# 6

# Survey of the State-of-the-Art in Flash-Based Sensor Nodes

Soledad Escolar Díaz, Jesús Carretero Pérez and Javier Fernández Muñoz
*Computer Science and Engineering Department. University Carlos III de Madrid, Madrid Spain*

## 1. Introduction

A wireless sensor network (WSN) is a distributed system composed of many battery-powered devices, which operate with no human intervention for long periods of time. These devices are called sensor nodes or *motes*.

Motes present features of both embedded and general-purpose systems (Han et al., 2005). Their tiny size, scarce resources, and their autonomous nature lead to strong restrictions of computation, communication, power, and storage. Typically, they are deployed in an *ad-hoc* fashion over a geographical area (e.g. a volcano, a glacier, an office), which is to be monitored. This means that —depending on the environment where they are installed— it could result very difficult to perform activities of maintenance such as the replacement of the node's batteries. Software built for the sensor nodes must be reliable and robust due to the difficulty for accessing sensor nodes, and sensor nodes must operate in an autonomous way even in presence of failures.

Motes are interconnected through wireless links and they execute a simple, small application, which is developed using a sensor node-specific operating system. Typically, sensor network applications consist of sensing the environment through different type of sensors (e.g. temperature, humidity, GPS, imagers), transforming analogical data into digital data in the node itself, and forwarding the data to the network. Data is forwarded through a multi-hop protocol to a special node denominated *gateway*, which is intended to redirect all data from the wireless network to a *base station* (e.g. PC, laptop), where the data will be permanently stored in order to allow data post-processing and analysis. Figure 1 shows the three elements previously described: sensor nodes, gateway, base station.

### 1.1 Data classification in a WSN

WSNs generate larger data sets as sampling frequency increase. Sensor nodes must manage data proceeding from different sources: *internal* data produced by the sensor node itself (e.g. sensor measurements, application data, logs), and *external* data transmitted by other nodes in the network (e.g. protocol messages, data packets, commands). Since the data

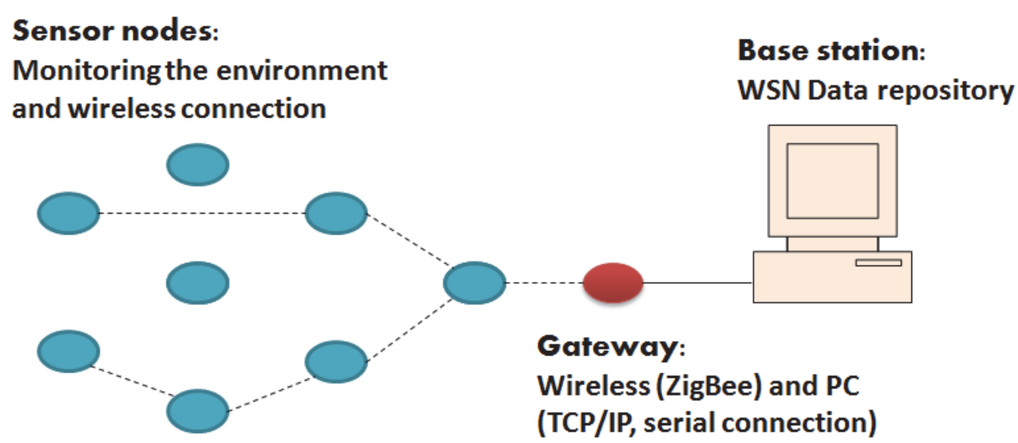



Fig. 1. A wireless sensor network: it is composed of a set of sensor nodes or motes, a communication gateway, and a base station.

memory[1] of a sensor node is a very scarce resource (typically 4 KB of RAM), nodes are forced to use other available devices to save data (such as the flash memory chip located outside the microcontroller) or to send data out of the node to prevent local storage. However, as explained below, the radio is the main consumer of energy in the sensor node and for this reason in many cases data are stored rather sent out. Subsequently, a tradeoff between the flash and radio power consumption is currently an important line of research (Diao et al., 2007) (Balani et al., 2005) (Shenker et al., 2003).

Other classification of data depends on the time when they were captured. In this sense, data may be *live* or *historical*. The first one corresponds to data acquired within a window of time and they are useful to detect meaningful events in quasi-real time (e.g. fire, earthquake). Moreover, in general, users do not want to renounce to the knowledge provided by the whole set of data, for example to identify trends or patterns and, therefore, historical data cannot be discarded. In this sense, flash memories can provide support for such an amount of data.

### 1.2 Importance of the flash memory chip in a sensor node

Flash memories have been embedded into sensor nodes from their earlier designs to nowadays. Along this time, these physical memories have suffered continuous updating, in order to be adapted to the specific features of sensor nodes. Specifically, they must be energy-efficient, since the energy is doubtless the most valuable and restricted resource. Many WSN applications found in the flash memory chip the only device that allow them satisfying their requirements, since it represents the device with the bigger capacity for permanent storage of application data in the sensor node.

There exist an increasing number of applications requiring the usage of non-volatile storage. Storing local and distributed data into sensor nodes has also promoted a set of high-level applications, which manage the network as a large database. Flash memories not just allow to store data when the RAM memory capabilities are near to be exceeded, but also they have made possible several relevant applications for sensor networks such as remote reprogramming of sensor nodes. Frequently, a WSN application could need both code

[1] The microcontroller of motes employ the Harvard Architecture which separates data and program into dedicated memories.

updates —for modifying the value of some variable— and important changes —as replacing the complete application's image. However, the unattended nature of sensor nodes, their ubiquity, and the inhospitable environments where they are deployed could difficult or even make impossible a manual installation of nodes. As an example, consider the extreme scenario where a sensor network has been affected by a virus disseminated from the base station. Another example more realistic is the necessity of degrading the application behaviour when the node's batteries are near to deplete, in order to increase the network lifetime. In both examples there is a necessity of reprogramming the network. Therefore, remote programming of sensor nodes is a fundamental task for ensuring the consistency of sensor network applications.

Consequently, flash memory chips, such as Atmel AT45DB, have become key devices that make possible a set of applications that currently are considered critical for wireless sensor networks.

This chapter presents a survey of the state-of-the-art in flash memories which are embedded into sensor nodes as external devices of general purpose. Along the chapter, we refer "flash memories" specifically to the flash memories which are external to microcontroller. At the beginning of the chapter we have presented a description of the technology of these flash memories, highlighting the more relevant features in the context of WSN, and their integration with other physical components hold into the sensor node. In the next section we describe the abstractions provided by several WSN-specific operating systems in order to manage the flash device. Then, we discuss the related work on flash-based sensor network applications, such as sensor nodes reprogramming and file systems. To conclude this chapter we provide our conclusions about this work.

## 2. Hardware technology

The hardware technology employed in sensor nodes manufacturing is an active research line that is carried out by both universities and by private companies around the world. The possibilities in this field are enormous because of the increasing need to look for new sensors for potential applications, advances in miniaturization, and the appearance of components to be integrated (e.g. GPS, scavengers). Since the sensor nodes are battery-powered devices, it is the most importance the looking for strategies at the hardware level that make an energy-efficient management of the devices.

The typical architecture of a *mote* presents the block diagram shown in Figure 2. It is composed of a set of hardware components which are described as follows:

- A microcontroller of low capacity which usually operates at very low frequencies (e.g. 7 MHz) and has an architecture ranging from 4-bit to 32-bit. It also integrates RAM and ROM memories, an Analogical-Digital Converter (ADC) and several *clocks* that enable local synchronizing. Some examples of microcontrollers are Atmega128L (Atmel, 2011) from ATMEL and MSP430 (Instrument, 2008) from Texas Instruments.
- The radio device provides wireless communication to the sensor node, and supports the WSN specific communication properties such as low energy, low data rate, and short distances. Some radio devices for motes are CC1000 (*CC1000 Single Chip Very Low Power RF Transceiver*, 2002) and CC2400 (*CC2400 2.4GHz Low-Power RF Transceiver*, 2003) from Chipcon, and

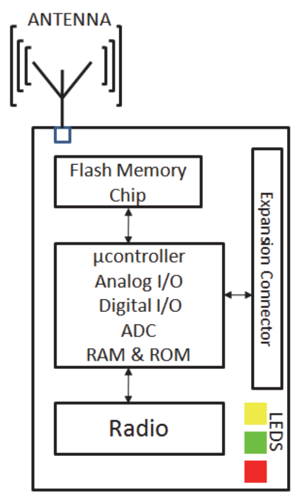



Fig. 2. Block diagram of a sensor node. It includes several interconnected physical devices such as the radio, the microcontroller and the flash memory chip.

nRF2401 (*nRF2401 Radio Transceiver Data Sheet*, 2003) radio transceiver from Nordic Semiconductors.

