A File System for System Programming in Ubiquitous Computing

Christian Decker, Michael Beigl, Albert Krohn

Telecooperation Office (TecO), University of Karlsruhe,
Vincenz-Priessnitz-Str. 1,
76131 Karlsruhe, Germany
{cdecker, beigl, krohn}@teco.edu
http://www.teco.edu

Abstract. In Ubiquitous computing small embedded sensor and computing nodes are one of the main enabling technologies. System programming for such small embedded systems is a challenging task involving various hardware components with different characteristics. This paper presents a file system for sensor nodes platforms providing a common organization structure and a lightweight and uniform access model for sensors and all other resources on sensor nodes. This mechanism forms an abstraction from different hardware, makes functions reuseable and simplifies the development on such systems. With ParticleFS an file system implementation on a sensor node platform is shown. As an example a telnet application running on sensor nodes was implemented demonstrating the usage of the approach for system programming on such platforms.

1 Introduction

In Ubiquitous computing (Ubicomp) and Pervasive Computing environments people are surrounded by a multitude of different computing devices. Typical representatives are PDAs, PCs and - more and more - embedded sensor nodes. Platforms are able to communicate, preferably wireless, and exchange information with each other. By collecting and interpreting information from sensors and network such devices can improve the functionality of existing applications or even provide new functionality to the user. For example, by interpreting incoming information as a hint to the current situation, applications are able to adapt to the user requirements and to support him in various tasks. Prominent examples where such technology is developed and tested are AwareHome[10] and Home Media Space[13].

An important area within Ubicomp is the embedding of sensor nodes in mundane everyday objects and environments. Such applications were explored for instance in a restaurant context at PLAY Research[12]. Within the scenario sensor enhanced objects supported dynamic workflows, information displays, and new interactions. Implemented applications checked for freshness of food or automatic negotiation of prices between menus depending on their history. The central enabling technologies for this application are small sensor nodes that are embedded into various objects in the restaurant. Devices used in this restaurant setting were Smart-Its[1], but similar devices exist including

Berkeley Motes[9], Ember[6] and MITes[15]. In general, these devices integrate computing capabilities – primarily an 8bit microcontroller – a wireless communication protocol – often customized – and various sensors. Most of the device platforms follow a modular concept where additional sensors can be added according to the needs of the application. The devices are battery powered and use energy saving mechanisms to last for months or even years. In particular for integration aspects they are very small. Figure 1 shows a 1 cubic centimeter (cm^3) device with a microcontroller, communication interface, sensors and battery.

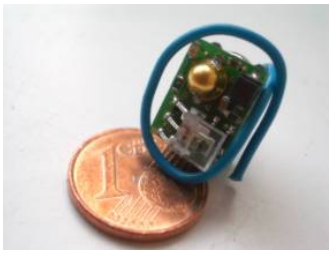

Fig. 1. TecO's Particle[16] (1 cm^3 integrating microcontroller, sensors, communication, battery)

System programmers of such embedded and integrated devices have to be aware of many constraints. The microcontroller is often a resource restricted 8-bit type, providing typically between a few kilobytes (KB) up to 512KB of Flash memory for storing programs and also small amount of RAM. Application programs are primarily written in assembly language or C. As a consequence of the embedding in everyday objects, applications heavily access the sensors as primary information source. Most systems provide developer support in one of two ways: An operating system protects access to resources by shielding lower level functions from direct access through the application. Communication between applications and system is made through events, such as in the TinyOS[9] used by the Motes. The other possibility is a library shielding the access to (sensor) hardware by providing abstract function interfaces. This concept avoids the overhead of event dispatching. Common to both methods is that they are not able to completely shield and protect the application due to restrictions of the used microprocessor platforms.

The approach we chose to follow in this paper, when designing the system software, places the developer at the center of interest. Our goal is to maximize the support for the system programmer, who is implementing applications for Ubicomp settings. We believe that a compact, simple-to-understand and simple-to-remember programming interface contributes most to the support for such a programmer. This assumption is supported by experiences that we gathered in various Ubicomp development projects presented in the next section. These experiences are motivated by examples in the next section. Based on these findings we present our system approach in section 3. In the center of this architecture stands the concept of a file system. All resources of the system are accessible only through the file system via a uniform access method, formed by six primary operations of the file system. We believe that this method of access is most appropriate for the programmers of small, embedded, wireless devices, as it provides a lightweight and compact interface to system functionality. We will show that such a compact file system architecture is also appropriate in terms of low resource consumption.

