A portable implementation of the FAT16 filesystem to TinyOS-2.x: non-volatile mass storage for low power sensor network nodes

anonymous

Abstract

As part of a strategy aimed at reducing the overall power consumption of a node associated with data acquisition in a sensor network while improving the safety of data availability through its storage on non-volatile media, we demonstrate a portable implementation of the FAT filesystem for the TinyOS-2.x executive environment for storing large amount of data on Secure Digital non-volatile mass storage media. Thanks to this tool, a user can store an amount of data incompatible with a radiofrequency link (excessive power consumption and communication duration) using a format still compatible with most modern operating systems so as to be usable by non-technical users, yet compatible with the limited resources of low power embedded sensor nodes. Furthermore, data of different origins or representing various physical quantities are stored in different files.

Keywords

sensor node, TinyOS-2.x, file system, mass storage, portability

1 Introduction

In the context of the deployment of a large number of sensors monitoring the spatial variations of a physical quantity (sensor network [1, 2]), radiofrequency transmissions of the data is often the main source of power consumption and hence battery life expectancy reduction. The emphasis in the development of sensor nodes is commonly towards providing a self-reconfigurable wireless link for transferring data over a network [3]. Reduced power consumption induces limited bandwidth (typically of the order of a few tens of kilobits per second) and reduced communication range (a few meters to a few hundred meters). Some applications require larger amounts than a few tens of kilobytes to be stored [4] during each measurement session, while these data are not necessarily needed immediately. The application fields we are interested in are temporary storage of large amount of data (up to 1 MB/hour), with sensor nodes located too far from any power supply or network access to route data towards the user: in these conditions, the data are stored locally until a user manually fetches the stored information. One such practical context is the Ny Alesund area in Svalbard (79^oN, Norway) where, beyond the isolation of an arctic environment when the area under investigation is beyond high-frequency radio communication range (a few kilometers at best), the use of the 2.4 GHz ISM band is forbidden due to the interferences induced to a radio-telescope running in this area. However, severe climatic conditions for both the user and the hardware prompts the use of a mass storage format widely available on any computer the end-user might bring to quickly fetch data: the FAT filesystem (File Allocation Table based filesystem) is selected as a compromise between a complexity compatible with an implementation of low power consumption microcontrollers, while still available on most modern operating systems.

Indeed, our purpose in implementing a filesystem on TinyOS is to store data on a low-power embedded sensor node in a format allowing the user to recover these informations on a general purpose computer without requiring a dedicated function for reading the information.

The reasons for this project are

1. reducing the duration the sensor node is stopped during data recovery, since the memory card can be quickly removed and exchanged with a new one. Such a strategy is more robust than RS232 data transfer, an important aspect especially in hostile environments.
2. although the memory card might be used without filesystem in raw write mode or custom formats [4, 5], such a solution induces
   - a post-processing step before the data are available
   - some technical knowledge from the user in order to recover data,
   - a poor data organization when storing data of different quantities due to the lack of different files for different kinds of data.

The selection of the filesystem answers several needs:

1. due to the minimal resources available, all journaled filesystems are unsuitable due to the excessive read-/write access to the non-volatile mass-storage medium,

2. portability to any platform and operating system so that any user might be able to recover data.

One application example where we have used such a configuration in is storing GPS data for double differences post-processing: a Thales AC12 GPS receiver providing phase information generates about 1.5 MB per hour of measurement, and such measurements performed at various locations allows for a post-processing yielding sub-centimetric accuracy [6]. Here we propose the use of the well known MultiMediaCard (MMC) and Secure Digital (SD) non-volatile mass media storage support for saving these data on the sensor node until a human operator physically reaches the node for retrieving the data. However, raw storage of data on the non-volatile memory induces some technical retrieval procedure such as the use of the Disk Dump (`dd` utility) under unix systems, unavailable to MS-Windows users. In order to provide a wider audience of users with a format compatible with any operating system running on the data retrieval computer, we have selected a file storage format compatible with most modern operating systems (unix – including Apple's MacOS X – and MS-Windows [7]) yet running on the reduced resources available on sensor nodes. Our selection of a file format has thus been limited to those available when the resources of computers were about those of today's sensor nodes. Amongst the available selection including Minix, CP/M and FAT16, the latter is the only system still widely available on most modern operating systems yet developed about 30 years ago [7], when the memory available in personal computers was a few tens of kilobytes as found in sensor nodes.

Many autonomous FAT-based filesystems implementations [8] exist for various microcontrollers [9], and some ports comply with the TinyOS-1.x executive environment and have been adapted to TinyOS-2.x. However, we wish to provide a hardware independent, portable implementation of FAT for TinyOS-2.x with a minimal memory footprint mostly defined by the user application. Our objective is thus to demonstrate the implementation of a FAT filesystem support in an executive environment widely used for developing software running on sensor nodes, and more specifically on Crossbow's (San Jose, USA) TelosB [10] and MicaZ commercial sensor nodes: TinyOS-2.x [11]. We emphasize on the portability aspect by using the low level SPI bus access functions provided by TinyOS, and demonstrate that the use of an executive environment makes porting this tool from one platform to another painless, even across different processor and hardware architectures. We demonstrate the use of this software for the storage of several tens of megabytes of data retrieved from OEM GPS receivers, and the simultaneous storage of data of a different kind (temperature of the sensor node) in a different file, hence providing data organization structures compatible with the need of most users.

2 FAT for TinyOS-2.x

The nesC language on which TinyOS-2.x is based provides an abstraction layer which clearly separates the hardware interfaces, the needed resources and the methods associated with each use of these resources. Thanks to this abstraction, the port of the methods from one platform to another is mostly a matter of configuring the hardware resources needed. However, beyond the framework provided by TinyOS-2.x for developing portable applications, the developer must separate the hardware dependent platform from the SD-card and filesystem drivers. Such a rule seems not to be followed by the port of the shimmer [9] FAT-filesystem port from TinyOS-1.x to TinyOS-2.x: as an example of CPU-dependent code within the SD-access part of this code, `tos/platform/shimmer/chips/sd/SDP.nc` mostly reproduces `msp430/usart/Msp430SpiNoDmaP.nc` (`SpiByte.write` function). Hence, working on another platform requires copying most of the FAT filesystem access functions and porting the low level functionalities to the new CPU. Our implementation of the FAT filesystem and low level SPI access to the SD card focuses on portability by removing all CPU-specific code. As will be seen in the description of the implementation, the relationship between the SD card and the platform is done through the platform configuration file, which defines which clock source to use, what pin performs as chip select (CS), while the hardware independent functions `SpiByte` and `GeneralIO` have been used otherwise throughout the driver so that it can be used on any CPU supported by TinyOS-2.x.

2.1 Accessing the mass storage medium

The hardware interface we will be interested in for storing data is the MultiMediaCard (MMC) format, now often replaced by the compatible Secure Digital (SD) format. Both memory cards support a slower, 3-wire synchronous communication protocol: SPI [1]. This bus is either available as an hardware interface in most microcontrollers, or is easily emulated using general purpose input-output pins (GPIO). In our implementation, we assume that low level SPI initialization functions (`start()` and `stop()` methods), as well as writing (`write()` method) and the associated reading function over this bus (reading over SPI requires writing dummy data and recording the signals on the input (MISO) pin as the bus is clocked by the microcontroller), are provided by the executive environment through the `SplitControl` and `SpiByte` interfaces respectively. An additional Chip Select digital signal must be defined for the driver to activate or deactivate the SD card sharing the SPI bus with other peripherals (such as a radiofrequency transceiver, as seen on the MicaZ platform).