- The battery provides energy to the sensor node (e.g. alkaline batteries). Motes usually hold two conventional batteries as power supplier. Numerous research projects focus on alternatives for energy harvesting, which are typically based on solar cells.
- Several LEDs (Light-Emitting Diode) are attached to the mote board with the main purpose of helping to debug. Typically, there are three LEDs integrated into a sensor node (red, green and yellow) although in some motes, an additional blue LED has been added.
- A *sensor board* usually contains several sensors and actuators, which are able to sense the environment. When the sensor board is present, the *expansion connector* acts as a bridge between the sensor board and the mote microcontroller.
- Several *I/O buses* transfer internal data between physical components (microcontroller, radio, and memory) in accordance with a specific I/O protocol. Different interfaces coexist in a sensor node (e.g. Serial Peripheral Interface (SPI), Inter Integrated Circuit ($I^2C$) and Universal Asynchronous Receiver/Transmitter (UART)).
- Finally, an external *flash memory* with longer capacities than the internal memories (RAM and ROM), in order to temporally store data provided by different sources (sensors, network, or logs). Some of the most popular flash memory chips integrated into sensor nodes are Atmel AT45DB (*Atmel AT45DB011 Serial DataFlash*, 2001) and ST M25P40 (*M25P40 Serial Flash Memory*, 2002).

In the next subsection we will focus on describing the hardware technology for these flash memories embedded in sensor nodes.

### 2.1 Flash memory technology

Flash memory chips embedded within sensor nodes provide an additional and auxiliary storage space for general purpose usages. Flash memory is a specific type of EEPROM (Electrically Erasable Programmable Read-Only Memory) that enables the access to n-bytes blocks in a single operation —instead of one operation per byte like EEPROM memories— thus increasing the speed of the operations. This memory is non-volatile, which means that energy is not needed to maintain the information stored in the chip. For these reasons the usage of such a type of memory has been extended to many others digital devices like cameras, mobile phones, and MP3 players.

Physically, a flash memory chip consists of an array of memory cells which are manufactured using transistors. Every one of these cells is able to store one single bit (0 or 1) —in traditional chips— or a set of bits —in modern chips. Depending on the type of logical gate employed two underlying technologies can be distinguished:

- NOR flash is manufactured using NOR gates. Every cell in its default state is logically equivalent to the binary ”1” value. This is interpreted as presence of voltage in the cell. NOR flash was first introduced by Intel in 1988.
- NAND flash uses NAND gates. In this case, every cell is in its default state set to the equivalent binary ”0” value, which means that there is no voltage measured in that cell. NAND flash was introduced by Toshiba in 1989.

There is also a third type of technology used in the manufacturing of flash memory chips: the CMOS technology. CMOS enables to build logical gates in different way than NOR and NAND flash through a specific type of transistors: p-type and n-type *metaloxidesemiconductor* field-effect transistors.

Regardless of the underlying technology used, there exist two basic low-level operations that operate on a cell-basis: 1) the programming operation, which consists of inverting the default state of a cell; and 2) the erasing operation, which consists of resetting its default state. From these two operations most of high-level operations over the flash memory can be constructed.

### 2.2 Flash memory architecture

NOR flash memory architecture is organized in segments also called blocks or sectors. The operation of erasing is block-oriented since the minimum unit to be erased is a block. It means that all cells in this block must be erased together. The operation of programming can be generally performed on a per-byte basis, but it requires that the block to be modified be previously erased before of writing on it. Another feature is that it enables the random access for readings. Typical block sizes are 64, 128, or 256 KB. One example of NOR flash memory chip is the ST M25P40 (*M25P40 Serial Flash Memory*, 2002) which is hold into TelosB and Eyes sensor node platforms. This device has a capacity of 4 Mbit and it is organized into 8 sectors. Another example is Intel Strataflash (*Intel Strataflash*, 2002) which is integrated into Intel Mote2 sensor node. It is the lowest cost-per-bit NOR memory chip, with a capacity of 32 megabytes, which are divided into 128 KB sectors.

On the other hand, NAND flash memory is organized in blocks and pages. Each block is divided into a fixed number of pages and each page has n-bytes of extension where m bytes ($m < n$) are usually reserved for storing the metadata related to the data in that page (e.g. an error correcting code (ECC)). Typical page sizes are 512, 2048, or 4096 bytes, but in devices such as motes this length is even smaller. The high-level operations in a NAND flash —readings and writings— are typically performed on a per-page basis while the operation of erasing is performed on the whole block. The access in NAND memories differs from the random access in NOR memories. NAND memories enable direct access to the block level

but only sequential access is allowed inside a block. A representative example is Samsung K9K1G08R0B (SAMSUNG, 2003) with a capacity of 128 MB, a length of page of 528 bytes (for programming) and a block size of 16 KB (for erasing).
The most notable example of flash chip using CMOS technology is the Atmel AT45DB041 (*Atmel AT45DB011 Serial DataFlash*, 2001) chip, which is integrated into Mica family and TelosA motes. The total capacity for this chip is 512 KB and it is divided into four sectors of 128 KB. Every sector is also divided into pages, each page is 264 bytes long (256 bytes for data and 8 bytes for metadata). The pages can only be written or erased as a whole and in order to maintain the consistency, pages should be erased before being written.
Unlike conventional flash memories, that enable random access to the data, this memory chip uses a serial interface to enable sequential access. The memory uses two intermediate page long RAM buffers to transfer data between the serial interface and main memory. Every buffer is identified by a code which specifies what buffer is being used. These buffers perform a read-modify-write operation to effectively change the contents of flash. Figure 3 shows the block diagram for AT45DB041.

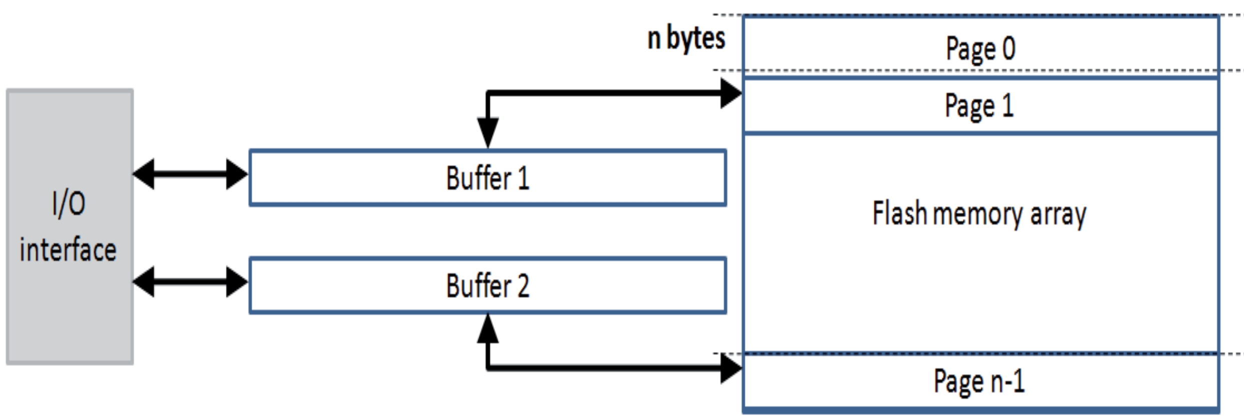



Fig. 3. Block diagram for AT45DB041 memory chip. It uses two page long RAM buffers to perform the operations on the memory.

Table 1 summarizes the features of the three technologies described above.

| Feature | NOR flash | AT45DB | NAND flash |
|---|---|---|---|
| Erase | Slow (seconds) | Fast (ms) | Fast (ms) |
| 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 |

Table 1. Features for different flash memory technologies: in the first column NOR technology (e.g ST M25P40 and Intel PXA27x), in the second column the AT45DB041 memory chip (CMOS technology), and finally in third column NAND technology (e.g. Samsung K9K1G08R0B).

### 2.3 Limitations of flash memory

One of the main limitations of the flash memory is that there exists a limit on the number of times a page can be written, typically around 10000 times. This feature makes necessary a mechanism which ensures a uniform distribution of writes over all the pages of a flash. This technique is called wear levelling. Wear levelling techniques should be implemented for preventing the usage of a page for the maximum number of times. Subsequently, an efficient management of the flash should have into account this feature.