2 Analysis: Ubicomp Development

We implemented various applications on the wireless networked embedded sensor platform Smart-Its Particles[16]. Many of these applications are clustered in larger settings. An example is the Aware Office[2] – an office environment running different applications distributed over several dozen of wireless embedded sensor and computing nodes. Most of these nodes are embedded into everyday objects such as chairs, tables, windows, pens and whiteboards. The available applications support different activities in office settings including meeting support, activity and occupation detection. When implementing such applications programmers can choose from a variety of different sensors on the Smart-Its Particle hardware. It is known that the use of multiple sensor sources tend to improve the quality of the output. Nevertheless we experienced that in practice programmers use mostly one sensor as input to the system. They also tend to use the sensor they have used in a project before if applicable or a sensor where they can re-use example programs. Being asked for the reason they answered that from their experience their unfamiliarity with (the access functions of) other sensors would delay the development process.

Other experiences with a library based access interfaces to resources come from the eSeal[5] project. In this project sensor devices are embedded in physical goods measuring environmental and logical conditions in which the goods are situated. The nodes acquired, interpreted and shared sensor data among each other in order to detect violations pre-defined limits. This compelled the programmer to use several different sensors. Sensors are accompanied by various options to enable and disable them in order to save energy. In the analysis we found that the system programmers always used the simplest but most energy consuming way to power sensors on and off. Although this results in higher energy consumption programmers explained that the complexity of various interfaces but also of the overall distributed application is too high, so they decided to lower complexity on other parts of the program as much as possible. Another finding in this context was interface breach. Each sensor is accessible through its own interface taking the sensor's features into account. This interface is often only slightly different from another sensor's own. Nevertheless, even though the difference is only small, development got often stuck in debugging because of not noting the differences in the interface when accessing the sensor.

From the findings in the conducted experimental analysis we conclude that supporting the programmer to follow the simplest and uniform way to access sensors should be major principles for system programming. These principles also apply to other resources like communication and memory of the sensor nodes. Such an approach allows programmers freedom to concentrate more on other important aspects like distributed programming logic.

We propose a file system as an appropriate abstraction for implementing these principles. The coherent, hierarchical name space is able present resources in general. Therewith, it creates a clear and simple-to-recognize structure of all resources. The file system operations are applicable to all files and form an uniform access model on the structure. This enables the system programmer to follow the simplest way in the development process. Finally, we believe, that the familiarity and long-term experience of system programmers with established file systems will help to standardize programming on the various present and upcoming sensor node platforms.

The paper continues with the system architecture of a sensor node incorporating a file system. The understanding is deepened in an analysis of the resources of a sensor node. As a result a suitable name space and operations of a file system are derived. In section 4 we present our implementation of ParticleFS and discuss its performance. With the telnet application utilizing the ParticleFS in section 5, it is demonstrated how a file system supports the system programmer. Section 6 discusses related work on file systems before the paper is concluded in section 7.

3 System Architecture

The file system provides a uniform access layer for an application on top of all resources available on sensor nodes (figure 2). An application has access to direct and mediated resources via the file system. Direct resources are the representation of available hardware on the sensor node. Mediated resources aggregate and interpret information from direct resources, but can also represent other functionality provided by an application. If mediated resources access direct resources, this is also carried out via the file system.

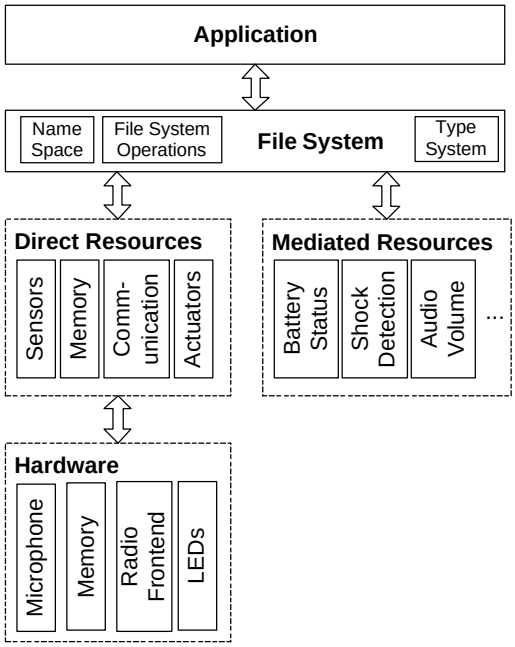

Fig. 2. System Architecture of a Sensor Node integrating a File System

The file system is composed of a name space presenting all resources in a hierarchical structure, file operations responsible for the uniform access to the resources and a type system supporting compatible operations on multiple resources. It is important to note, that the application has no direct access to the hardware. All calls are made through the file system. In the next subsections the components of the architecture will be explained in detail.

3.1 Resources

The above-mentioned sensor nodes (Motes, Ember, MITes and Particles) have a microprocessor unit (MPU) at their core and incorporate various sensor, wireless communication, memory and actuation hardware. They are typically powered by regular batteries. We will analyze available resources and their methods of access on these sensor nodes and thereby distinguish between direct and mediated resources.