Having defined which hardware resources are needed on the platform – in all our discussion we shall need access to 3 data lines (Master In Slave Out MISO, Mater Out Slave In MOSI, Clock CK) and one Chip Select digital signal – the low level SD card access routines are used to initialize the communication mode (SPI) and data block size (512 bytes).

These low level communication functions (Fig. 1) provide the basic means for implementing the SD-card SPI-mode initialization functions as well as low level memory

[1]The SD card, used in SPI mode, is connected as follows to the MSP430F1611: Chip Select is P3.0 on pin 28, MOSI is P3.1 on pin 30, MISO is P3.2 on pin 30 and the clock on P3.3 on pin 31. For the MicaZ, we connect the SD to the 51-pin connector, with the Chip Select connected to LED1 on pin 10 (PA2), MISO to USART1_RXD on pin 19 (PD2), MOSI to USART1_TXD on pin 20 (PD3) and the Clock is generated by USART_CLK on pin 15 (PD5).

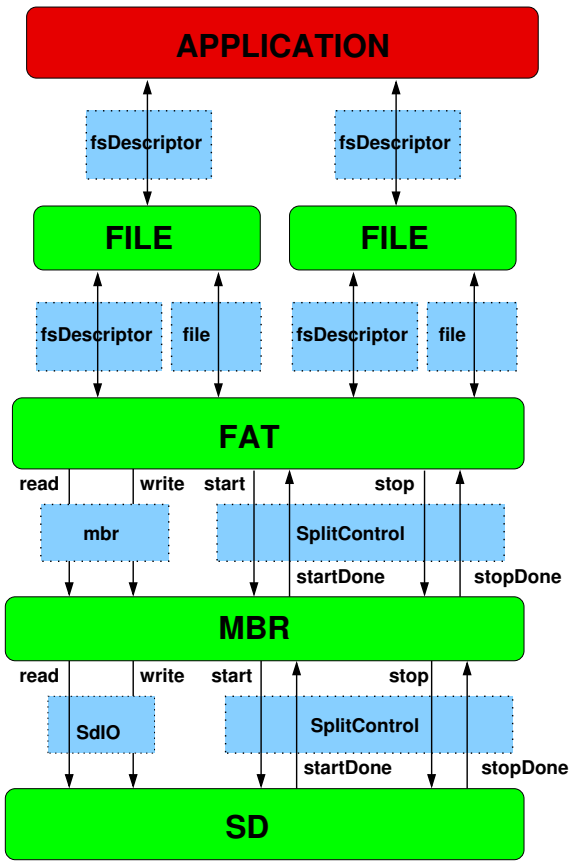

Figure 1. FAT16 drivers organization: the layer closest to the hardware uses the SPI-access functions provided by the executive environment to initialize the non-volatile mass-storage medium, and provides low level memory block writing and reading functions. Above this layer, the various abstraction layers associated with handling the Master Boot Record (MBR), defining the starting point of the File Allocation Table (FAT), which includes the first cluster address of each file stored in the SD-card, have been implemented as hardware-independent drivers. These layers are used by the user Application when writing data to a file.

block writing and reading. Indeed, all transactions with the SD card are performed with 512-byte chunks of data, defining the minimum volatile memory (RAM) needed by the driver. This block size defines the minimum memory requirement for exchanging data with the SD card: in raw write mode, the user *must* provide a 512-byte large array as a buffer of data to be written. At this stage of the development, raw-writing data on a non-formatted medium is possible, but does not answer our needs for a storage format widely available on most modern operating systems, hence our need to add the FAT format layer.

2.2 Formatted data storage

Once the communication in SPI mode is acknowledged by the memory card, the actual FAT filesystem layer is activated (Fig. 3). A file is either created by the application, or data are appended to existing files. The file size is updated after each writing step. Cache handling is not implemented by the driver because of the excessive memory requirement associated with caching: we have selected to implement a driver with minimal memory requirement, and leave the selection of caching data (with the risk of loosing the data which have not yet been stored on the non-volatile medium) to the user depending on his application and the data storage rate.

Since the basic block size for accessing an SD or MMC card is 512 bytes large, we need at any time at least two such buffers to hold the data manipulated for file storage, typically the data to be written next in the file and a copy of the FAT to be updated. Thanks to memory use optimizations, our current implementation of the FAT16 filesystem has been validated on the MSP430F149 (2 kB RAM), the MSP430F1611 (10 kB RAM) and ATMega128 (4 kB RAM). As opposed to the raw write mode in which the user has to provide a 512-byte buffer, in the formatted data storage the user provides a buffer the size of the data to be written, and the driver manages its own 512-byte large internal buffer to append the user data to the existing information in the file used.

3 Implementation

The filesystem implementation is split in four mostly independent modules (Fig. 1):

- access to the SD card through the TinyOS-2.x configuration file is independent of the Master Boot Record (MBR) decoding, allowing a RawWrite (no organization of the data compatible with multiple files or filesystem) access to the non-volatile mass storage medium
- access to the MBR is independent from the SD and FAT filesystem layers, and can be hence used for other filesystems than FAT
- files are accessed atop the FAT filesystem layer

The separation is actually more defined between functional blocks:

- storage from a hardware point of view (accessing the SD card). This part provides the low level raw-write and read functions through a portable SPI interface to the SD card.
- storage in terms of partitioning the medium
- storage in terms of formatting and hence the filesystem, compatible with an access from personal computers running most common operating systems.

The SD card is configured through the platform definition file which must be updated for each new board or CPU architecture. This configuration part includes assigning hardware or software USART pins to the communication with the SPI-compatible pins of the SD card, and in case of hardware USART configuring the data transmission parameters (bits/sentence, communication speed, phase ...).

The file descriptor receives as parameters the name of the file to be opened, so that multiple files can be opened at a same time in order to store data in various files depending on the various sources of these data.

The steps for writing data on the filesystem follow these steps:

- check the available space is the current cluster and sector, and if necessary allocate a new cluster and update of both FATs
- write the data with, if necessary, a concatenation step with the older data already written in the sector being accessed
- once the data have been written in the filesystem, the file size is updated.

In order to reduce the amount of memory used and chances of data loss, any request to write data on the non-volatile (SD) mass storage medium is immediately executed (no data cache is performed).

Since each writing step is independent of the previous one and performed only if the previous one succeeded, the risks of data loss are only associated with the following cases:

- power supply loss between two writing sequences, yielding the loss of the last block which was assumed to be written on the non-volatile storage medium (at most 512 bytes)
- power supply loss while writing data: the data loss is the same as mentioned in the previous case
- power supply loss while updating the FAT and its backup copy. Since two copies of the FAT are defined at any time, data retrieval is always possible.

In the worst case, the maximum amount of data lost is 512 bytes.

Here we will describe in details the implementation of the communication protocol with the SD card, but more significantly the FAT filesystem and its use.

3.1 Synchronous bus for communicating with the SD card

A synchronous bus shares a common clock signal between the master (processor) and the slave (SD card), providing improved communication bandwidth with respect to asynchronous busses. SPI is one implementation of a synchronous protocol, provided as a hardware peripheral on most modern microcontrollers, and especially on all platforms supported by TinyOS-2.x. Such a bus requires three signals – Master to Slave and Slave to Master communication, bus Clock – and an additional Chip Select signal for enabling a given peripheral connected to the bus. The communication bandwidth of this bus as implemented in low-power microcontrollers is in the Mb/s range. In the case of the MSP430 series, the hardware peripheral is shared with the asynchronous communication peripheral (USART).

The hardware implementation of the synchronous bus on MSP430 processors only provides a single byte buffer [12, chap.14]. However, as opposed to the asynchronous bus communication, all transactions on the synchronous are triggered by the Master and hence no data loss can occur due to significant data processing delays on slower microcontrollers.