Another aspect to be considered is the access time, which is comparable with the disk access time (in the order of milliseconds). However, flash write operations consume more time and energy than read operations, since they require to erase the page before being written. This feature has forced to develop different high-level techniques intended to minimize the number of times that one page goes to be written —which impacts also in those previously described related to wear levelling—. For example, a simple technique consists of managing a page long buffer with the data to be written and transfer it to the flash only when the buffer is complete. SENFIS file system which is described later employes this approach. When the buffer contents are downloaded to the flash memory, the buffer must be cleared in order to be used again.

### 2.4 Specific issues to sensor nodes

Since motes are battery-powered devices, an efficient power consumption policy is of critical importance in order to increase their lifetime. The motes typically operate in cycles of snoozing (low power mode), processing, and transmitting. Radio transmitting is the operation that consumes most energy. In fact, transmitting one bit consumes about as much power as executing 800-1000 instructions (Hill et al., 2000). The total energy consumption of a node is computed as the addition of the energy spent by each physical component: radio, sensors, leds, CPU, and flash memory. It is important to point out that, in order to perform efficiently their tasks, every physical component in the sensor node might stay in different states for a time, and each state has a specific current draw, which is generally supplied by the manufacturer. Therefore, the consumption due to a physical component, for instance the radio, is the sum of the current draws corresponding to each one of the states.

Flash memory chips distinguish several states for representing the activity to perform. For example AT45DB041 presents the following four states: standby, read, write, and load. The first one and the last one are the lowest consumption states —here the devices act as if disconnected and no operation can be performed in these states— while the other two indicate reading and writing respectively. The current draws for every state expressed in milliamperes (mA) are $2 \times 10^{-3}$, 4, 15, and $2 \times 10^{-3}$ respectively. Subsequently, if the application access pattern to flash memory is known, in particular the time spent in every flash state, it is possible to compute easily the energy model for the node's flash as:

$$E_{flash} = E_{standby} * T_{standby} + E_{read} * T_{read} + E_{write} * T_{write} + E_{load} * T_{load}$$

where $T_i$ corresponds to the time spent in `i` state and $E_i$ corresponds to the current draw in `i` state.

## 3. Operating system support

Operating systems (OSes) specifically designed to meet the WSN applications requirements and the sensor nodes restrictions constitute the backbone of the software architecture. The main challenge of the WSN operating systems is to manage the scarce hardware resources of the sensor nodes in an efficient and energy-aware way. OSes use the low-level interface (from HAL or hardware layer) to compose bigger grained operations, which are exported to the upper levels through a well-defined interface. Thus, OS abstractions are intended to mask the complexity of the hardware levels and facilitate the programming. High-level applications will use these abstractions that the OS provides to access the hardware resources in a transparent, simple way. Figure 4 depicts the software architecture of a sensor node.

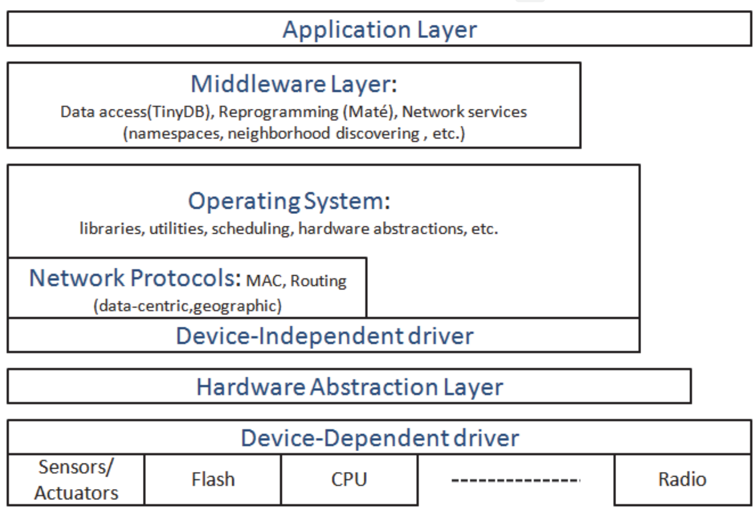



Fig. 4. A multi-layer software architecture for sensor nodes.

WSN OSes are very heterogeneous. They present one of the two following execution models: event- and thread-based. This is a relevant feature because it determines the programming model used and, subsequently, the interface provided by every OS is strongly coupled to its execution model. In this section we review the abstractions provided by a set of representative, popular operating systems in order to enable the flash memory access.

### 3.1 TinyOS 1.x

TinyOS (Hill et al., 2000) is the *de facto* standard for WSN operating systems, multi-platform, and open-source. Its execution model is event-based, but it presents some differences with regard to the pure model; in particular it distinguishes two scheduling levels: events and tasks. Events are preemptive and, therefore, they can be viewed as highest priority functions. On the other hand, there exists a second scheduling level, the tasks that can be viewed as non-preemptive functions (of lower priority) that run to completion. Tasks can only be interrupted by events, but not by other tasks.

TinyOS is written in nesC (Gay et al., 2003), a component-based programming language based on C that allows programming interfaces and components. The components in nesC communicate between each other through bi-directional interfaces. A nesC application is built through a configuration component, which declares the components and their connecting interfaces. A TinyOS application can be viewed as a combination of configuration and implementation components, whose behaviour and state is distributed through several interconnected components. Using this paradigm, TinyOS provides the implementation of hardware- and OS-level components as well as flexible abstractions to be used by the application level. Figure 5 shows an example of TinyOS application.

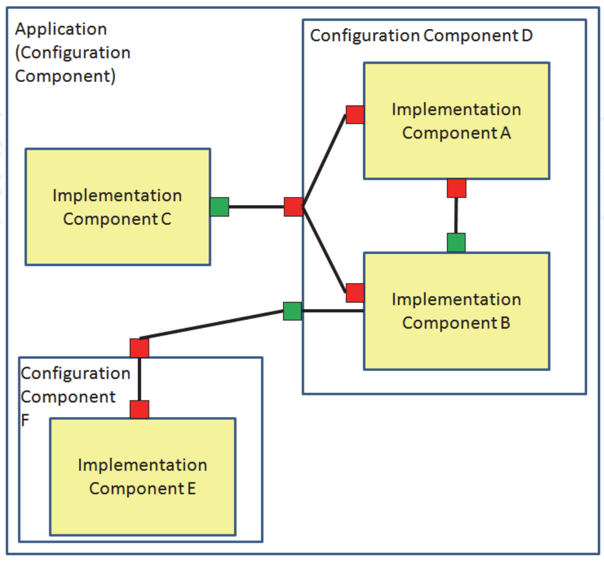



Fig. 5. A simple TinyOS application. It is composed of a set of components interconnected through interfaces, where one component provides the interface's implementation and the other one uses it. Configuration components allow encapsulate several components within it to build high-level components.

There exist two versions of TinyOS with substantial differences. We focus on describing the specific differences related to the flash memory management that they do. The first version of TinyOS provided three different interfaces to access data stored in the flash memory chip, every one of them located at different abstraction levels, from the lowest to the highest level:

1. A low-level implementation, based on pages (PageEEPROM).
2. A high-level memory-like interface, based on bytes (ByteeEEPROM).
3. A simple, basic file system (Matchbox).

### 3.1.1 The low-level implementation

The lowest level abstraction provides flash memory access on a per-page basis and it gives direct access to per-page read, write, and erase operations. The implementation is composed of several files:

- An interface file that includes the prototype of a set of commands (or functions) and events. Commands in TinyOS are functions that are implemented by the same or other component. For example, read and write operations over the flash memory take in TinyOS the shape of commands. On the other hand, in an event-based system events are signalled when an occurrence of some action takes place. Specifically, for the operations with high hardware latencies —which means that the operation response will be available only a time later— TinyOS requires declaring the corresponding event in the interface. When the response is obtained, the event is signalled from the hardware level to upper levels including also the

| Commands |
|---|
| result_t write(eeprompage_t page, eeprompageoffset_t offset, void *data, eeprompageoffset_t n) |
| result_t erase(eeprompage_t page, uint8_t eraseKind) |
| result_t sync(eeprompage_t page) |
| result_t syncAll(void) |
| result_t flush(eeprompage_t page) |
| result_t flushAll(void) |
| result_t read(eeprompage_t page, eeprompageoffset_t offset, void *data, eeprompageoffset_t n) |
| result_t computeCrc(eeprompage_t page, eeprompageoffset_t offset, eeprompageoffset_t n) |
| Events |
| result_t writeDone(result_t result) |
| result_t eraseDone(result_t result) |
| result_t syncDone(result_t result) |
| result_t flushDone(result_t result) |
| result_t readDone(result_t result) |
| result_t computeCrcDone(result_t result, uint16_t crc) |

Table 2. Low-level interface for flash memory access provided by TinyOS 1.x.

operation result. Note that in this latter case, another high-level component should provide one implementation for each event. Table 2 shows the API provided at this abstraction level. It includes a very reduced set of basic operations for reading, writing, and erasing; additionally it allows to compute the CRC for a memory page.