Direct Resources
Direct resources are represented by hardware components available on the sensor node device. The sensor nodes have access on

- various sensor hardware, e.g. light, audio, acceleration, temperature sensors,
- wireless communication interface
- memory in form of internal, i.e. contained in the MPU, and external memory
- power supply, typically regular batteries
- actuators, e.g. LEDs, buzzer or small display

Sensors. Sensor hardware is accessed with a multitude of possibilities. Analog sensors are sampled via the MPU's analog-digital converter. Digital sensors provide for instance a duty cycle, which needs to be sampled and interpreted, or they provide a bus interface. Typical interfaces used on sensor nodes are I2C and 1-Wire bus.
Communication. A widely used access method to the wireless communication interface is a serial line communication connected to communication component. The communication component with the transceiver is then responsible for the channel access, data modulation and the communication protocol. On other platforms like Smart-Its Particles the transceiver is completely controlled by the MPU, including methods for channel access, data modulation and an own communication protocol.
Memory. The MPU's internal memory consists of Flash-ROM for programs and RAM for program variables. The Harvard architecture common for MPUs on the considered platforms imposes the separation of both types. A system programmer has only limited control over the internal memory because the compiler determines the usage. This makes the internal memory inappropriate to be accessed via a file system. External memory is included in form of flash memory or EEPROM devices. Storage for arbitrary data is provided ranging from a few kilobytes up to half a megabyte. External memory components provide typically a serial interface like I2C or SPI interface.
Power Supply. Energy is one of the key resources for sensor node platforms. Batteries are a crucial component and often limit the usage of the platform. Having access to this resource enables the application to optimize its runtime behavior. Motes and Smart-Its Particles are both able to measure the voltage of the supplying battery.
Actuators. Embedded sensor nodes have capabilities to present states and events. Commonly used on all mentioned sensor node platforms are LEDs. Current Motes and Smart-Its Particles integrate a buzzer for acoustic notifications. Complex actuators are displays, controlled via serial line, I2C or by a proprietary protocol. However, displays are seldom used, because they require a lot of energy and often the embedding in everyday objects prohibits their usage.

Mediated Resources
These resources apply operations like combination, aggregation an interpretation on data from direct resources. An example for a mediated resource is the average volume level of a sound source. It requires the aggregation of an array of audio samples from the direct resource microphone, which are then used to compute the average volume level. Mediated resources may access a multitude of other direct and also mediated resources. The concept of mediated resources can even be applied to functions within an application. A system programmer can separate functions from the application and present them in the file system. In this way even pure computational functions can be presented as mediated resources. In order to serve as a uniform access layer to direct and mediated resources a file system has to organize them in an appropriate name space. In order to retrieve information from these resources a file system needs a suitable and applicable set of operations for accessing these resources.

3.2 Name Space

In the file system resources are organized in a hierarchical name space. Files are the smallest entities and directly identify resources. Files are further organized in directories which are special files identifying a collection of files. By recursion a file tree can be built up with a single top directory representing the root. A resource in this tree is then clearly identified by the complete path starting from the root down to the single file along the tree structure. In order to separate the directories from each other along this path the delimiter "/" is used. A single "/" indicates the root directory. There are 3 predefined directories in the root directory. The directory /dev/ holds all direct resources. In /context/ the system organizes mediated resources based directly or indirectly on sensors. Finally, resources for storing application data are located in /usr/. Within these directories the resources may further hierarchically ordered using subdirectories. Arbitrary data is physically stored in the external memory which is represented as the direct resource /dev/eMem. In order to include those data in a structured manner, the /usr/ directory contains a file system view of the /dev/eMem resource. This is considered as a mediated resource and therefore files in /usr/ are also mediated resources.

Resource	Explanation
/	This identifies the root directory.
/dev/	Directory containing files providing access to direct resources.
/dev/SLI0	File representing a light sensor. SLI1 is the second one, if available.
/dev/SAU	This is the file for retrieving for sampled the microphone.
/dev/SVC	This is the file for retrieving the battery voltage.
/dev/eMem	The external memory is accessed for reading and writing via this file.
/dev/comm.	This file provides access to the communication interface.
/context/	This directory contains files for accessing mediated resources
/context/audiovolume	This file is for computing the audio volume
/context/batterystatus	Mediated resource describing battery in 3 states (full, good, weak)
/usr/	Application stores arbitrary data as regular files in this directory.
/usr/myfile	File containing arbitrary application data; created by an application

Table 1. Example of a Name Space of Resources presented as File System

In table 1 we present an example of a name space. For sensor hardware in the /dev/ directory we used a three-letter-abbreviation indicating a sensor. We found it more expressive than the technical name of the sensor and reusable on different platforms. For instance, SLI indicates a Sensor for LIght. If there is more than one sensor of a type available, the sensors may be enumerated by an additional number behind the identifier.