SPI bus communication is a standard part of TinyOS and no dedicated development is needed: we will focus on using the standard functions for portability compliance: the only requirement is the definitions of the configuration parameters (clock source and prescaler, phase sign ...) and activating the peripheral. In the case of the MSP430 based platforms, this hardware dependent configuration step is taken care of by the `Msp430SpiConfigure` interface.

Since no portable SD-card support is available for TinyOS-2.x – only TinyOS-1.x ports are available, with hardly any compliance with the newer hierarchy and portability model, we decided to write our own implementation of the communication protocol in full compliance with the portability rules defined by TinyOS-2.x, and most significantly using the low-level access function provided by the executive environment platform implementation. In order to reach this target, we will use the SPI-compatible mode of the SD-card communication protocol (as opposed to the hardly documented native protocol).

SD card communication requires the compliance with the following requirements:

- reading and writing is performed on data blocks 1 to 512 bytes large, with a default value of 512 which will be user throughout our implementation,
- when accessing data blocks 512 bytes large, the addresses must be multiples of this value,
- an arbitrary number of blocks can be erased

3.1.1 SPI communication

The first layer (listing 1), closest to the hardware, is the low level SPI driver based on a the MSP430 functionalities provided by TinyOS-2.x (`Msp430Spi0C`) and its *SpiByte* interface providing a single function command uint8_t write(→ ↪ uint8_t tx);. This unique function both sends a byte on the bus and returns the value read during the same clock cycles.

```
#include "hardware.h"
configuration PlatformSdC {
  provides {
    interface SplitControl;
    interface SpiByte;
  }
}
implementation {
  components projetSdP;
  SplitControl = projetSdP.Control;

  components new Msp430Spi0C() as SpiC;
  projetSdP.Msp430SpiConfigure <- SpiC.→
      ↪Msp430SpiConfigure;
  projetSdP.Resource -> SpiC.Resource;
  SpiByte = SpiC;
}
```

Listing 1. PlatformSdC

3.1.2 Interface usage

The second step aims at defining the interaction between the SD driver and the higher abstraction layers. Beyond the rawrite functionality in which the user must provide a 512-byte data array, the driver handling more complex functions will have to allocate one local 512-byte buffer for managing temporary data (storing informations from partially filled data blocks for example).

Since such a memory allocation significantly impacts the memory usage of the application, a blocking function model was selected with the function returning only once all op-

erations associated with storing on the SD card have been completed. The user application provides a buffer including the data to be written, which is not copied at the driver level: cache handling is managed at the user application level and not at the driver level in order to minimize the memory usage impact at 512 bytes.

The interface is as follows:

```
interface SdIO {
  /**
   * Command for writing a data block
   * Blocking function
   *
   * @param addr: adress at which data are→
       ↪ written
   * @param buf: data array
   *
   * @return SUCCESS if the command has →
       ↪completed
   */
  command error_t write(uint32_t addr, →
      ↪uint8_t*buf);

  /**
   * Data block read request
   * Blocking function
   *
   * @param addr: adress at which data are→
       ↪ read
   * @param buf: data array
   * @param count: array length
   *
   * @return SUCCESS if the command has →
       ↪completed
   */
  command error_t read(uint32_t addr, →
      ↪uint8_t*buf, uint16_t *count);
}
```

Listing 2. SD module usage interface

Both functions use as argument the address (in bytes) at which data are accessed, a buffer in which data are stored (either for writing or reading), and in case of a read request, the data length to be read.

3.1.3 Bandwidth measurement

A dedicated application was developed for measuring the data transfer rate in the raw write mode: a 512-byte buffer is written 2048 times. We have observed that 113 seconds are needed to write 1 MB, resulting in a data transfer rate of 9 KB/s or 32 MB/hour.

3.2 Storage medium and partition

A physical storage medium appears as a large array segmented in smaller blocks – sectors – each 512-bytes large in the case of the SD card.

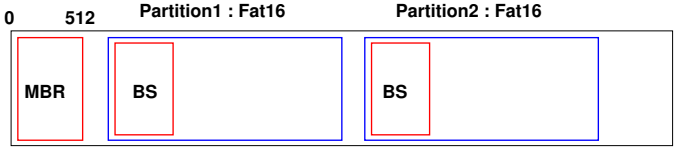

Figure 2. Partitioning example of a mass storage medium

Partitioning (Fig. 2) a large storage medium creates at the beginning of the array (address 0) a data structure called the Master Boot Record (MBR), used to

- provide informations concerning the mass storage medium,
- store, if needed, an executable file for booting from this mass storage medium,
- provide informations such as the size and address of the partitions.

This last part is most significant for us since these informations are needed to access the partition. The MBR includes four 16-bytes fields which each defines the properties of a partition: the only part of interest to us is the address of the beginning of the partition, at offset 0x1C6 in the case of the first partition.

3.3 Structure

Before getting into the details of the filesystem itself, we must define what a cluster is. A cluster is the basic block unit as seen from the filesystem, and defined as multiple sectors. The number of sectors in each cluster is defined in the area defining the partition.

The FAT16 filesystem is structured as illustrated in Fig. 3.

3.3.1 Boot Sector

At the beginning of a partition is the Boot Sector (BS). It performs at the partition level a role similar to that of the MBR at the physical storage medium level. It includes all the informations needed to use the partition.

Amongst other data, the BS provides the sizes of:

- the partition,
- various areas within the partition,
- a cluster,
- ...

The only part of fixed size is the BS as defined in the standards, while accessing all the other areas of the partition is based on the values read there.

3.3.2 FAT (File Allocation Table)

From now on, to avoid confusion, the filesystem will be named *FAT16* while the linked lists will be called *FAT*.

The filesystem includes two *FATs*. The second is used as a backup copy and should be, if all performs well, a copy of the first one.

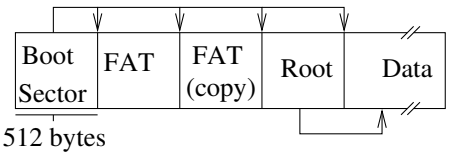

Figure 3. FAT filesystem organization on the mass storage medium.

The FAT filesystem requires the following steps to access data ordered in different files (Fig. 3):

- the Boot Sector, always located at the beginning of a partition and of fixed size, provides the address to the

FAT and its backup copy, as well as to a Root Directory. The medium can be divided into multiple partitions, which are defined by their starting address provided in the Master Boot Record (MBR). The MBR is always located at the beginning of the medium.

- the Root Directory provides the starting point of each file created on the filesystem. It includes as many entries as there are files or directories.
- the FAT provides a linked list of addresses where to find the successive data included in each file
- this linked list provides the addresses in the Data area where to retrieve the informations.

This structure is a linked list of cluster. The content of a file being stored as blocks of several non-contiguous sectors, the relation from one to the other is provided by the *FAT*. The cluster defining the beginning of a file is indicated as an entry in the RootDirSector area.

The first 16 bits of a FAT provides an information concerning the kind of medium, the next two are the status of the partition. The first usable cluster in this area is indexed by number 2, the last is dependent on the partition size. Each cluster index is coded as 16-bit values, the index being of three types (Fig. 4):

- 0x0000 to mention that a cluster is not used and hence can be reserved,
- 0xFFFF to mention that the cluster is used and additionally defines the end of a file,
- a value between 0x0000 and 0xFFFF is the index of the next cluster in the list.

Fig.4 exhibits a simple example, in which two files are defined in the FAT:

- the first file starts at the second cluster (information obtained at the beginning of the file),
- continues at cluster 3 (value found in cluster 2),
- finishes at cluster 9 (since it starts with value 0xffff).

The same analysis shows that the second file starts at the 6th cluster.

3.3.3 Root directory

The root directory of a partition (*RootDirSector*) defines all the files and directories available from the top-most directory, with each file entry defining all the data needed to access the informations stored there. The most important informations are the location of the first cluster, and the file size.