- A set of TinyOS components that provides the implementation for the API shown in Table 2. This implementation must invoke the hardware-level drivers which carry out the operations at the physical level. Several issues are left to the application level: firstly, flash memory must be accessed with mutual exclusion and therefore two operations over flash cannot be issued at the same time; secondly, the application must implement the event handlers corresponding to the commands that it invokes; thirdly, the high-level application must specify as arguments low-level details, such as the number of page to be read or written as well as the offset within the page. For instance, if the memory has a capacity of 512 KB —like AT45DB041 memory flash— there exists 2048 pages every one of them with 264 bytes to be read or written; both data must be specified for each operation on the flash.

#### 3.1.2 High-level implementation

This interface provides read, write, and logging operations. The implementation operates on a per-byte basis and thus the operations must specify what is the position or offset in bytes from which data go to be read or written. This offset is expressed as an absolute value with regard to the total capacity of the flash memory. For instance, if the memory has a capacity of 512 KB the offset should ranges between 0x000000 and 0x07FFFF. In this sense, the abstraction level is still very low because it forces to the programmer to manage hardware details in order to invoke the operations. Table 3 shows the interface provided for this approach.

#### 3.1.3 Matchbox

The idea of providing the file abstraction to access the data stored is a very useful approach typically performed at the operating system level. This approach is very attractive for users because it prevents them of managing hardware-level information such as offsets or number of pages. Matchbox (Gay, 2003) is the first file system developed for sensor nodes and it was

| Commands |
|---|
| command result_t read(uint32_t offset, uint8_t* buffer, uint32_t numBytesRead); |
| command result_t write(uint32_t offset, uint8_t *data, uint32_t numBytesWrite); |
| command result_t erase(); |
| command result_t append(uint8_t* data, uint32_t numBytes); |
| command uint32_t currentOffset(); |
| command result_t request(uint32_t numBytesReq); |
| command result_t sync(); |
| command result_t requestAddr(uint32_t byteAddr, uint32_t numBytesReq); |
| **Events** |
| event result_t readDone(uint8_t* buffer, uint32_t numBytesRead, result_t success); |
| event result_t writeDone(uint8_t *data, uint32_t numBytesWrite, result_t success); |
| event result_t eraseDone(result_t success); |
| event result_t appendDone(uint8_t* data, uint32_t numBytes, result_t success); |
| event result_t syncDone(result_t success); |
| event result_t requestProcessed(result_t success); |

Table 3. High-level interface for flash memory provided by TinyOS 1.x.

included into the first version of TinyOS. The major goals of Matchbox are reliability (detection of data corruption) and low resource consumption. Matchbox offers operations for directories and files. Files are unstructured and are represented simply as a sequence of bytes. Matchbox allows the applications to open two files simultaneously, but it supports only sequential reads and appending writes and it does not allow random access to files. It also provides a simple wear levelling policy. The Matchbox code size is small, around 10 KB. The minimum footprint is 362 bytes and it increases when the number of files grows. For each flash memory page an 8-byte CRC is used to verify the integrity of the file during recovery from a system crash.
Table 4 shows the complete interface provided by the Matchbox file system. As shown, it is composed of a reduced number of primitives to manage the data stored in the flash using the file abstraction. In this sense, a file is a stream flow that can be read and updated. Additional operations on a file are renaming and deleting a file. Note that the writing operation only enables adding data at the end of the file. The advantages of this approach is that the user manages entities that are identified through file names in order to access data; subsequently, users do not need to be conscious about low-level details such as the number of the page to be read or written.

### 3.2 TinyOS 2.x

The second version of TinyOS focuses on designing high-level portable interfaces while the implementation is leaved up to the manufacturers. This approach satisfies the design principles of TinyOS 2.x (Handziski et al., 2005), where a layered design of the software architecture is proposed with the final goal of achieving portability. Subsequently, the abstractions provided by TinyOS 2.x are high-level and platform-independent interfaces, while their low level implementation is platform-specific. This represents an important difference with regard to the previous version of TinyOS, where both the interface and the implementation are specific-device. TinyOS 2.x proposes four non-volatile storage entities, every one of them is intended for store data with different nature and requirements:

- Volumes are fixed-size units in which the flash memory is organized for general purposes.

| Commands |
|---|
| command result_t delete(const char *filename); |
| command result_t start(); |
| command result_t readNext(); |
| command uint32_t freeBytes(); |
| command result_t open(const char *filename); |
| command result_t close(); |
| command result_t read(void *buffer, filesize_t n); |
| command result_t getRemaining(); |
| command result_t rename(const char *oldName, const char *newName); |
| command result_t append(void *buffer, filesize_t n); |
| command result_t reserve(filesize_t newSize); |
| command result_t sync(); |
| **Events** |
| event result_t deleted(fileresult_t result); |
| event result_t ready(); |
| event result_t nextFile(const char *filename, fileresult_t result); |
| event result_t opened(fileresult_t result); |
| event result_t closed(fileresult_t result); |
| event result_t readDone(void *buffer, filesize_t nRead, fileresult_t result); |
| event result_t remaining(filesize_t n, fileresult_t result); |
| event result_t renamed(fileresult_t result); |
| event result_t appended(void *buffer, filesize_t nWritten, fileresult_t result); |
| event result_t reserved(filesize_t reservedSize, fileresult_t result); |
| event result_t synced(fileresult_t result); |

Table 4. Matchbox interface provided by TinyOS 1.x.

- Large objects that may occupy an undetermined space and they are intended to store a great amount of data, for instance a binary image that is received from the radio device.
- Loggers are intended to store records of fixed size, such as results and events.
- Small objects (of a few hundred bytes) with a transactional behaviour.

### 3.2.1 Volumes

Flash chip is divided in several volumes whose size must be specified at compilation time. The properties of the volumes are specified using an XML file where three data are specified per each volume: name, size, and base (optional). Note that such description is platform-independent. As an example consider the next XML file where four volumes of 65536 bytes are defined:

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

When the application that defines the volumes configuration is compiled, the chip-specific implementation translates such configuration to the equivalent nesC code that must allocate the space for every volume. There is no restriction about how this translation should be done. Applications can simply use every one of the volumes previously defined instantiating the

| Commands |
|---|
| command error_t read(storage_addr_t addr, void* buf, storage_len_t len); |
| command error_t computeCrc(storage_addr_t addr, storage_len_t len, uint16_t crc); |
| command storage_len_t getSize(); |
| command error_t write(storage_addr_t addr, void* buf, storage_len_t len); |
| command error_t erase(); |
| command error_t sync(); |
| **Events** |
| event void readDone(storage_addr_t addr, void* buf, storage_len_t len, error_t error); |
| event void computeCrcDone(storage_addr_t addr, storage_len_t len, uint16_t crc, error_t error); |
| event void writeDone(storage_addr_t addr, void* buf, storage_len_t len, error_t error); |
| event void eraseDone(error_t error); |
| event void syncDone(error_t error); |

Table 5. Interface for volumes and large objects provided by TinyOS 2.x.

generic component `BlockStorageC` which receives as argument the name of the volume to be used. Data stored in the volume can be read, written, and erased. Table 5 shows the interface to be used in order to access volumes. Note that the application specifies the relative address to the volume on which the operation is done.

### 3.2.2 Large objects

Large objects are a specific type of data with an interesting semantic: each byte in the object is written at most once. These data are written once and rarely it goes to be overwritten. In the WSN field, there are several examples of this type of data: considers for example binary files for network reprogramming or reliable packet whose contents must keep invariable. TinyOS 2.x provides the same interface for large objects than for volumes (see Table 5). In the same way, an instance of the generic component `BlockStorageC` must be created in order to access the large object.

