

A File System for System Programming in Ubiquitous Computing

CHRISTIAN DECKER, TILL RIEDEL, MICHAEL BEIGL AND ALBERT KROHN

*TecO, University of Karlsruhe
Vincenz-Priessnitz Str. 1, 76131 Karlsruhe, Germany*

{cdecker, riedel, beigl, krohn}@teco.edu

Abstract In Ubiquitous computing small embedded sensor and computing nodes are 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 which organizes all computational and sensory functionality of a sensor node as resources in a uniform name space. It further provides a lightweight and uniform access model for all these resources. This mechanism forms an abstraction from different hardware, makes functions re-useable and simplifies the development on such systems. With ParticleFS a concrete file system implementation on a sensor node platform is shown. Application cases demonstrate sensor logging, an interactive shell, executables, a pipe mechanism and remote access capabilities of the ParticleFS.

Keywords: *Particle Computer, File System, Sensor Nodes, Ubicomp Development*

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[1] and Home Media Space[2].

The central enabling technologies for Ubicomp scenarios are small sensor nodes that are embedded into various objects or even attached to human beings. Devices

used in the research field include Smart-Its[3], Berkeley Motes[4], Ember[5] and MITes[6] and other similar devices. In general, these devices integrate computing capabilities – primarily an 8-bit 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[7] (1 cm^3 integrating microcontroller, sensors, communication, battery)

The sensor nodes run application programs for acquiring situational information using the nodes' set of several sensors as primary information source. Developers of such programs have to be aware of many constraints. For instance, the microcontroller is often a resource-restricted 8-bit processing unit, providing typically between a few kilo bytes (KB) and 512KB Flash memory for programs and only a small amount of RAM for volatile data. Most systems support the sensor data acquisition in one of two ways: An operating system takes care of the management of hardware resources and shields lower level functions from direct access through the application. Communication between applications and system is made through asynchronous events, such as in TinyOS[4] used by the Motes. Although this leads to a uniform abstraction between application and system, it imposes additional complexity for developers and in particular memory overhead through the handling of asynchronous events. The authors of [8] analyze this overhead in more details. The other approach is a library shielding the access to (sensor) hardware by providing abstract function interfaces. Synchronous calls through libraries avoid the overhead of event dispatching, reduce the complexity for the developer and can be optimized for resource-constrained sensor nodes. However, this concept obliterates a clear separation between application and system. As a consequence, the application ends-up in a variety of different

abstractions to lower level functions. Common to both methods is that they are not able to completely shield and protect the application due to limited capabilities of the used microprocessor platforms.

The approach we chose to follow in this paper, places the developer at the center of interest. In order to maximize the support for the developer of Ubicomp applications, we aim to combine the convenience of the library-based approach and the uniform abstraction formed by operating systems. We believe that a uniform, compact, and simple-to-remember programming interface contributes most to this support. This assumption is underpinned by experiences that we gathered in various Ubicomp development projects presented 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 a sensor node are accessible only through the file system via a uniform access model consisting of seven primary operations. As a result, we achieve a uniform abstraction between application and underlying sensor node platform with its resources and we keep the semantic of synchronous function calls. We believe that this method of access is most appropriate for the programmers of small, embedded, wireless sensor nodes. 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[9]. Many of these applications are clustered in larger settings. An example is the Aware Office[10] – 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 applications support different activities in office settings including meeting support, activity and occupation detection. When implementing such applications developers 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 developers 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[11] 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 developer 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 developers always used the simplest but most energy consuming way to power sensors on and off. Although this results in higher energy consumption developers explained that both the complexity of various interfaces and the complexity 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 developer 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 developers to concentrate more on other important aspects like distributed system logic.

We propose a file system as an appropriate abstraction for implementing these principles. The coherent, hierarchical name space is able to represent all resources. In particular, the name space allows to create meaningful names and groups of resources according to the developer's needs. Therewith, it creates a clear and simple-to-recognize structure of all resources. The file system operations are applicable to all resources and form a uniform access model on the structure. This enables the developer to follow the simplest way in the development process. Finally, we believe, that the familiarity and long-term experience of developers

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. Various application cases utilizing the ParticleFS are presented in section 5. The examples show how the file system supports the developer in many application aspects. 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 which are available on sensor nodes (Figure 2). In general, all computational and sensory functionalities of a sensor node are considered as resources and represented in the file system. 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. Mediated resources also access direct resources via the file system.

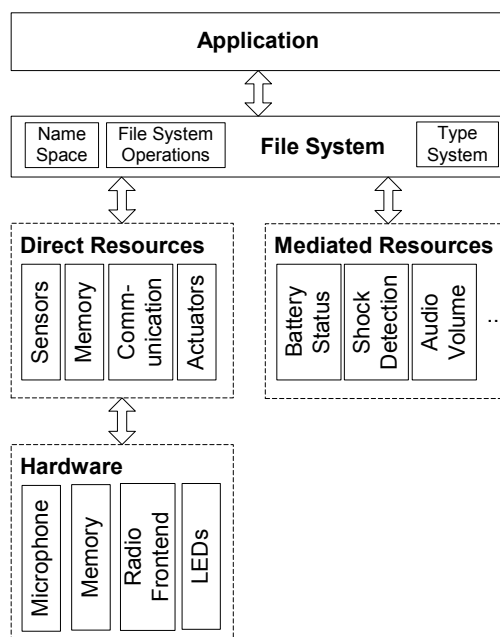


Fig. 2. System architecture of a sensor node integrating a file system

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.