For each file entry, the first byte exhibits a particular value amongst:

- **0x00**, the sequence is complete, there is no more file
- **0xE5**, the file has been deleted
- **0x4n**, for the beginning of a long name, with n the number of lines needed to store the long name ($n \in [1:9]$),
- in all other cases, this character is the first letter of the file name.

Finding a file requires reading all entries in this area and analyzing the properties of each field one after the other. If only the short (DOS-like) name is of interest, the search is performed as follows: in the beginning, we might expect either a deleted file, or a long filename.

- in the former case, we jump 32 bytes to analyze the next entry,
- in the latter case one must jump $n * 32$ bytes to access the short DOS (8.3 format) filename.

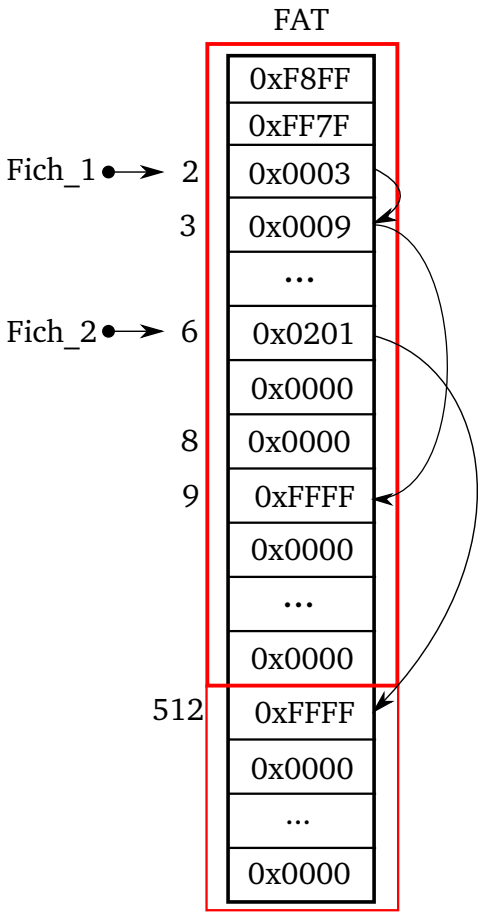

Figure 4. FAT example

3.3.4 Data area

The rest of the partition includes the data organized as cluster blocks, each divided in sectors. Accessing the data requires the use of the *FAT* for moving along the linked list, as mentioned earlier.

3.4 Power consumption issue: suspend mode

Accessing data in a FAT structure requires following linked lists. Hence, when opening a file or when accessing a new partition, one must find the last used cluster of a file or the first unused cluster free for writing new data. Jumping from one cluster to another requires many read access to the physical medium, even reading all entries when searching for the first empty cluster. For an application waking up periodically for storing environmental data, this overhead is hardly acceptable since it greatly increases the initialization duration, and thus the energy consumption associated with data storage. One possible solution is to store some of the informations concerning the filesystem in memory to

avoid re-initiating the whole filesystem analysis during each wakeup step, under the assumption that the microcontroler is the sole source of data on this filesystem and provided that a reinitalization sequence is performed when a new card is inserted during data retrieval. Thus, a sleep state associated with the file descriptor and the partition is provided for storing in memory (RAM) the cluster allocation table. As opposed to shutting down the filesystem during each data storage completion, this strategy reduces the initialization overhead when used with application that periodically wake up. When data coherence is the priority, the most reliable solution is shutting down the filesystem once the data have been stored on the non-volatile medium, and only relying on the FAT filesystem for retrieving cluster allocation informations.

The second issues is concerned with writing the acquired data, since the SD card requires data to be written as 512-bytes blocks and hence appending new data to partially filled clusters. The issue arises when the amount of data is smaller than the block size, a case often met when monitoring scalar quantities (temperature, rain level, wind speed ...): appending new data requires knowing how filled the last block already is. Rather than reading the content of the last block and analyzing the resulting data to identify the position of the last data, we store a variable at the file descriptor level which indicates how filled the last block is. This variable is initialized at a default value of bytesUsed = fileLength & 0→ ↪x1FF;, and provides the following information:

- the last block can be directly written if the variable is equal to 0 (empty block)
- go to the next cluster if the variable is equal to 512 (last block is filled)
- read the content of the last block and append the new data to the existing ones using the `memcpy` function.

Having developed the issues and solutions we have adopted, we will describe the actual implementation of these concepts under TinyOS-2.x.

3.5 Implementation

3.5.1 MBR

Managing the MBR is limited to

- transmitting the startup or shutdown commands to the storage medium
- finding the starting position of the partition during the initialization step

```
[...]
if (call SdIO.read(0,buf,512) == →
    ↪SUCCESS){
  debPartition = (*(uint32_t *)&buf[→
      ↪pos+8])<<9;
  mbrState = MBR_IDLE;
  error = SUCCESS;
}
[...]
```

- adding this offset during the data writing or reading steps:

```
command error_t mbr.read(uint32_t →
    ↪offset, uint8_t *buffer) {
  [...]
  error = call SdIO.read(debPartition→
      ↪+offset,buffer,512);
  [...]
}
command error_t mbr.write(uint32_t →
    ↪offset, uint8_t *buffer) {
  [...]
  error = call SdIO.write(→
      ↪debPartition+offset,buffer→
      ↪,512);
  [...]
}
```

3.5.2 FAT16

Above the MBR layer, the FAT module performs the most significant processing steps: when initializing the storage, it calls the MBR driver and analyzes the BS in order to retrieve all the needed informations:

```
[...]
if (call mbr.read(0,buf) == SUCCESS) {
  BytsPerSec = (buf[12]<<8)+buf[11];
  SecPerClus=buf[13];
  RsvdSecCnt = (buf[15]<<8)+buf[14];
  NumFATs=buf[16];
  RootEntCnt = (buf[18]<<8)+buf[17];
  FATSz = (buf[23]<<8)+buf[22];
  FirstFatByts = RsvdSecCnt*BytsPerSec;
  // root directory
  RootDirByts = (RsvdSecCnt + NumFATs*→
      ↪FATSz)*BytsPerSec;
  // first data sector position
  FirstDataSector = RootDirByts +
    fatAlignSup(RootEntCnt,BytsPerSec)*→
        ↪BytsPerSec;
  CountOfClusters = FATSz*((BytsPerSec/2)→
      ↪-2);
  if (fatState != FAT_SUSPEND)
    lastFreeCluster = 3; // start at →
        ↪beginning
[...]
```

It performs cluster allocation by first finding the offset and sector position of the last used area (*i.e.* position of the first available cluster). The offset provides the position with respect to the beginning of the sector being used and defines whether all cluster entries, within this sector, have been analyzed.

```
[...]
getOffsetAndSector(lastFreeCluster,&fatSec→
    ↪,&off);
if (off < 512) {
  currentFatSec = (512*(fatSec-1))+→
      ↪FirstFatByts;
  if (call mbr.read(currentFatSec, buf) ==→
      ↪ FAIL)
    goto end;
}
```

Next, the sector is analyzed until a free cluster is found, or the search follows on the next sector:

```
do {
  if (off >= 512) {
    fatSec++; // next sector
    currentFatSec = (512*(fatSec-1))+→
        ↪FirstFatByts;
    off = 0;  // offset reset
```

```
    // read a new data sector
    if (call mbr.read(currentFatSec, buf) →
        ↪== FAIL)
      goto end;
  }
  // sector found
  if ( (*(uint16_t*)&buf[off]) == 0x00){
    error = SUCCESS;
    break;
  }
  lastFreeCluster++; // goto next cluster
  off+=2; // next position to read
} while(lastFreeCluster < CountOfClusters)→
    ↪;
[...]
```