### 3.2.3 Loggers

Storing the internal data generated in the sensor node itself is a common requirement for many WSN applications. Consider for example the need of scientists to know with a certain accuracy the value of the sensor readings in order to extract patterns that can help to predict some event. Such a logging should be reliable since data should not be lost and they should survive to a crash or reboot. Logs can be defined as linear —data are written sequentially from the beginning at the end of the log— or circular —if the log is full the data overwritten the beginning of the log—. Loggers access —both linear and circular— is always sequential. Loggers work on a per-record basis, where one record is the logic data structure to be read or written in every operation. The commit operation ensures that the data committed are successfully stored in flash and that they can be therefore recovered. Data not committed do not ensure this feature. Table 6 depicts the interface for loggers provided by TinyOS 2.x. In order to access logs the application instantiates the `LogStorageC` component, which takes two input arguments: a volume identifier and a boolean argument specifying whether the log is circular or not.

| Commands |
|---|
| command error_t read(void* buf, storage_len_t len); |
| command storage_cookie_t currentOffset(); |
| command error_t seek(storage_cookie_t offset); |
| command storage_len_t getSize(); |
| command error_t append(void* buf, storage_len_t len); |
| command error_t erase(); |
| command error_t sync(); |
| **Events** |
| event void readDone(,void* buf, storage_len_t len, error_t error); |
| event void seekDone(error_t error); |
| event void appendDone(void* buf, storage_len_t len, bool recordsLost, error_t error); |
| event void eraseDone(error_t error); |
| event void syncDone(error_t error); |

Table 6. Interface for loggers provided by TinyOS 2.x.

| Commands |
|---|
| command error_t read(storage_addr_t addr, void* buf, storage_len_t len); |
| command error_t write(storage_addr_t addr, void* buf, storage_len_t len); |
| command error_t commit(); |
| command storage_len_t getSize(); |
| command bool valid(); |
| command error_t mount(); |
| **Events** |
| event void readDone(storage_addr_t addr, void* buf, storage_len_t len, error_t error); |
| event void writeDone(storage_addr_t addr, void* buf, storage_len_t len, error_t error); |
| event void commitDone(error_t error); |
| event void mountDone(error_t error); |

Table 7. Interface for small objects provided by TinyOS 2.x.

#### 3.2.4 Small objects

Some sensor network applications need to store their configuration. This configuration includes the initial data to be assigned to the application variables such as the mote identity, sampling rates, or thresholds. These critical data must be stored in a non-volatile support in order to be sure that on sensor failures —reboot or crash— the configuration can be recovered. A characteristic of this type of data is its small size, frequently of a few hundred bytes. Another interesting feature is the transactional behaviour of the operations performed on this type of data: each read is a separate transaction, all writes up to a commit defines a single transaction. Table 7 presents the interface provided by TinyOS 2.x for small objects. The application must instantiate the generic component `ConfigStorageC` previously to the data access. As shown, the operation must specify the address to be read or written.

### 3.3 Contiki

Contiki (Dunkels et al., 2004) is an operating system designed for networked and memory constrained systems, developed in the Swedish Institute of Computer Science (SICS) by Adam Dunkels as the leader of the project in 2003. Contiki is open source and it was written in the C programming language. The typical size of Contiki applications is around kilobytes, which means a bigger footprint than the applications developed in TinyOS. In spite of this, it could be considered the second most extended operating system for programming sensor nodes. At the

| Functions | Description |
|---|---|
| int cfs_open (const char *name, int flags); | Open a file |
| void cfs_close (int fd); | Close an open file |
| int cfs_read (int fd, void *buf, unsigned int len); | Read data from an open file |
| int cfs_write (int fd, const void *buf, unsigned int len); | Write data to an open file |
| cfs_offset_t cfs_seek (int fd, cfs_offset_t offset, int whence); | Seek to a specified position in an open file |
| int cfs_remove (const char *name); | Remove a file |
| int cfs_opendir (struct cfs_dir *dirp, const char *name); | Open a directory for reading directory entries |
| int cfs_readdir (struct cfs_dir *dirp, struct cfs_dirent *dirent); | Read a directory entry |
| void cfs_closedir (struct cfs_dir *dirp); | Close a directory opened with cfs_opendir() |

Table 8. Coffee file system interface.

operating system level Contiki provides an only way for accessing data: a file system. This file system is called Coffee (Contiki's Flash File System (Coffee)) (Tsiftes et al., 2009). Coffee is a flash-based file system, designed as a combination of extents and *micro log* files. The concept of micro log files is introduced to record file modifications without requiring a high consumption of memory space. In fact, every open file uses a small and constant memory footprint. Coffee provides POSIX-style primitives to manage both files and directories (see Table 8). Other outstanding features of Coffee are garbage collection, wear levelling techniques in order to avoid memory corruption, and fault recovery.

### 3.4 LiteOS

LiteOS (Cao et al., 2008) is a UNIX-like multi-threaded operating system with object-oriented programming support for wireless sensor networks. It includes several features of the Unix systems (e.g. a shell or the programming environment), which increase its footprint leaving it too far from operating systems such as TinyOS. LiteOS includes a built-in hierarchical file system called LiteFS (Cao & Abdelzaher, 2006). LiteFS supports both file and directory operations, and opened files are kept in RAM. Directory information is stored in the EEPROM while the serial flash stores file metadata. LiteFS implements two wear levelling techniques, one for the EEPROM chip and the other one for the serial flash. LiteOS provides also a UNIX-like shell that facilitates the interaction of the user with the file system by using UNIX-based commands (e.g. ls, cd, cp, mv, rm). The complete set of LiteFS primitives is presented in Table 9. As shown, there are basic primitives for managing files and directories (e.g. open, close, read, and write) as well as for administrating the sensor node (e.g. checking and formatting the EEPROM and flash memories).

### 3.5 Comparison

The three implementations provided by TinyOS 1.x are AT45DB-specific and subsequently in this first approach the portability was sacrificed. In general, the abstraction level offered by the OS is very low even when bigger grained entities such as files are managed. The applications are forced to be worried about hardware details (number of the page, offset), which makes programming complex and error-prone. Clearly the advantage of this approach is that it prevents the overload imposed by the management at the OS level. In TinyOS 2.x the major goal was increasing the portability though richer interfaces that are platform-independent. TinyOS 2.x renounces to offer a general implementation at the OS level due to the heterogeneity of the different flash devices. The interfaces provided are recommended for a particular type of data: small objects, volumes, loggers and large object.

| Functions | Description |
|---|---|
| FILE* fopen(const char *pathname, const char *mode); | Open file |
| int fclose(FILE *fp); | Close file |
| int fseek (FILE *fp, int offset, int position); | Seek file |
| int fexist(char *pathname); | Test file/directory |
| int fcreatedir(char *pathname); | Create directory file |
| int fdelete(char *pathname); | Delete file/directory |
| int fread(FILE *fp, void *buffer, int nBytes); | Read from file |
| int fwrite(FILE *fp, void *buffer, int nBytes); | Write to file |
| int fmove(char *source, char *target); | Move file/directory |
| int fcopy(char *source, char *target); | Copy file/directory |
| void formatSystem(); | Format file system |
| void fchangedir(char *path); | Change current directory |
| void fcurrentdir(char *buffer, int size); | Get current directory |
| int fcheckEEPROM(); | Check EEPROM Usage |
| int fcheckFlash(); | Check Flash Usage |
| void fsearch(char *addrlist, int *size, char *string); | Search by name |
| void finfonode(char *buffer, int addr); | Get file/directory info |

Table 9. LiteFS file system interface.