3.3 File-based Operations

After the analysis of available resources and organizing them in a name space, a suitable set of operations is needed in order to access them. A communication interface is accessed by the widely-used operations send() and receive(). Similar to that, widely-used access methods for memory are read() and write(). Both approaches abstract from detailed processes going on below, e.g. a specific organization of the memory, or channel access of the communication interface. This enables a shielding of hardware differences on different platforms. Taking those considerations into account we analyzed the usage of read() and write() for other resources. Both are generically understood for transferring data in both directions which makes them suitable for accessing various resources. We identified them as fundamental for the access model. However, their implementation is different depending on the platform and the resources. As a consequence we demand that each resource is coupled to its specific read() and write() operations. Hereby, the file system only references these specific operations. Their implementation remains in a specific access library for a resource. In that way an abstraction is achieved, since the generic read() and write() are called, but the specific access behavior on a resource is kept by transparently calling the specific read() and write() through the file system. Additionally, every resource presented as a file is coupled with a type as a file attribute. Types can be used to check for compatibility when accessing different resources in combination. This is important for mediated resources since they may be derived from other resources. The type system will be explained in more detail in the next section. The access model supports types in the process of mounting and by providing type queries. The set of operations is summarized in table 2. It bases on our long-term practical experience with the Smart-Its Particle platform and insights of similar platforms. We use the C syntax for the operation set. Note, size_t is an abstract data type which can be replaced with an appropriate platform specific one.

Operation	**Explanation**
size_t read (int fd, void* buf, size_t n)	Reads n data bytes from the resource identified by fd to buf; returns number of bytes or -1 if error occurred
size_t write (int fd, void* buf, size_t n)	writes n data bytes from buf to the resource fd; returns number of bytes or -1 if an error occurred
int open(char* resource_path)	Returns a descriptor for the resource; -1 if it is not valid.
int getType(int fd)	Returns the type of a resource fd; -1 if fd is not valid.
int mount (char* resource_path, int type, (*pFunc) read, (*pFunc) write)	Creates a resource in the name space. Type and function pointers to their specific read and write operations are given. -1 is returned if the resource_path already exists.
int umount(char* resource_path)	Removes a resource; -1 is returned if it is not valid.

Table 2. Generic Access Functions of the File System

Our access model is now based on read() and write() operations which are coupled to specific resources and a type system supporting compatible operations across different resources. Both read() and write() operations are also fundamental in POSIX[17]. POSIX, the portable operating interface, defines data types, return codes, functions for file operations, process handling, security issues and error reporting. The definitions describe an interface which is designed in a way making them easily portable to other systems. Both operations apply to all functions (e.g. processes, file operations etc.) within the standard. Indeed, it was intended to use the POSIX syntax for read() and write() operations. From the POSIX point of view, we achieved a very lightweight file access model by leaving out access permissions, security issues and process handling. Return codes where simplified and there is no detailed error reporting mechanism like errno() in POSIX.

3.4 Type System

The type of a resource allows to check for compatible resources, when using them together. For example: The communication interface may not accept raw sensor values from the /dev/ resources. They should be extended by meta information. This is done by reading them not directly from /dev/, but from a mediated resource in /context/. By checking the type of the resource the communication interface /dev/comm can decide whether to accept or not accept the resource. Furthermore, higher-level functions aggregating data from low-level resources can adapt their behavior according to the types. The designed file system knows the following types:

Type	**Explanation**
directory	indicating directories, such as /dev/ or /usr/.
regular file	each file containing arbitrary data in /usr/ is considered to be a regular file.
custom types	these types are defined by a system programmer for all other resources, e.g. each file identifying a direct resource in /dev/ has its own type.

Table 3. Types used within the File System

While the directory type and the regular file type are integrated in the system, custom types can be defined freely. It enables the system programmer to define the compatibilities between resources. However, the system programmer has to take care of declaring resources with consistent types. For instance, using resources together which work on different data formats, but are accidentally declared with the same type may result in unexpected behavior of an involved resource. The system does not provide checks for such inconsistencies.

4 Implementation

We implemented the previously suggested design in a file system called ParticleFS for our Smart-Its Particle platform. The devices comprise a communication board with a PIC18f6720 microcontroller. The communication uses a TR1001 transceiver with the customized protocol AwareCon[3] especially designed for ad-hoc networking in Ubicomp environments. Furthermore the board carries a 512KB flash memory component.

Various forms of sensor and other add-on boards can be attached to the communication board. The following sensors are currently implemented on the sensor board: two 2D-accelerometers enabling the measurement in three dimensions, a light sensor, a microphone, a force sensor and a temperature sensor. The boards are powered by a single AAA battery. The implementation was carried out with the goal to limit the resource consumption of the file system. Internal memory usage regarding RAM and ROM on the microcontroller was aimed to be kept minimal. Furthermore, the file system's effects on the overall runtime behavior were analyzed in order to estimate consequences for calls on hardware operations, e.g. sensor sampling via the file system.