This loop iterates as long as no free cluster has been found and as long as the whole FAT has not been analyzed. Once a free cluster is identified, it is tagged as used and end of the file:

```
(*(uint16_t *)&buf[off])=0xffff;
[...]
```

The two FATs are updated so that the previous final cluster indexes the newly allocated cluster, while the new cluster is tagged as the final file position. During this update, two conditions are possible:

- both clusters are located in the same sector of the FAT, and in this case one writing step to the FAT is needed:

```
if (*cluster != 0)
  (*(uint16_t *)&(buf[offOr]))=→
      ↪lastFreeCluster;
if (call mbr.write(currentFatSec, buf→
    ↪) == FAIL ||
  call mbr.write(currentFatSec+(FATSz→
      ↪*512),buf)==FAIL)
  goto end;
}
[...]
```

- both clusters are in different sectors: in this case, two successive writing steps are needed during the update

```
if (fatSec != secOr) {
  if (call mbr.write(currentFatSec, →
      ↪buf)== FAIL ||
    call mbr.write(currentFatSec+(→
        ↪FATSz*512),buf)== FAIL)
    goto end;
  if (*cluster != 0) {
    tmp = (512*(secOr-1))+→
        ↪FirstFatByts;
    if (call mbr.read(tmp, buf)== →
        ↪FAIL) goto end;
    (*(uint16_t *)&(buf[offOr]))=→
        ↪lastFreeCluster;
    if (call mbr.write(tmp, buf) == →
        ↪FAIL ||
      call mbr.write(tmp+(FATSz*512),→
          ↪buf)==FAIL)
      goto end;
  }
```

Once the FATs are updated, the variables associated with the file descriptor are updated:

```
*cluster = lastFreeCluster;
lastFreeCluster++; // next since the →
    ↪current is used
```

The final step is performed while writing the data: if the sector to be used is beyond the number of sectors within the current cluster, a new cluster reservation is performed:

```
[...]
if (*secteur >= SecPerClus || *cluster == →
    ↪0) {
  //update cluster value
  if (fatReserveCluster(buf, cluster) == →
      ↪FAIL)
    goto end;
  *bytesUsed = 0; // new cluster
  *secteur = 0;
}
```

and the absolute position (with respect to the beginning of the SD card) of the current sector (in bytes) is computed:

```
sector = computeSecFromCluster(*cluster)→
    ↪+((*secteur)*512);
memset(buf,'\0',512);
```

If the sector under investigation is not empty, its content is recovered so the new data can be appended:

```
/* read only of data already there */
if (*bytesUsed != 0) {
  if (call mbr.read(sector, buf) == FAIL)
    goto end;
}
```

The amount of data already stored is checked in order to assess whether all the additional data will fit within the sector:

```
if (*bytesUsed + s > 512){
  (*secteur)++; // next sector
  size = 512 - *bytesUsed;
} else {
  if (*bytesUsed + s == 512) (*secteur)++;
  size = s;
}
```

Finally, the sector is filled, the sector number updated and the amount of bytes stored:

```
memcpy(buf+*bytesUsed,c,size);
if (call mbr.write(sector,buf) == FAIL)
  goto end;
s-=size;
*bytesUsed = (*bytesUsed+size) & 0x1FF;
c += size;
[...]
```

These steps are iterated until all the informations to be written are stored on the card.

3.5.3 file

The `file` driver performs similarly to the file descriptor in most modern operating systems: it manages all the methods associated to a file.

During the initialization step, the entry for a file is searched:

```
[...]
// first root sector
if (call fat.fatReadRoot(0,buf) == FAIL)
  return ret;
do {
  c = buf+offset;
  if (c[0] == 0x00) { // end of file
    break;
  // long file name: skip
  } else if ((uint8_t)c[0] == 229 || c[11]→
      ↪ == 15) {
```

```
  } else if (c[11] == 0x20) { // file →
      ↪found
    strncpy(fileName,c,7);
    fileName[8]='\0';
    if (!strncmp(fileName,name,strlen(name→
        ↪))) {
      ret = SUCCESS;
      break;
    }
  }
  // next entry
  offset += 32;
  // if sector overflow, goto next
  if (offset >= 512) {
    sector++;
    offset -= 512;
    call fat.fatReadRoot(sector,buf);
  }
}while(c[0] != 0x00); // return data
*off = offset;
*sect = sector;
[...]
```

All variables associated with a file are filled: in case of an empty file (cluster number equals 0), a new allocation request is performed. If the file descriptor is in sleep mode (*suspend()* command), the search for the cluster describing the end of file is not needed since a variable already contains this informations:

```
/* record file name */
c = (buf+offset);
// first cluster position
dataLocation = c[26];
// file length
fileLength = (*(uint32_t *)&(c+sec)[28]);
// position within root directory sector
rootSector = sec;
rootOffset = offset;
// Cluster of end of file
// if file is empty
if (dataLocation < 2) {
  // cluster reservation
  if (call fat.reserveCluster(&→
      ↪dataLocation,buf) == FAIL){
    return FAIL;
  }

  lastCluster = dataLocation;
  lastSecteur = 0;
  bytesUsed = 0;
  call fat.fatReadRoot(rootSector,buf);
  (*(uint16_t *)&(buf+rootOffset)[26]) = →
      ↪dataLocation;
  call fat.fatWriteRoot(rootSector,buf);
} else {
  if (fileState == FILE_NOINIT) {
    if ((lastCluster = call fat.→
        ↪fatLastCluster(dataLocation,buf))
                == -1)
      return FAIL;
    if ((lastSecteur = call fat.→
        ↪fatLastSecteur(lastCluster,buf))
                == -1)
      return FAIL;
  }
  bytesUsed = fileLength & 0x1FF;
}
```

During a write request, the buffer including the data is provided to the filesystem, as well as the number of bytes to be written. Once the data have been written, the file size is updated. This behavior, although requiring an additional writing step, provides additional safety concerning the data storage even under power failure during the writing step, since the previous data stored can always be recovered thanks to the filesystem consistency.

4 Usage example

As a basic example of *FAT16* use, the first program initializes the filesystem, opens a file in which a string is written, and the file is closed before shutting down the filesystem. The second example illustrates the use of the suspend and wakeup modes for storing periodic temperature readings.

4.1 Using FAT16

The various modules are connected together and the file name is defined in the configuration file (listing 3):

```
configuration fatTestAppC {}
implementation {
  components fatTestC as App, MainC, fatC;
  components new TimerMilliC() as Timer0;
  components new fileC("toto.txt");
  App.Boot -> MainC.Boot;
  App.Timer0 -> Timer0;
  App.fatControl -> fatC;
  App.file -> fileC;
  App.fileControl -> fileC.fileControl;
  fileC.fat -> fatC.fat;
}
```

Listing 3. Application configuration file

The actual application (listing 4)

- mounts the partition when a timer condition is met
- upon partition initialization success, the file is opened
- once the file is opened, a string is appended
- the file is closed and the partition unmounted.

```
module fatTestC {
  uses {
    interface Boot;
    interface Timer<TMilli> as Timer0;
    interface SplitControl as fatControl;
    interface SplitControl as fileControl;
    interface fat;
    interface file;
    }
}
implementation {
  event void Boot.booted()  {
    call Timer0.startOneShot(500);
  }
  event void Timer0.fired() {
    call fatControl.start();
  }
  event void fatControl.startDone(error_t →
      ↪err){
    if (err == SUCCESS) call fileControl.→
        ↪start();
  }
```

```
  event void fileControl.startDone(error_t→
      ↪ err) {
    if (err == SUCCESS)
      call file.write("hello world",11)
  }
  event void file.writeDone(error_t err) {
    if (err == SUCCESS) call fileControl.→
        ↪stop();
  }
  event void fileControl.stopDone(error_t →
      ↪err) {
      if (err == SUCCESS) call fatControl.→
          ↪stop();
    }
  event void fatControl.stopDone(error_t →
      ↪err){}
  [...]
}
```

Listing 4. Application file.