However, as deduced from their interfaces the abstraction level of the application is low since they must specify again hardware details for accessing data.
TinyOS 1.x and other operating systems as Contiki and LiteOS provide abstractions in the shape of file systems to data access. The advantages of this approach is that the user manages entities that are identified through file names in order to access data; subsequently, users do not need to be conscious about low-level details such as the number of the page to be read or written. It definitively facilitates the programming and prevents errors. Table 10 shows a brief comparison among the file systems studied in this section and in the following section.

| | ELF | Matchbox | LiteFS | SENFIS |
|---|---|---|---|---|
| 1 | TinyOS | TinyOS | LiteOS | TinyOS |
| 2 | Mica2 | Mica Family Motes | MicaZ | Mica Family Motes |
| 3 | Dynamic | Static | Dynamic | Dynamic |
| 4 | RAM, EEPROM, Flash | Flash | RAM, EEPROM, Flash | RAM, EEPROM, Flash |
| 5 | 14 bytes (per flash page) | 8 bytes (per flash ge) | 8 bytes (per flash page) | 8 bytes (per flash page) |
| | 14 bytes | | 168 bytes RAM | 1062 bytes flash |
| | 14 bytes per i-node (RAM) | | 2080 bytes ROM | 208 bytes RAM/EEPROM |
| 6 | Unlimited | 2 (Read/Write) | 8 | 64 |
| 7 | Sensor Data | Data files | Data | Data stream |
| | Configuration Data | | Binary applications | Binary applications |
| | Binary program Image | | Device Drivers | |

Table 10. Comparison among different file systems for sensor nodes. Features: 1:Operating system; 2:Sensor platforms; 3:Memory allocation; 4:Memory chips used; 5:Metadata size; 6:Number of files opened; 7:Types of files.

## 4. Taxonomy of applications

Wireless sensor network applications are often multi-disciplinary and obey many types of requirements. Several authors have classified WSN applications according to their application domain (Akyildiz et al., 2002; Xu, 2002). In this section we will focus on those WSN applications that use the flash memory chip to carry out their operations. Thus, we distinguish three main type of applications: 1) file systems to store both internal and external data; 2) data-centric middlewares that provide an abstraction of the sensors network as a long database; and 3) applications for network reprogramming. These three types of applications use the flash memory chip as data storing support. Note that in a four category should appear the applications that use the flash for specific purposes. Figure 6 shows this classification. In following subsections we review some relevant examples in each category.

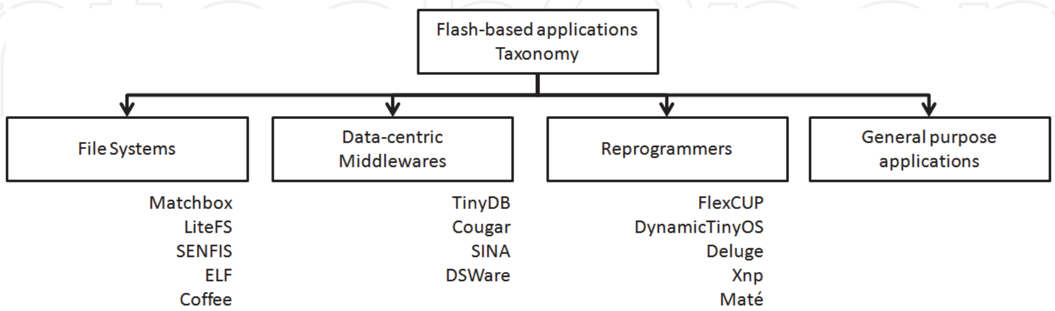



Fig. 6. A classification of applications that use the flash memory chip.

### 4.1 File systems

In addition to the OS-specific file systems presented in the previous section, we review here two file systems that were designed with no regard to be OS-independent: ELF and SENFIS. The usage of file systems is justified: the continuous data production through a wide set of versatile applications drives researchers to think about different methods of data storing and recovering, which can provide an efficient abstraction to give persistent support to the data generated into the sensor node.

#### 4.1.1 ELF

ELF (Dai et al., 2004) is a file system for WSNs based on the log file system paradigm (Kawaguchi et al., 1995). The major goals of ELF are memory efficiency, low power operation, and support for common file operations (such as reading and appending data to a file). The data to be stored in files are classified into three categories: data collected from sensors, configuration data, and binary program images. The access patterns and reliability requirements of these categories of data are different. Typically, the reliability of sensor data is verified through the CRC checksum mechanism. For binary images a greater reliability may be desirable, such as recovery after a crash. Typically, traditional log-structured file systems group log entries for each write operation into a sequential log. ELF keeps each log entry in a separate log page due to the fact that, if multiple log entries are stored on the same page, an error on this page will destroy all the history saved until that moment. ELF also provides a simple garbage collection mechanism and crash recovery support.

#### 4.1.2 SENFIS

SENFIS (Escolar et al., 2008; 2010) is a file system designed for Mica family motes and intended to be used in two scenarios: firstly, it can transparently be employed as a permanent storage for distributed TinyDB queries (see next subsection), in order to increase their reliability and scalability; secondly, it can be directly used by a WSN application for

| Primitive Prototype | Description |
|---|---|
| int8_t open (char *filename, uint8_t mode) | Open a file |
| result_t close (uint8_t fd) | Close a file |
| int8_t write (uint8_t fd, char *buffer, int8_t length) | Append data to a file |
| int8_t read (uint8_t fd, char *buffer, int8_t length) | Read from a file |
| result_t rename(char *oldname, char *newname) | Rename a file |
| result_t lseek (uint8_t fd, uint32_t ptr) | Update the offset of a file |
| result_t stat(uint8_t fd, struct inode *inode) | Obtain metadata of a file |
| result_t delete (uint8_t fd) | Delete a file |

Table 11. Basic high-level interface for SENFIS.

permanent storage of data on the motes. SENFIS uses the flash for persistent storage and RAM as a volatile memory. The flash chip is divided into blocks called segments, whose pages are accessed in a circular way, guaranteeing an optimal intra-segment wear levelling. The global wear-levelling is a best-effort algorithm: a newly created file is always assigned the lowest used segment.

In SENFIS, the flash is organized in segments. For instance, for AT45DB041 the flash may consist of 64 segments of 32 pages each. Each segment may be assigned to at most one file but a file can use an arbitrary number of segments. A segment is written always sequentially in a circular way. For implementing this behaviour a pointer to the last written page is kept in the segment metadata structure which is stored in a *segment table*. Every segment in this table records a pointer to the first page of the segment, a pointer to the next segment as well as a counter indicating the number of times the pages of this segment have been written. To minimize the number of times that a page flash is accessed the reading and writing operations use an intermediate cache such as shown in Figure 7. SENFIS provides a POSIX-style interface which is shown in Table 11.

SENFIS uses a writing buffer to reduce the number of times that a page is accessed. Figure 7 shows graphically this behaviour.

### 4.2 Data-centric middlewares

The most common approach to bridge the gap between the applications and low-level software, has been to develop a middleware layer mapping one level into the other. A survey of middleware is given in (Marrón, 2005) where a taxonomy of middlewares is discussed. In particular, authors identify data-centric middlewares as those ones that operate the sensor network as a database abstraction. Most of them rely on some form of SQL-like language in order to recover the data stored in different memories within the sensor node (RAM, EEPROM, and external flash). There exist different data-centric middlewares such as Cougar (Fung et al., 2002), TinyBD (Madden et al., 2005), DSWare (Li et al., 2003) and SINA (Jaikaeo et al., 2000)); some of them are summarized in the following paragraphs.

#### 4.2.1 TinyDB

TinyDB (Madden et al., 2005) focuses on acquisitional query processing techniques which differ from other database query techniques for WSN in that it does not simply postulate the a priori existence of data, but it focuses also on location and cost of acquiring data. The acquisitional techniques have been shown to reduce the power consumption in several orders of magnitude and to increase the accuracy of query results. A typical query of TinyDB is active

