
- Allows random reads and writes.
- Has simple transactional behaviour: each read is a separate transaction, all writes up to a commit form a single transaction.
- At reboot, the volume contains the data as of the most recent successful commit.

Small objects are accessed by instantiating a ConfigStorageC component which takes a volume id argument:

```
generic configuration ConfigStorageC(volume_id_t volid) {
  provides {
      interface Mount;
      interface ConfigStorage;
  }
} ...
```

A small object MUST be mounted (via the `Mount` interface) before the first use.

The `Mount` and `ConfigStorage` interfaces (briefly presented in Appendix B) contain the following operations (all split-phase except `ConfigStorage.getSize` and `ConfigStorage.valid`):

- `Mount.mount`: mount the volume.
- `ConfigStorage.valid`: return TRUE if the volume contains a valid small object.
- `ConfigStorage.read`: read some bytes starting at a given offset. Fails if the small object is not valid. Note that this reads the data as of the last successful commit.
- `ConfigStorage.write`: write some bytes to a given offset.
- `ConfigStorage.commit`: make the small object contents reflect all the writes since the last commit.

- `ConfigStorage.getSize`: return the number of bytes that can be stored in the small object.

For full details on arguments, etc, see the comments in the interface definitions.

5. Implementations

An AT45DB implementation can be found in tinyos-2.x/tos/chips/at45db.

An ST M25P implementation can be found in tinyos-2.x/tos/chips/stm25p.

6. Authors' Addresses

David Gay
2150 Shattuck Ave, Suite 1300
Intel Research
Berkeley, CA 94704

phone - +1 510 495 3055
email - david.e.gay@intel.com

Jonathan Hui
657 Mission St. Ste. 600
Arched Rock Corporation
San Francisco, CA 94105-4120

phone - +1 415 692 0828
email - jhui@archedrock.com

7. Citations

Appendix A. HAA for some existing flash chips

A.1 AT45DB

The Atmel AT45DB family HPL is:

```
configuration HplAt45dbC {
  provides interface HplAt45db;
} ...
```

The `HplAt45db` interface has flash->buffer, buffer->flash, compare buffer to flash, erase page, read, compute CRC, and write operations. Most of these operations are asynchronous, i.e., their completion is signaled before the flash chip has completed the

[1] David Gay. "Design of Matchbox, the simple filing system for motes. (version 1.0)."

[2] TEP 2: Hardware Abstraction Architecture.

operation. The HPL also includes operations to wait for asynchronous operations to complete.

A generic, system-independent implementation of the HPL (`HplAt45dbByteC`) is included allowing platforms to just provide SPI and chip selection interfaces.

Different members of the AT45DB family are supported by specifying a few constants (number of pages, page size).

The AT45DB HAL has two components, one for chip access and the other providing volume information:

```
component At45dbC
{
  provides {
    interface At45db;
    interface Resource[uint8_t client];
    interface ResourceController;
    interface ArbiterInfo;
  }
} ...

configuration At45dbStorageManagerC {
  provides interface At45dbVolume[volume_id_t volid];
} ...
```

Note that the AT45DB HAL resource management is independent of the underlying HPL's power management. The motivation for this is that individual flash operations may take a long time, so it may be desirable to release the flash's bus during long-running operations.

The `At45db` interface abstracts from the low-level HPL operations by:

- using the flash's 2 RAM buffers as a cache to allow faster reads and writes
- hiding the asynchronous nature of the HPL operations
- verifying that all writes were successful

It provides cached read, write and CRC computation, and page erase and copy. It also includes flush and sync operations to manage the cache.

The `At45dbVolume` interface has operations to report volume size and map volume-relative pages to absolute pages.

A.2 ST M25P

The ST M25P family HPL is:

```
configuration Stm25pSpiC {
  provides interface Init;
  provides interface Resource;
  provides interface Stm25pSpi;
}
```

The `Stm25pSpi` interface has read, write, compute CRC, sector erase and block erase operations. The implementation of this HPL is system-independent, built over a few system-dependent components providing SPI and chip selection interfaces.

Note that these two examples have different resource management policies: the AT45DB encapsulates resource acquisition and release within each operation, while the M25P family requires that HPL users acquire and release the resource itself.

The ST M25P HAL is:

```
configuration Stm25pSectorC {
  provides interface Resource as ClientResource[storage_volume_t volume];
  provides interface Stm25pSector as Sector[storage_volume_t volume];
  provides interface Stm25pVolume as Volume[storage_volume_t volume];
}
```

The `Stm25pSector` interface provides volume-relative operations similar to those from the HPL interface: read, write, compute CRC and erase. Additionally, it has operations to report volume size and remap volume-relative addresses. Clients of the ST M25P HAL must implement the `getVolumeId` event of the `Stm25pVolume` interface so that the HAL can obtain the volume id of each of its clients.

Appendix B. Storage interfaces

These interfaces are presented briefly here for reference; please refer to the TinyOS documentation for a full description of the commands, events and their parameters.

B.1 BlockStorage interfaces

The BlockStorage interfaces are:

```
interface BlockRead {
  command error_t read(storage_addr_t addr, void* buf, storage_len_t len);
  event void readDone(storage_addr_t addr, void* buf, storage_len_t len,
                      error_t error);

  command error_t computeCrc(storage_addr_t addr, storage_len_t len,
                             uint16_t crc);
  event void computeCrcDone(storage_addr_t addr, storage_len_t len,
                            uint16_t crc, error_t error);

  command storage_len_t getSize();
}

interface BlockWrite {
  command error_t write(storage_addr_t addr, void* buf, storage_len_t len);
  event void writeDone(storage_addr_t addr, void* buf, storage_len_t len,
                       error_t error);

  command error_t erase();
  event void eraseDone(error_t error);

  command error_t sync();
  event void syncDone(error_t error);
}
```

B.2 ConfigStorage interfaces

The ConfigStorage interfaces are:

```
interface Mount {
  command error_t mount();
  event void mountDone(error_t error);
}

interface ConfigStorage {
  command error_t read(storage_addr_t addr, void* buf, storage_len_t len);
  event void readDone(storage_addr_t addr, void* buf, storage_len_t len,
                      error_t error);

  command error_t write(storage_addr_t addr, void* buf, storage_len_t len);
  event void writeDone(storage_addr_t addr, void* buf, storage_len_t len,
                       error_t error);

  command error_t commit();
  event void commitDone(error_t error);

  command storage_len_t getSize();
  command bool valid();
}
```

B.3 LogStorage interfaces

The LogStorage interfaces are:

```
interface LogRead {
  command error_t read(void* buf, storage_len_t len);
  event void readDone(void* buf, storage_len_t len, error_t error);

  command storage_cookie_t currentOffset();

  command error_t seek(storage_cookie_t offset);
  event void seekDone(error_t error);

  command storage_len_t getSize();
}

interface LogWrite {
  command error_t append(void* buf, storage_len_t len);
  event void appendDone(void* buf, storage_len_t len, bool recordsLost,
                        error_t error);

  command storage_cookie_t currentOffset();

  command error_t erase();
  event void eraseDone(error_t error);
  command error_t sync();
  event void syncDone(error_t error);
}
```