4.2 Temperature storage and suspend mode

The application configuration file (Fig. 5) is similar to the previous one, with the additional use of *DemoSensorC* for temperature monitoring.

```
configuration storeTempAppC {}
implementation {
  components storeTempC as App, LedsC, →
      ↪MainC;
  App.Boot -> MainC.Boot;
  App.Leds -> LedsC;
  components new TimerMilliC() as Timer0;
  App.Timer0 -> Timer0;
  components fatC;
  App.fatDescriptor -> fatC;
  components new fileC("temp.txt") as →
      ↪fileADC;
  App.fileADC -> fileADC;
  App.fileADCDescriptor -> fileADC.→
      ↪fileDescriptor;
  fileADC.fat -> fatC.fat;
  components new DemoSensorC() as Sensor;
  App.readADC -> Sensor;
}
```

Figure 5. Periodic temperature acquisition configuration file.

The actual application adds the use of the ADC12 peripheral:

```
#include "Timer.h"
#define TIMER_SLEEP 3600000
#define SHOW_ERROR do { \
  call Leds.led1Off(); \
  call Leds.led0On(); } while(0)

module storeTempC {
  uses {
    interface Leds;
      interface Boot;
      interface Timer<TMilli> as Timer0;
    interface Read<uint16_t> as readADC;
      interface fat;
    interface fsDescriptor as →
        ↪fatDescriptor;
    interface file as fileADC;
    interface fsDescriptor as →
        ↪fileADCDescriptor;
  }
}
```

and the *implementation* part is divided in four functional blocks:

```
implementation {
  enum {
    APP_NOINIT,
    APP_STOP,
    APP_SLEEP,
    APP_ADC
  };
  uint8_t appState = APP_NOINIT;
  uint8_t *tampon=NULL;

  event void Boot.booted() {
    tampon    = (uint8_t *) malloc(8*→
        ↪sizeof(uint8_t *));
    appState  = APP_NOINIT;
    call Timer0.startPeriodic(TIMER_SLEEP)→
        ↪;
  }
  event void Timer0.fired() {
    error_t error = FAIL;
    call Leds.led1Off();
    call Leds.led0Off();
    if (appState == APP_NOINIT || appState→
        ↪ == APP_STOP)
      error = call fatDescriptor.open();
    else if (appState == APP_SLEEP)
      error = call fatDescriptor.resume();
    if (error == FAIL)
      SHOW_ERROR;
  }
```

with at first the definition of the variables needed to store the status of the application over time and the array needed to collect the data to be written on the card. The filesystem states are defined lines 2 to 7 and a variable stores this status so that when the timer triggers an alarm, the filesystem is initialized (*call fatDescriptor.open()*) or awaken (*call fatDescriptor.resume()*) .

The other parts of the application are concerned with timer handling for periodic data acquisition:

```
  event void fatDescriptor.openDone(→
      ↪error_t error){
    if (error == SUCCESS) error = call →
        ↪fileADCDescriptor.open();
    if (error == FAIL)    SHOW_ERROR;
  }
  event void fileADCDescriptor.openDone(→
      ↪error_t error) {
    if (error == SUCCESS){
      atomic { appState = APP_ADC;}
      error = call readADC.read();
    }
    if (error == FAIL) {
      call fatDescriptor.close();
      SHOW_ERROR;
    }
  }
  event void fatDescriptor.resumeDone(→
      ↪error_t error){
```

```
    if (error == SUCCESS) error = call →
        ↪fileADCDescriptor.resume();
    if (error == FAIL)    SHOW_ERROR;
  }
  event void fileADCDescriptor.resumeDone(→
      ↪error_t error) {
    if (error == SUCCESS){
      atomic { appState = APP_ADC;}
      error = call readADC.read();
    }
    if (error == FAIL) {
      call fileADCDescriptor.close();
      SHOW_ERROR;
    }
  }
```

The second block is concerned with initializing or waking up the filesystem: a successful filesystem operation induces the same task on the file itself. Upon completion, data acquisition is performed:

```
  event void fileADCDescriptor.closeDone(→
      ↪error_t error) {
    if (error == SUCCESS) error = call →
        ↪fatDescriptor.close();
    if (error==FAIL)    SHOW_ERROR;
  }
  event void fatDescriptor.closeDone(→
      ↪error_t error){
    if (error == FAIL)  SHOW_ERROR;
    else          atomic {appState = →
        ↪APP_STOP;}
  }
  event void fileADCDescriptor.suspendDone→
      ↪(error_t error) {
    if (error == SUCCESS) error = call →
        ↪fatDescriptor.suspend();
    if (error == FAIL)    SHOW_ERROR;
  }
  event void fatDescriptor.suspendDone(→
      ↪error_t error){
    call Leds.led1Off();
    if (error == FAIL)    SHOW_ERROR;
    else            atomic { appState = →
        ↪APP_SLEEP;}
  }
}
```

The third block, similar to the previous, handles shutdown or suspending of the file and filesystem:

```
  event void readADC.readDone(error_t →
      ↪result, uint16_t data) {
    uint16_t inter, i, z;
    float val = (((data/4096.0)*1.5)→
        ↪-0.986)/0.00355;
    memset(tampon,'\0',8*sizeof(uint8_t));
    for(i=0,z=100;i<3;i++,z/=10){
      inter = val / z;
      tampon[i] = inter+'0';
      val -= inter*z;
    }
    tampon[3] = '\n';
    appState = APP_SLEEP;
    if (call fileADC.write(tampon,4) == →
        ↪FAIL)
      SHOW_ERROR;
  }
  event void fileADC.writeDone(error_t →
      ↪error) {
    if (error == SUCCESS)
      error = call fileADCDescriptor.→
          ↪suspend();
    if (error == FAIL){
      call fileADCDescriptor.close();
      SHOW_ERROR;
    }
  }
```

Finally, the last part is concerned with data processing and handling. Although the shortest part in this particular example, it is the core aspect of the program. *readADC.readDone(...)* is executed when the microcontroler completes the temperature measurement. Following a conversion to a string format, a write request is performed: upon completion, the file system is suspended, until the next alarm due to the periodic timer.

5 Speed and portability assessment

5.1 Bandwidth

In order to assess the writing speed, a dedicated application similar to the one previously described for raw writing on the medium has been developed. A test duration of 341 s provides a bandwidth estimate of 3.2 kB/s, or 11.5 MB/hour. This result, three times slower than the raw write bandwidth, is consistent with the overhead associated with formatted data storage.

5.2 Portability and practical use assessment

One application for the storage of large amount of data on non-volatile media is for the periodic measurement of GPS position. Whether short period (one measurement every second) or long periodic measurements (several minutes every day), the amount of data generated this way is in the megabyte range every day. The typical data transfer rate of GPS receivers is 4800 bauds on an asynchronous, RS232 compatible link. Hence, our assessment of the speed at which the FAT filesystem stores data has focused on the integrity of the data stream continuously transfered at the rate of 4800 bauds. Experiments were performed on a custom MSP430F1611-based board clocked with both 32 kHz LFXT and a 4 MHz high frequency clocks. We have also validated that this software runs on the standard (MSP430F611 based) TelosB platform and (ATmega 128 based) MicaZ OEM boards (Fig. 7). Working on these two platforms, based on different processor architectures, demonstrates the portability of the software used for running TinyOS-2.x. In the former cases (MSP430) we have used the hardware USART SPI port, while on the MicaZ we have connected the SD card to the USART1 port also used for communicating with the on-board flash memory [13]. The GPIO LED1 pin of the MicaZ was used as Chip Select pin.