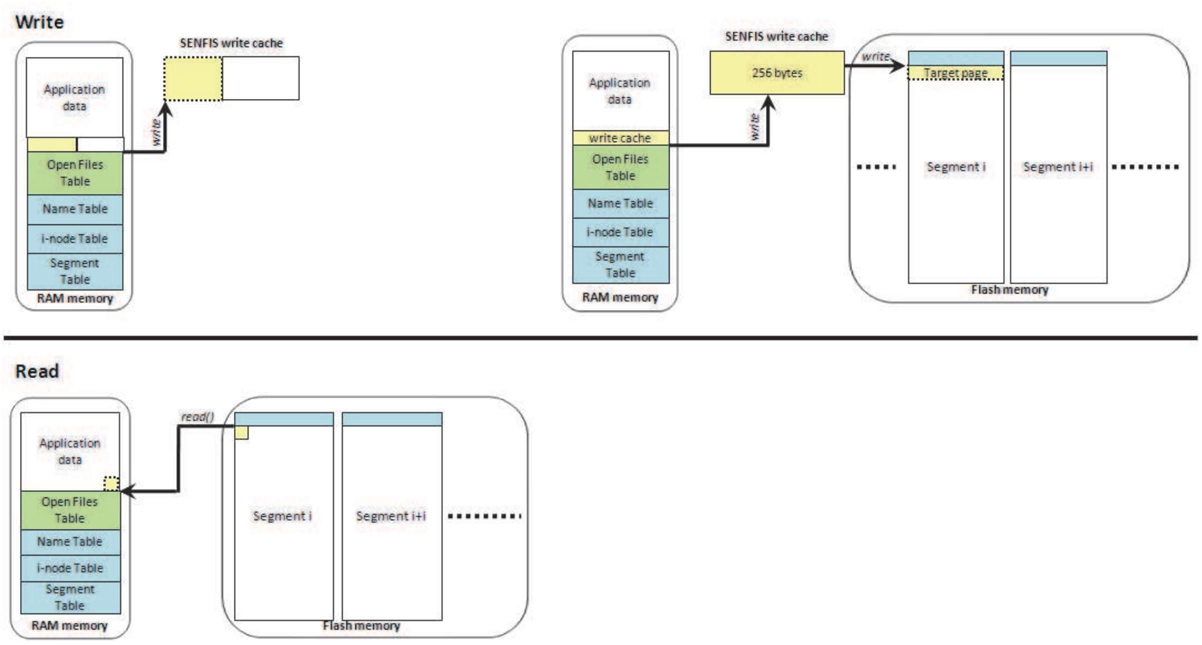



Fig. 7. Writing and reading operations in SENFIS: 1) Above, the writing operation which appends data to the end of a file. The modification is done in a small buffer cache in RAM and it is committed to the flash either when a page is completely written or when the RAM is full. The first case tries to avoid that a page is committed to flash several times for small writes; 2) below, the reading operation which get the data from the flash to an application buffer. If the data is already in the small buffer cache, it is copied to the application buffer from there.

in a mote for a specified time frame and is data intensive. The results of a query may produce communication or be temporarily stored in the RAM memory. In TinyDB the sampled values of the various sensor attributes (e.g. temperature, light) are stored in a table called sensors. The columns of the table represent the sensor attributes and the rows the instant of time when the measure was taken. Projections and transformations of sensor table are stored in materialization points. A materialization point is a type of temporal table that can be used in subsequent select operations. Materialization points are declared by the users and correspond to files in our system. TinyDB query syntax is similar to SQL SELECT-FROM-WHERE-GROUPBY clause, supporting selection, join, projection and aggregation. In addition TinyDB provides SAMPLE PERIOD clause defining the overall time of the sampling called epoch and the period between consecutive samples. The materialization points are created by CREATE STORAGE POINT clause, associated with a SELECT clause, which selects data either from the sensor table or from a different materialization point.

#### 4.2.2 Cougar

Cougar (Fung et al., 2002) is another data-centric middleware approach intended to address the goals of scalability and flexibility in monitoring the physical world. In Cougar system sensor nodes are organized in clusters and they can assume two roles: cluster leader or signal processing nodes. The leaders receive the queries and plan how they must be executed within of a cluster; in particular, they must decide what nodes the query should be sent to, and keep waiting for the response. On the other hand, signal processing nodes generate data from their sensor readings. Signal processing functions are modelled by using Abstract Data Type (ADT). Like TinyDB, Cougar uses a SQL-like language to implement queries.

#### 4.3 Network reprogramming applications

Code dissemination for network reprogramming is nowadays one of the important issues in the WSN field. WSN applications are conceived to execute for the maximum period of time. However, during their lifetime is very probable that the application needs to be total or partially updated. There are several reasons for it as to meet new requirements or to correct errors detected at execution time. There exist in the literature a large set of applications that enables this feature. Despite every particular implementation, a common characteristic of all of them is the employment of the flash memory to store the updates that are received from the network. In fact, there is no other choice due to the limited capacity of the node RAM memory. According to (Munawar et al., 2010) applications for remote reprogramming can be classified in four main categories:

- Full-image replacement: the first approach for network reprogramming operated disseminating in the network a new image to replace the current application running in the nodes. Examples of this type of reprogrammers are Deluge and XNP, which are both TinyOS 1.x specific. In a first step, the image was received from the network and locally stored in the node flash. Once the packet reception was completed, the sensor node reboots which makes a copy of the binary stored in the flash into the microcontroller. The main disadvantage of this approach is that even for small updates the transmission of the full image should be done, which impacts negatively on the waste of energy in the sensor node.
- Virtual machines: with the goal of reducing the energy consumption in which the previous approach incurs, there exist different works that propose disseminating virtual machine code (byte-code) instead of native code, since the first is in general more compact than the second one. The most relevant example is Mate (Levis & Culler, 2002). Maté disseminates to the network packets denominated capsules which contain the binary to be installed. In the sensor nodes the byte-code is interpreted and installed in the sensor node. The advantage of this approach is that reduces significantly the program size that travels through the network, which decreases the energy consumption due to the communication as well as the storing cost.
- Dynamic operating systems: there exists WSN operating systems that include support for the dynamic reprogramming of sensor nodes. For example, in Contiki applications can be more easily updated, due to the fact that Contiki supports load dynamic of programs on the top of the operating system kernel. In this way, code updates can be remotely downloaded into the network. There are, however, certain restrictions to perform this since only application components can be modified. LiteOS (Cao et al., 2008) is another example of this type of OSes. LiteOS provides dynamic reprogramming at the application level which means that the operating system image can not be updated. To do this it manages the modified HEX files instead using ELF files —as Contiki— in order to store relocation information.
- Partial-image replacement approach is based on disseminate only the changes between the current executable installed in the network and the new version of the same application. This is the most efficient solution since only is sent the piece of code that

needs to be updated. There are in the literature several works using this approach. Zephyr (Panta et al., 2009) compares the two binary images at the byte-level and send only a small delta, reducing the size of data to be sent. FlexCup (Marrón et al., 2006) is an efficient code update mechanism that allows the replacement of TinyOS binary components. FlexCup is specific for TinyOS 1.x and does not include the new extensions of nesC. Dynamic TinyOS (Munawar et al., 2010) preserves the modularity of TinyOS which is lost during the compilation process and enables the composition of the application at execution time.

## 5. Conclusions

Through this chapter we have analyzed the main features of the flash memory chip as well as their main applications within the wireless sensor networks field. We have described the different technologies employed in the manufacturing of flash memory given specific examples used by the sensor nodes. The sensor node architecture has been presented while the flash memory has been introduced as an important component that possibilities a great amount of usages which would not be possible without its presence.

We have described some relevant WSN operating systems highlighting the different abstractions that they provide at the application level in order to access the data stored in the flash. As discussed, in general the portability has been sacrificed and the implementation is typically device-specific. The abstraction level provided by the OSes is very low since the application must manage hardware level details such as the number of the page to be read or written and the offset within the page, which make complex the applications programming. To alleviate this problem, the operating systems can supply a basic implementation of a file system to facilitate the data access. Here, the users manipulate abstract entities called file descriptors which allow to uncouple the data from its physical location. Subsequently, file systems simplify the data access but in general they do not completely address the issues regarding to the flash memory such as the implementation of wear levelling techniques to prevent reaching the maximum number of times that a page can be written. For this reason, the literature presents some other file systems that has been proposed in order to improve the features or the performance of the existing files systems included into the operating systems.

Recently, the attention paid to the flash memory chip trends to grow due to the appearance of new applications that will use the flash memory to perform their tasks. Since the flash chip represents the device with the bigger capacity for permanent storage of application data in the sensor node, there exist an increasing number of applications that require its usage to be able to satisfy their requirements, for example, applications for dynamic reprogramming.

Finally in this chapter, we have identified a taxonomy of WSN applications that uses the flash memory providing specific examples of applications in each category of the taxonomy. We will envision that the number of emerging applications that will use the flash memory as basis for their operations will continue increasing.

## 6. Acknowledgements

This work has been partially funded by the Spanish Ministry of Science and Innovation under the grand TIN2010-16497.

## 7. References

Akyildiz, I. F., Su, W., Sankarasubramaniam, Y. & Cayirci, E. (2002). Wireless sensor networks: a survey., *Computer Networks* 38(4): 393–422.

Atmel (2011). Atmel 8-bit AVR microcontroller Datasheet, *Available in:* URL: *http://www.atmel.com/dyn/resources/prod_documents/doc2467.pdf* .

*Atmel AT45DB011 Serial DataFlash* (2001). URL: *http://www.datasheetcatalog.com/datasheets_pdf/A/T/4/5/AT45DB.shtml*.

Balani, R., chieh Han, C., Raghunathan, V. & Srivastava, M. (2005). Remote storage for sensor networks.