4.1 ParticleFS

ParticleFS implements a main table, a subdirectory table and a file storage table (figure 3). The main table holds all resources including directories, the type of the resource and the function pointers to the resource specific read() and write() operations. Each entry is preceded with a number – the resource descriptor. Additionally, the main table is linked to the subdirectory table. Latter table orders all resources or directories from the main table by referencing back all subdirectories and resources for each directory.

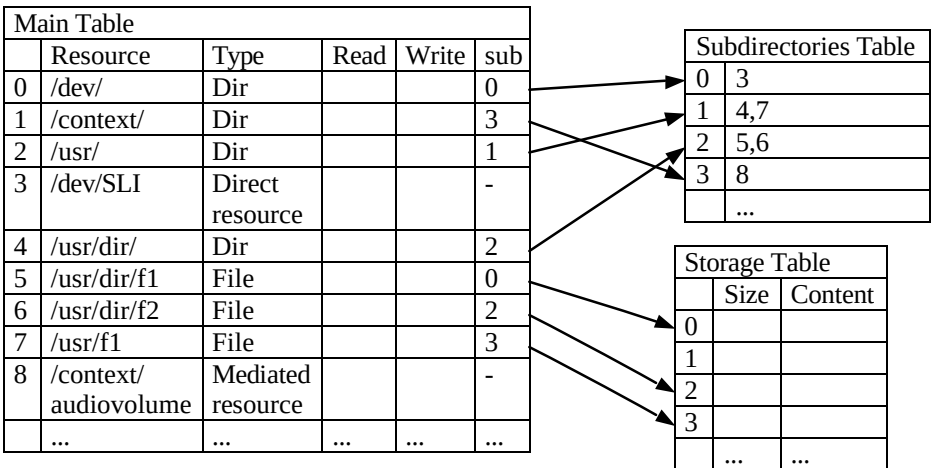

Fig. 3. File Tables of the ParticleFS (function pointers are left out)

Regular files in the main table are also referenced by the subdirectory pointer. However, since the type of regular files differs from the directory type, the pointer is interpreted as a reference to the storage table. Each entry of that table represents the content of a regular file and is preceded by its file size attribute. The storage table is stored on the external flash. Therefore only the first two tables are kept in RAM, minimizing the overall consumption. Directed and mediated resources are only referenced in the main table along with their function pointers. As a positive effect, the system programmer can benefit from this organization since it enables an easy way to re-use common functions.

4.2 Functionality of ParticleFS

During the boot up of the hardware when the ParticleFS starts it mounts several resources. It creates the /context/ directory for mediated resources, the /usr/ directory for file storage and the /dev/ directory for direct resources. The file system also mounts the known sensors in the /dev/ directory. Resources mounted at startup, in particular sensors in /dev/, are not expected to be umounted during the runtime of the file system. Each sensor driver has to provide a specific read() and write() function referenced in the file system. Additionally, there is a file library providing read() and write() functions for regular files and directories. Now, consider the following two representative scenarios:

Access a direct resource (e.g. /dev/SAU – the audio sensor). Figure 4 illustrates the access of the audio sensor through the file system. In the first step a call of the open() operation is needed to retrieve the resource descriptor from the main table. The system's read() operation uses the descriptor to retrieve the function pointer for the resource's specific read() function, here readMic(). The specific read() function is called and starts sampling the microphone using the MPU's analog-digital-converter. The sampled values are then returned in the given buffer from the read operation.

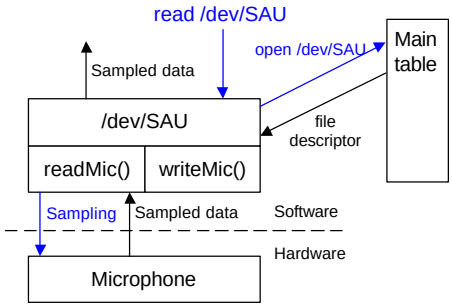

Fig. 4. Reading a Sensor (Direct Resource)

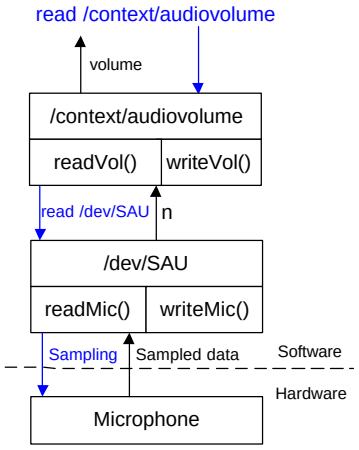

Fig. 5. Reading the Audio Volume (Mediated Resource)

Access a mediated resource (e.g. /context/audiovolume). In figure 5 the file system's read() operation will call the specific read() function of /context/audiovolume, here readVol() in order to compute the average volume for n samples from the microphone. That read() function will invoke the system's read() operation for /dev/SAU like in the previous scenario, but n times. The resource descriptor of /dev/SAU can be re-used minimizing the effort to find the entry in the main table when the audio sensor is accessed again. Finally, the average volume for the period of samples will then be computed and returned to the file system's call on /context/audiovolume.