Our experimental assessment of data integrity when stored on our implementation of the FAT filesystem includes sending through an RS232 link a know file, and searching differences between the original and stored files, and on the other hand storing GPS NMEA sentences and post-processing these data in search of inconsistencies (notice that each NMEA sentence is terminated with a checksum). In both cases, and with data files up to 11 MB large, no

inconsistency was detected when data transfer rates up to 9600 bauds were used.

We have performed current consumption measurements on a custom board including a direct (3.3 V) power supply to all the circuits, an MSP430F1611 processor, a 4066 analog multiplexer, an unpowered FT232RL USB to RS232 converter (although unpowered, we have observed a leakage current from the MSP430 UART to the FT232RL) and an LTC1157 FET transistor driver for switching the power supply to peripherals such as a GPS receiver or remote controlled instruments. The baseline is measured at 8 mA when no access to the SD card is needed. Waking up the SD card and writing to its cache RAM rises the power consumption to 38 mA (125 mW). Brief bursts of current are observed at 60 mA (200 mW), probably associated with the transfer of the data cached in the SD RAM to its flash memory [14] (Fig. 6). As a reminder, the software running on the custom MSP430F1611 based board we ran these measurements on clocks the SPI bus on the 4 MHz high frequency clock source.

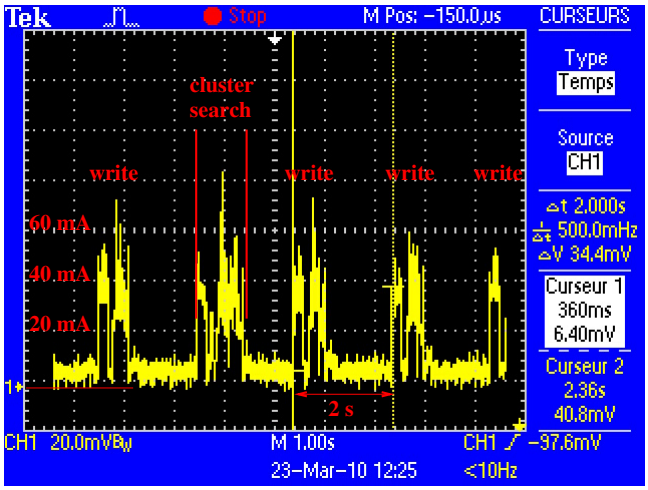

Figure 6. 512-bytes data blocks are stored on an SD card every 2 seconds. Every four writing operations, the search for a new available cluster requires additional accesses to the mass-storage medium, as seen here during the second access. Here the voltage drop over a 1 Ω resistor is displayed, so that the ordinal voltage reading (20 mV/division) also provides a current consumption in A, under a constant power supply of 3.3 V.

These short term current consumption seem to indicate that in terms of power management, storing data on non-volatile media yields power consumption of the same order of magnitude than low data-rate radiofrequency communication (bluetooth, or 802.15.4 and the often associated Zigbee protocol), but with wakeup times closer to those found in 802.15.4 (a few tens of milliseconds) than bluetooth (a few seconds). The bandwidth available for storing data is much greater (a few tens of kilobytes per second) than for radiofrequency data transmission, yielding shorter wakeup time and hence energy consumption. Obviously, the drawback of safely storing data on a non-volatile memory is that the data are not retrieved until a human operator actually fetches the memory task, a task sometime difficult depending on environmental/climatic conditions.

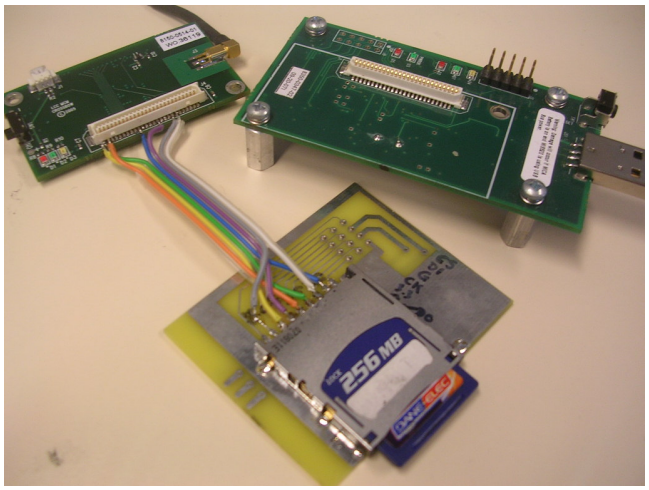

Figure 7. Experimental setup for validating the FAT16-formatted data storage on an SD card non-volatile mass storage medium connected to the USART1 port of a MicaZ OEM board.

In order to confirm these short term measurements, we performed a comparison of the evolution of battery level of 5 MicaZ and 2 TelosB motes used in the following way:

1. 5 motes communicate environmental data and battery voltage through the 802.15.4 link every two minute: two MicaZ motes are located about 10 m away from the sink, two MicaZ motes are close to the sink (about 50 cm) to assess the influence of the transmission range, one TelosB located close to the sink sends its battery voltage. Amongst the two sets of MicaZ motes located close and far from the sink, one continuously activates the wireless link peripheral (as would be needed to receive messages as well as send data) while the other releases the resource after sensing the data. The latter algorithm, allowing the processor to go into sleep mode, significantly extends the life expectancy of the node, while the two nodes continuously activating the wireless link die after 5 days of activity (Fig. 8).
2. two motes, one MicaZ and one TelosB, store environmental data and battery voltage on an SD card every 2 minutes, and transmit through the wireless link the battery voltage every hour. In all cases, the resources are released after completing their task. The SD card storage yields an increased power consumption with respect to the direct data transmission, but significantly saves energy compared to the strategy requiring constant radiofrequency peripheral activity.

Although the evolution of the battery voltage is an indirect indicator of the current consumption, it is nevertheless the actual quantity of interest to the final user. One observes that after 5 days the motes which keep the radiofrequency module continuously active die since the supply voltage reaches the threshold voltage of 2 V below which the mote no longer

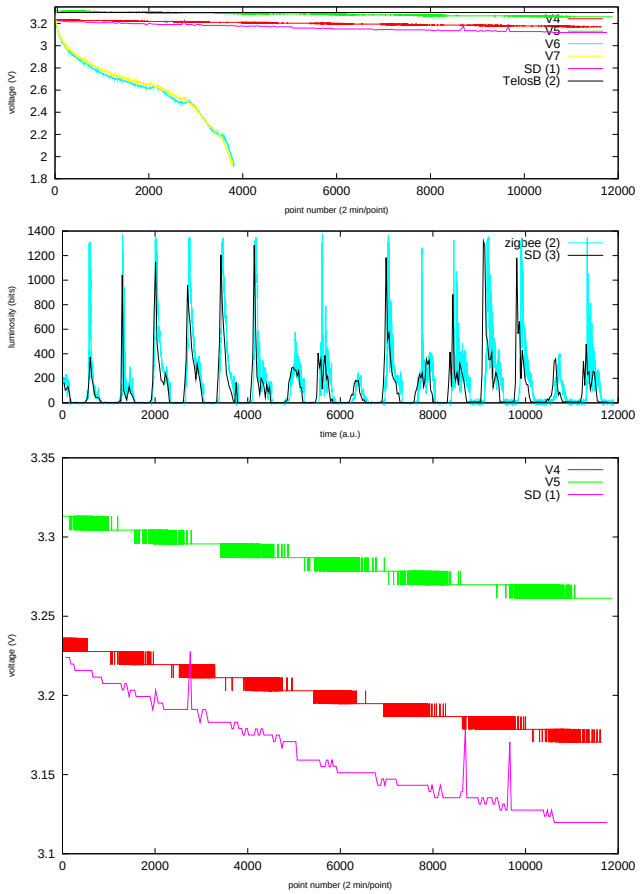

Figure 8. Top: comparison of the evolution of the battery voltage of one TelosB and five MicaZ motes: two of each platform were transmitting battery voltage and physical parameter measurements ever 2 minutes through a wireless link without releasing the peripherals after use (hence the excessive power consumption: sensor nodes 6 and 7), two of each platform were running the same program while releasing the resources (sensor nodes 4 and 5), and one of the MicaZ is storing data on an SD card every 2 minutes while transmitting its battery voltage every hour (sensor node 1). Middle: luminosity measurement on the TelosB node, providing an intuitive marker of time evolution (each luminosity maximum is reached around noon). Bottom: zoom on the power consumption of nodes 4, 5 and the MicaZ platform storing data on the SD-card. An energy saving strategy induces less power consumption on a 802.15.4 network: the SD-card equipped node exhibits and increased power supply voltage drop with respect to the nodes using wireless transmission.