Cao, Q. & Abdelzaher, T. (2006). LiteOS: a lightweight operating system for C++ software development in sensor networks, *SenSys '06: Proceedings of the 4th international conference on Embedded networked sensor systems*, ACM, New York, NY, USA, pp. 361–362.

Cao, Q., Stankovic, J. A., Abdelzaher, T. F. & He, T. (2008). LiteOS, A Unix-like operating system and programming platform for wireless sensor networks, *Information Processing in Sensor Networks(IPSN/SPOTS)*, St. Loius, MO, USA.

*CC1000 Single Chip Very Low Power RF Transceiver* (2002). URL: *http://focus.ti.com/lit/ds/symlink/cc1000.pdf*.

*CC2400 2.4GHz Low-Power RF Transceiver* (2003). URL: *http://focus.ti.com/lit/ds/symlink/cc2400.pdf*.

Dai, H., Neufeld, M. & Han, R. (2004). Elf: an efficient log-structured flash file system for micro sensor nodes, *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*, ACM, New York, NY, USA, pp. 176–187.

Diao, Y., Ganesan, D., Mathur, G. & Shenoy, P. (2007). Rethinking data management for storage-centric sensor networks.

Dunkels, A., Gronvall, B. & Voigt, T. (2004). Contiki - a lightweight and flexible operating system for tiny networked sensors, *Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks*, LCN '04, IEEE Computer Society, Washington, DC, USA, pp. 455–462. URL: *http://dx.doi.org/10.1109/LCN.2004.38*

Escolar, S., Carretero, J., Isaila, F. & Lama, S. (2008). A lightweight storage system for sensor nodes, *in* H. R. Arabnia & Y. Mun (eds), *PDPTA*, CSREA Press, pp. 638–644.

Escolar, S., Isaila, F., Calderón, A., Sánchez, L. M. & Singh, D. E. (2010). Senfis: a sensor node file system for increasing the scalability and reliability of wireless sensor networks applications, *The Journal of Supercomputing* 51(1): 76–93.

Fung, W. F., Sun, D. & Gehrke, J. (2002). Cougar: the network is the database, *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, SIGMOD '02, ACM, New York, NY, USA, pp. 621–621. URL: *http://doi.acm.org/10.1145/564691.564775*

Gay, D. (2003). The Matchbox File System, URL: *http://webs.cs.berkeley.edu/tos/tinyos-1.x/doc/matchbox-design.pdf*.

Gay, D., Levis, P., von Behren, R., Welsh, M., Brewer, E. & Culler, D. (2003). The nesc language: A holistic approach to networked embedded systems, *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, ACM, New York, NY, USA, pp. 1–11.

Han, C.-C., Kumar, R., Shea, R., Kohler, E. & Srivastava, M. (2005). A dynamic operating system for sensor nodes, *Proceedings of the 3rd international conference on Mobile systems, applications, and services*, MobiSys '05, ACM, New York, NY, USA, pp. 163–176.
URL: *http://doi.acm.org/10.1145/1067170.1067188*

Handziski, V., Polastrey, J., Hauer, J.-H., Sharpy, C., Wolisz, A. & Culler, D. (2005). Flexible Hardware Abstraction for Wireless Sensor Networks, *2nd European Workshop on Wireless Sensor Networks (EWSN 2005)*, Istanbul, Turkey.

Hill, J., Szewczyk, R., Woo, A., Hollar, S., Culler, D. & Pister, K. (2000). System architecture directions for networked sensors, *SIGPLAN Not.* 35: 93–104.
URL: *http://doi.acm.org/10.1145/356989.356998*

Instrument, T. (2008). Msp430x1xx 8mhz datasheet, *Available in:* URL: *http://www.ti.com/lit/gpn/msp430c1101* .

*Intel Strataflash* (2002).
URL: *http://www-mtl.mit.edu/Courses/6.111/labkit/datasheets/28F128J3A.pdf.*

Jaikaeo, C., Srisathapornphat, C. & chung Shen, C. (2000). Querying and tasking in sensor networks.

Kawaguchi, A., Nishioka, S. & Motoda, H. (1995). A flash-memory based file system, *USENIX Winter*, pp. 155–164.
URL: *citeseer.ist.psu.edu/kawaguchi95flashmemory.html*

Levis, P. & Culler, D. (2002). Mate: a tiny virtual machine for sensor networks, *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, ACM, New York, NY, USA, pp. 85–95.

Li, S., Lin, Y., Son, S. H., Stankovic, J. A. & Wei, Y. (2003). Event detection services using data service middleware in distributed sensor networks.

*M25P40 Serial Flash Memory* (2002).
URL: *http://www.datasheetcatalog.org/datasheet/stmicroelectronics/7737.pdf* .

Madden, S. R., Franklin, M. J., Hellerstein, J. M. & Hong, W. (2005). TinyDB: an acquisitional query processing system for sensor networks, *ACM Trans. Database Syst.* 30(1): 122–173.

Marrón, P. J. (2005). Middleware approaches for sensor networks. University of Stuttgart, Summer School on WSNs and Smart Objects. Schloss Dagstuhl, Aug. Germany. URL: *http://www.vs.inf.ethz.ch/events/dag2005/program/lectures/marron-2.pdf.*

Marrón, P. J., Gauger, M., Lachenmann, A., Minder, D., Saukh, O. & Rothermel, K. (2006). Flexcup: A flexible and efficient code update mechanism for sensor networks.
URL: *http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.71.8622*

Munawar, W., Alizai, M. H., L, O. & Wehrle, K. (2010). Dynamic tinyos: Modular and transparent incremental code-updates for sensor networks.

*nRF2401 Radio Transceiver Data Sheet* (2003). URL: *http://www.nvlsi.no/.*

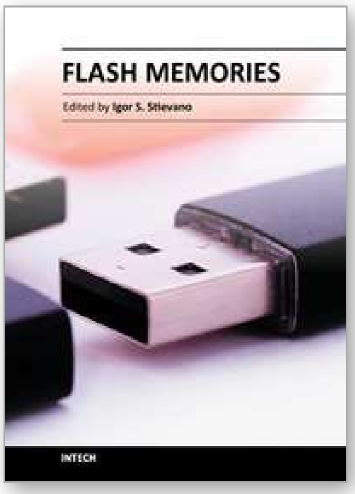
Panta, R. K., Bagchi, S. & Midkiff, S. P. (2009). Zephyr: efficient incremental reprogramming of sensor nodes using function call indirections and difference computation, *Proceedings of the 2009 conference on USENIX Annual technical conference*, USENIX'09, USENIX Association, Berkeley, CA, USA, pp. 32–32.
URL: *http://portal.acm.org/citation.cfm?id=1855807.1855839*

SAMSUNG (2003). Samsung K9K1G08R0B, 128M x 8 bit NAND Flash Memory.

Shenker, S., Ratnasamy, S., Karp, B., Govindan, R. & Estrin, D. (2003). Data-centric storage in sensornets, *SIGCOMM Comput. Commun. Rev.* 33(1): 137–142.

Tsiftes, N., Dunkels, A., He, Z. & Voigt, T. (2009). Enabling Large-Scale Storage in Sensor Networks with the Coffee File System, *Proceedings of the 8th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN 2009)*, San Francisco, USA.
URL: *http://www.sics.se/ adam/tsiftes09enabling.pdf*

Xu, N. (2002). A survey of sensor network applications, *IEEE Communications Magazine* 40.




**Flash Memories**
Edited by Prof. Igor Stievano

ISBN 978-953-307-272-2
Hard cover, 262 pages
**Publisher** InTech
**Published online** 06, September, 2011
**Published in print edition** September, 2011

Flash memories and memory systems are key resources for the development of electronic products implementing converging technologies or exploiting solid-state memory disks. This book illustrates state-of-the-art technologies and research studies on Flash memories. Topics in modeling, design, programming, and materials for memories are covered along with real application examples.



**InTech Europe**
University Campus STeP Ri
Slavka Krautzeka 83/A
51000 Rijeka, Croatia
Phone: +385 (51) 770 447
Fax: +385 (51) 686 166
www.intechopen.com

**InTech China**
Unit 405, Office Block, Hotel Equatorial Shanghai
No.65, Yan An Road (West), Shanghai, 200040, China
中国上海市延安西路65号上海国际贵都大饭店办公楼405单元
Phone: +86-21-62489820
Fax: +86-21-62489821