Accessing sensors does not necessarily mean only read accesses. All sensors have to provide a specific write() operation. Usually, it refers to an empty function. However, some sensors implement a non-empty one for configuration purposes. For instance, write() functions of analog sensors may configure the bit resolution of the analog-digital converter when sampling.

4.3 Discussion of ParticleFS

In our current implementation the main table can hold up to 50 resource entries where each one is 30 characters long at maximum. Additional for each resource is a type of one byte, function pointers for the specific read/write functions – each 2 bytes long – and the subdirectory pointer consisting of one byte identifying the position of the entry holding all subdirectories or resources in the parent directory. The subdirectory table can reference 15 subdirectories or other resources for each parent directory. However, there are only 10 parent directories possible. It is important to note that the focus of the implementation was not to create deep and complex file trees. We do not believe that this will be convenient on the systems we are targeting. We expect that the structure is rather flat with many resources distributed over a few directories. The storage table uses the external flash of 512KB to store files. The structure of this table imposes a 50 byte segment alignment of all data. Files smaller than this size do not use the remainder. Larger files will allocate a multiple of 50 bytes segments. The design decision was made according to our expectation, that our system will handle large file for sensor logging, and small files for intermediate results or configuration data. Altogether, the file system consumes 1850 bytes of RAM for holding the tables and has access to the complete external memory. However, in future implementations parts of the main and subdirectory table will be swap out to the external memory. The implementation of the minimal set of operations from section 3.3 required 2106 byte program Flash memory.

Currently, the ParticleFS implementation does not support an event mechanism and has no notion of interrupts. All file system operations run synchronously, i.e. the call returns after the completion of the operation. When accessing a resource through the file system the resolution process to determine the specific read() or write() function slows down the execution process. However, once the file descriptor is obtained, it can be reused for further uses. So, each access to a resource is then only preceded by a table lookup for the specific read() or write() function and call of this function. On the Particles' 18F6720 MPU the delay for the lookup and the additional call is below 5 microseconds. The delay of the analog-digital-converter to sample a new sensor value is about 30 microseconds. I2C sensors are even slower. So, the resolution of the access through the file system had no noticeable effect on sensor sampling.

Both read() and write() operations are character oriented. We found, that exchanging arrays of byte is the most generic way to deal with the diversity of resources. Mediated resources can transform those raw data to structured ones and can further work with them. In order to ensure the semantic meaning the types were introduced to check for compatibility. Nevertheless, the view on the resources is local. Up to now, we have not implemented a mechanism which can combine file systems of multiple device.

The current implementation for regular file storage does not focus on frequent updates of files. Such updates will trash the storage table. As a consequence, a continuous writing of a large file for sensor logging might fail, if there is an insufficient number of consecutive 50 bytes segments left. Compacting files regularly may be a solution. However, read() and write() calls on regular files are resolved to specific functions in the file library which is not part of the file system. This flexibility makes it easy to transparently implement a new concept for storing data files. This shows how the file system fulfills the principles from the motivation by providing a uniform access to resources and supports the system programmer to follow a simple way.

5 Application: Telnet for Smart-Its Particles

Telnet is a console to remotely access all functionality of the file system ParticleFS and let users inspect all resources of an embedded sensor node interactively. It enables a user to actually login on the Particle and browse the file system, read sensors and activate actuators.

5.1 Components

The telnet application is divided into a proxy and a telnet server. The proxy as shown in figure 6 runs on a personal computer connected to a UDP network. It is responsible for sending commands to the server running a specific. A bridge in the UDP network transports messages from the proxy to the RF network of the Smart-Its Particles and vice versa. The proxy awaits a connection from a regular telnet client. If connected, each string encapsulated in the telnet protocol from the client is extracted by the proxy's command parser and given to the Particle communication module. The latter is sending this string to the telnet server encapsulated in the Particle protocol.

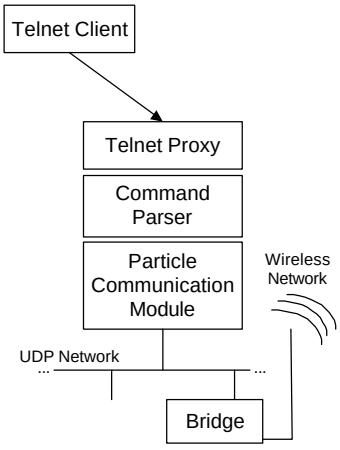

Fig. 6. Telnet Proxy on Personal Computer

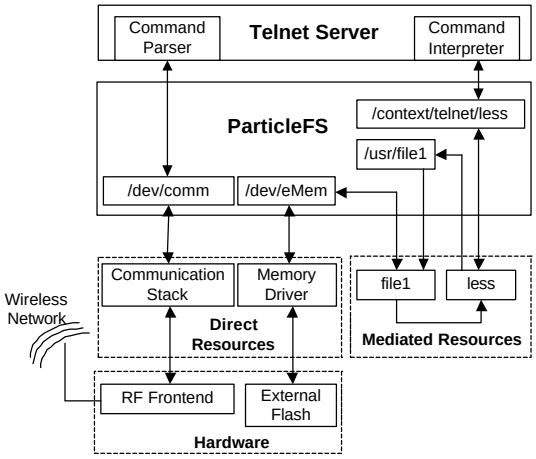

Fig. 7. Telnet Server on Particle demonstrating "less /usr/file1"