runs. The mote storing data on the SD card still runs fine after $\simeq$ 20 days, but consumes significantly more power than the motes transmitting data through the 802.15.4 link and releasing the resource after sending the data. Finally, distance (and hence transmitted power) does not seem to significantly alter the global power consumption, and environmental conditions (temperature due to heating from the sun illuminating the mote) seem to be a more significant factor defining battery life. This last conclusion induces significant design constraints when developing sensor nodes for cool regions: keeping the battery in ice or in dark packages for improved heating by the sun might significantly extend the sensor node life expectancy.

We have demonstrated storage of large amount of data (>1.5 MB/hour) which would otherwise require, beyond the data acquisition time, another 20 minutes (at a constant emission rate of 9600 bauds as often found on 802.15.4 or bluetooth interfaces, assuming no additional delay due to wireless protocol management) to transmit through a wireless link. Hence, we believe that many applications in which the data are not needed in real time, but are to be recovered by non-technically oriented operators using mobile (laptop) personal computers running commonly found operating systems, would benefit from using a FAT16 filesystem as demonstrated here. The practical demonstration of long term data storage in an application incompatible with short range wireless data transmission is illustrated in Fig. 9, with the logging over 11 hours of the GPS position of a moving vehicle, recorded every second (continuous NMEA sentences transmitted by an ET312 GPS receiver), yielding a 9.2 MB file which does not exhibit visible data storage error.

All necessary software – platform and application files for TinyOS-2.x as well as some basic sample programs including recording data streams acquired from the asynchronous RS232 port and analog to digital converters, stored in different files on a FAT filesystem, is available for download at `http://www.trabucayre.com/` and to the link associated with this manuscript at `http://jmfriedt.free.fr`.

6 Conclusion

We have presented and characterized the use of a commonly available filesystem – FAT16 – for the storage of large amount of data recorded by a sensor node. Data integrity was assessed for transfer rates up to 9600 bauds. The implemeted driver emphaseized portability, low memory footprint and provides a sleep mode in which some variable store the state of the storage medium, saving initialization time upon resum, as often needed in periodic data sampling applications.

While power consumption during storage on non volatile Secure Digital (SD) card media is of the same order of magnitude than real time transmission over a low-power wireless communication protocol, we propose that local storage is safer and requires less time (and hence global energy consumption for data retrieval and storage) than real time wireless transfer. We consider this strategy as a safe alternative to a wireless link in remote areas where the closest basestation is further than the communication range of the wireless network hardware used, as long as a human operator has access to the network node. Furthermore, using a filesystem compatible with most operating systems running on personal computers widens the audience of operators on the field able to retrieve the data stored on the non-volatile medium.

Acknowledgment

J.-MF wishes to acknowledge the French National Research Agency (ANR) funded program HydroSensorFLOWS, under the direction of M. Griselin and C. Marlin,

Figure 9. Example of storing a GPS track onboard a moving vehicle – incompatible with low-power radiofrequency data transmission – driving from the west to east of France – a total of 9.2 MB stored during the 11 hour trip. The red track is an overlay over the satellite view of the path followed during the trip.

for motivating this work on non-volatile mass storage. Although the software and hardware described in this presentation were not used during this program aimed at monitoring the behavior of a polar glacier, the working conditions close to Ny Alesund (Norway) and its radio-telescope preventing the use of radiotransmitters (including the ones running in the 2.45 GHz ISM band, used by 802.15.4 and bluetooth) prompted this study. The interaction with end-users working in the fields of hydrology and geography emphasized the need for user-friendly interfaces compatible with standard widely available on most personal computers.

7 References

[1] L. Krishnamurthy, R. Adler, P. Buonadonna, J. Chhabra, M. Flanigan, N. Kushalnagar, L. Nachman & M. Yarvis, *Design and deployment of industrial sensor networks: experiences from a semiconductor plant and the north sea*, Proc. of the 3rd international conference on embedded networked sensor systems, pp.64-75, (2005)

[2] T. Liu, C.M Sadler, P. Zhang & M. Martonosi, *Implementing software on resource-constrained mobile sensors: experiences with impala and zebranet*, MobiSYS'04 Proceedings (6-9 Juin 2004)

[3] M. Ilyas, *The handbook of Ad Hoc Wireless Networks*, CRC Press, 2003.

[4] G. Marthur, P. Desnoyers, D. Ganesan, and P. Shenoy, *Capsule: an energy-optimized object storage system for memory-constrained sensor devices*, Proc. of the Fourth ACM Conference on Embedded Networked Sensor Systems (Sensys), 2006

[5] N. Tsiftes, A. Dunkels, Z. He & T. Voigt, *Enabling Large-Scale Storage in Sensor Networks with the Coffee File System*, Proceedings of the 2009 International Conference on Information Processing in Sensor Networks (2009)

[6] American Society of Civil Engineers, *NAVSTAR Gobal Positioning System Surveying*, American Society of Civil Engineers Press, 2000

[7] FAT File System: The Story Behind Innovation, available at `http://web.archive.org/web/20040214211109/http://www.microsoft.com/mscorp/ip/tech/fathist.asp`

[8] L. Selavo, A. Wood, Q. Cao, T. Sookoor, H. Liu, A. Srinivasan, Y. Wu, W. Kang, J. Stankovic, D. Young & J. Porter, *LUSTER: wireless sensor network for environmental research*, Sensys (2007)

[9] S. Patel, K. Lorincz, R. Hughes, N. Huggins, J.-H. Growdon, M. Welsh, P. Bonato, *Analysis of Feature Space for Monitoring Persons with Parkinson's Disease With Application to a Wireless Wearable Sensor System*,Engineering in Medicine and Biology Society, 2007. EMBS 2007. 29th Annual International Conference of the IEEE, pp.6290-6293, 22-26 Aug. 2007.

[10] J. Polastre, R. Szewczyk, and D. Culler *Telos: Enabling Ultra-Low Power Wireless Research*, IPSN 2005.

[11] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and D. Culler, . *TinyOS: An Operating System for wireless Sensor networks. In Ambient Intelligence*, Springer-Verlag, 2005.

[12] MSP430x1xx Family User's Guide (2006), `focus.ti.com/lit/ug/slau049f/slau049f.pdf`

[13] L. Selavo, G. Zhou, and J.A. Stankovic, *SeeMote: In-Situ Visualization and Logging Device for Wireless Sensor Networks*, Broadnets, 2006.

[14] *SanDisk Secure Digital Card, Product Manual*, Version 1.9, Document No. 80-13-00169, December 2003 available at `http://www.cs.ucr.edu/~amitra/sdcard/ProdManualSDCardv1.9.pdf`