The telnet server is implemented on a Particle (figure 7). It places a command interpreter on top of the file system. The interpreter enables more complex operations for inspecting, copying, moving or deleting of regular files. A command parser reads strings from the communication interface using the file system's /dev/comm resource. The strings are parsed and forwarded to the interpreter for invocation. All interpreter operations are mounted in the file system as mediated resources under /context/telnet/ and can be called as commands in the telnet console.

5.2 Functionality of Particle Telnet

Once a telnet client is connected to the proxy and finally to the telnet server on the Particle, commands can be remotely invoked on the Particle's file system. We representatively describe the invocation of the command "less /usr/file1" as it is shown in figure 7. The command displays the content of a file /usr/file1. We present further commands where the file system supports the telnet application.

The command parser parses "less /usr/file1" and identifies the command, here "less" and the parameter, here "/usr/file1". These information are given to the interpreter which invokes "/context/telnet/less" by calling the file system's read() operation on it. The file system resolves this call into the mediated resource "less" and invokes the specific implementation for read() on it. In the same way the "less" resource resolves the parameter "/usr/file1" into the "file1" resource. By invoking "file1" it accesses /dev/eMem and therefore resolves this direct resource into the read() function in the memory driver. Finally, this function accesses the file on the external flash memory. The file content is returned to next previous caller until it is written to /dev/comm by the command parser and then presented in the telnet console of the client.

Browsing and file operations. The interpreter provides commands like "dir", "copy", "delete", "write" for listing directories, copy, deletion and writing of files. The commands allow to interactively inspect the file system and all available resources on the embedded sensor node. Further, these commands support modifications of the file system, since one can create and delete own directories and files and store arbitrary data.

Accessing sensors. The "less /context/dev/SLI" command prints the current light sensor value on the console. However, the "less" command denies the access on the direct resource /dev/SLI, because those resources return raw data which may contain non-printable characters. The file system supports this behavior by providing resource type information. The "less" command decides accordingly, whether the access to a resources is granted or not. Access to resources under /context/dev/ is granted since they return sensor values in a printable format.

Combining resources. The file system supports combination of resources. Commands utilize resource type information in order to adapt their behavior accordingly. The command "write /dev/comm /dev/SLI" reads a light sensor value and sends it via the communication interface. The "write" command can also be applied for strings in combination with regular files, e.g. "write /usr/file 'hello world!' ". Hereby, the type information causes the write command to store the given string in /usr/file.

New commands. The set of commands which the interpreter can invoke may not be sufficient for an embedded sensor node. The file system contributes to the integration of new commands of the interpreter by mounting them as mediated resources in the file system. As a consequence, all telnet functionality is disclosed by adequate mediated resources in the file system. This enables a system programmer to easily add and interactively test new functions.

6 Related Work

UNIX operating systems and various derivates like Linux incorporate peripheral devices such as keyboard, mouse, sound card as special files in their file system. Plan9[14] implements this idea consequently. All resources are accessed in a file-based manner with

file-based operations. Our work was mainly inspired from this idea, but differs in the way the application sees the resources. In contrast to Plan9, there are no client specific local name spaces imposed on the file system. As a result file servers managing theses views are not needed. This forms a very lightweight access model especially appropriate for resource limited sensor nodes.

In Ubicomp the idea of file systems is also already established for system-wide data access and sharing method as well as for data storage on sensor nodes. Dynamo[11], a file system for smart room applications residing in Stanford's iRoom, is targeted on office collaboration scenarios. Hereby, file-oriented data is consistently replicated across various devices from personal computer infrastructure down to personal devices, such as PDAs. Further, users can choose situations like meetings, coffee breaks and others, in which they want to share certain portions of their file system. The context-aware file system (CFS) [8] integrates this principle consequently. The user's personal data are organized in directories using his name while context driven data is organized based on contextual information. Contexts may be physical presence, location or data format requirements of the user's personal device. According to the current context, the file system's awareness limits the visibility of data stored in this file system. These examples show, that file systems are thought of middleware solutions for data management in mobile scenarios. The Dynamo and CFS rely on strong infrastructure support provided by personal computers. The smallest entity using those middleware services is a PDA-like device. In contrast, our approach with ParticleFS is self-contained and implements the file system on individual sensor nodes, small enough to be integrated in everyday objects. Berkeley Motes and the BTNodes[4] from the ETH implement sensor node file systems. Motes follow with MatchBox[7] a very straight approach for storing data on an external Flash memory component. The file system supports only sequential reads and appending writes. This distributes write accesses over the entire Flash memory space and contributes to the memory’s life time especially under high write access loads. The Micro-ROM implementation of the BTNodes provides a simple program space file system. Hereby, data files are linked together as part of the application program. This results in a read-only data structure only suitable for the application it was linked with. In contrast to those examples we extend the file system concept by integrating other resources like sensors, memory and communication into the file system. Apart from data storage, the file systems forms a uniform access model and supports system developers for embedded sensor nodes.

7 Conclusion and Future Work

By bringing the file system approach down to the sensor node we continue existing system support concepts in Ubicomp. System programmers are provided with a compact, easy-to-understand interface, with which they are very familiar. The uniform interface and the consistent integration of all resources in one name space support system programmers to follow the simplest way during development. The capability of extension by just adding another file/resource let them easily add new functions on the sensor node while maintaining the uniform interface. Having the file abstraction directly on the sensor node enables also a homogeneous integration of small sensor systems into previous work in system support for Ubicomp in particular for middleware solutions.

In future work we will integrate an event mechanism, which is able to present and handle hardware interrupts and self defined events in the file system. The research on the minimal set of file system operation will proceed, as more experience will be gained. Future file systems for sensor nodes will also incorporate remote resources in the local file system. Finally, with the telnet application we have implemented an evaluation environment for rapidly testing these new file system capabilities.

References

1. Beigl, M., Zimmer, T., Krohn A., Decker, C., Robinson, P.: Smart-Its - Communication and Sensing Technology for UbiComp Environments. Technical Report ISSN 1432-7864 (2003)
2. Beigl, M., Zimmer, T., Krohn, A., Decker, C. Robinson, P.: Creating Ad-Hoc Pervasive Computing Environments. In the Adjunct Video Proceedings of the Pervasive 2004, Vienna, Austria (2004)
3. Beigl, M., Krohn, A., Zimmer, T., Decker, C., Robinson, P.: AwareCon: Situation Aware Context Communication. Proceedings of UBICOMP 2003, Oct. 12-15, Seattle, USA
4. BTnodes - A Distributed Environment for Prototyping Ad Hoc Networks. Available online: http://www.btnode.ethz.ch [Accessed: 01/2005]
5. Decker, C., Beigl, M, Krohn, A., Kubach, U., Robinson, P.: eSeal - A System for Enhanced Electronic Assertion of Authenticity and Integrity of Sealed Items. Pervasive 2004, Austria
6. Ember – Wireless Semiconductor Solutions. Available from: http://www.ember.com/
7. Gay, D. The Matchbox File System. Available online: http://webs.cs.berkeley.edu/tos/tinyos-1.x/doc/matchbox-design.pdf [Accessed: 01/2005]
8. Hess, C., Campbell, R. An application of a context-aware file system. In Personal and Ubiquitous Computing Volume 7, Issue 6, pp. 339–352, December 2003, ISSN:1617-4909
9. Hill, J., Szewczyk, R., Woo, A., Hollar, S., Culler, D., and Pister, K.: System Architecture Directions for Networked Sensors. ASPLOS-IX, 2000.
10. Kidd, C.D., Orr, R.J., Abowd, G., Atkeson, C.A., Essa, I., MacIntyre, B., Mynatt, E., Starner, T. and Newstetter, W.: The Aware Home: A Living Laboratory for Ubiquitous Computing Research. In the Proceedings of CoBuild'99. October 1999.
11. Lamarca, A., Rodrig, M.. Oasis: An Architecture for Simplified Data Management and Disconnected Operation. ARCS 2004, Augsburg, Germany
12. Maze, R. Holmquist, LE.: Smart-Its Workshop on Interactive Scenarios. Available online: http://play.tii.se/projects/smart-its/restaurant.html [Accessed: 09/2004]
13. Neustaedter, C. and Greenberg, S. The Design of a Context-Aware Home Media Space: The Video. Video Proceedings of UBICOMP 2003, Seattle, USA (2003)
14. Pike, R. Presotto, D., Dorwards, S., Flandrena, B., Thompson, K., Trickey, H., Winterbottom, P.: Plan 9 from Bell Labs. Computing Systems, vol.8, no.3, pp.221-254, 1995.
15. Tapia, E.M., Intille, S., Larson, K.: MITes: Wireless Portable Sensors for Studying Behavior. In Adjunct Demo Proceedings of UBICOMP 2004, Nottingham, UK (2004)
16. TecO Smart-Its Particle: http://particle.teco.edu [Accessed: 09/2004]
17. The OpenGroup. IEEE Std 1003.1, 2004 Edition. Available online: http://www.unix.org/single_unix_specification/ [Accessed: 09/2004]