3.1.1 Direct Resources

Direct resources are represented by hardware components available on the sensor node device. The sensor nodes comprise typically the following components:

- 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, speaker or small display

The access methods to those hardware components differ strongly. Sensors are sampled by the MPU's analog-digital converter or implement a separate protocol using bus interfaces like I2C, SPI or 1-Wire. The communication interface may be built on top of a serial connection to a transceiver. 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. The usage of the internal memory of the MPU is determined statically by the compiler. Sensor nodes usually do not implement a dynamic memory

management due to the limited memory resources. External memory offers storage space for arbitrary data and is accessed via I2C or SPI interfaces. Finally, actuators like speaker or displays implement proprietary protocol.

Due to the multitude of hardware components all sensor node platforms encapsulate the access to the components as drivers. Drivers represent resources at the bottom of the system architecture and the lowest level of abstraction for application developers.

3.1.2 Mediated Resources

These resources apply operations like combination, aggregation and 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 extends the concept of libraries. Like libraries it exploits the reusability of different functional modules. The developer benefits since he can specify an appropriate level of abstraction by creating a mediated resource combining a set of other mediated and direct resources. Different levels can be created by forming different aggregations of resources into a mediated resource. Nevertheless, since all resources are organized in one name space, the uniformity of all abstractions is guaranteed. This characteristic significantly extends the library based approach implemented on many nowadays sensor node systems.

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.

File names and therefore resource descriptions can be very expressive. A developer can specify any string and can create expressive human-readable identifiers independently from the underlying system. The hierarchical structure allows to group the resources together in any semantic arrangement defined by a developer. Additionally, the directory structure implements a generic method to navigate through these arrangements. This contributes positively to the application development process since the developer can individually follow the most expressive way for his task.

When different levels of abstractions are created through mediated resources, the name space guarantees uniformity of these abstractions. The details of the abstractions are hidden and not noticeable in the name space since all resources are handled in the same way, namely as files. Compared to an approach where abstractions are formed only by libraries, the name space can be considered as a meta library summarizing all libraries in a single and uniform organizational structure. However, it does not work as a prefix to these libraries, but rather as a organizational structure for expressive resource description representing the library abstractions.

The tree-based design allows to alter the arrangement of resources in the name space easily and efficiently and even during runtime. This enables the creation of different views on the resources of underlying sensor node platform. Furthermore, these views are not fixed and can be changed during runtime like the name space can accept or remove files during runtime.

The following example in table 1 illustrates a name space as it may appear on a sensor node. The delimiter “/” is used in order to separate the directories from each other. A single “/” indicates the root directory. The example lists the directory /audiolib/ which may contain various mediated resources included in a audio library. Then, for instance, the resource /audiolib/audioVolume indicates the library function for computing the volume of an audio signal.

| Resource | Explanation |
|-----------------------|--|
| / | The / identifies the root directory. |
| /dev/ | Directory containing drivers providing access to direct resources. |
| /dev/light | File representing the driver for a light sensor. |
| /dev/audio | File representing the sampling function for a microphone. |
| /audiolib/ | Directory aggregating the functions of an audio library |
| /audiolib/audioVolume | File representing library function for computing the audio volume |
| /usr/ | Application stores arbitrary data as regular files in this directory. |
| /usr/file1 | File containing arbitrary application data; created by an application. |

Table 1. Example of a file system's name space of resources

Table 1 just shows an example. As previously explained, the developer is free to choose other expressive names and arrangement in directories in order to support his tasks. The functionality of libraries is not affected.

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 complete the uniform access model. For instance, a communication interface is accessed by the widely-used operations `send()` and `receive()`. Similarly, 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 or driver for a resource. In that way a uniform 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 |
|---|---|
| <code>size_t read</code> (<code>int fd, void* buf, size_t n</code>) | Reads <code>n</code> data bytes from the resource identified by <code>fd</code> to <code>buf</code> ; returns number of bytes or <code>-1</code> if error occurred |
| <code>size_t write</code> (<code>int fd, void* buf, size_t n</code>) | writes <code>n</code> data bytes from <code>buf</code> to the resource <code>fd</code> ; returns number of bytes or <code>-1</code> if an error occurred |
| <code>int open(char* resource_path)</code> | Returns a descriptor for the resource; <code>-1</code> if it is not valid. |
| <code>int close(int fd)</code> | Frees the descriptor of a resource; <code>-1</code> if <code>fd</code> is not valid |
| <code>int getType(int fd)</code> | Returns the type of a resource <code>fd</code> ; <code>-1</code> if <code>fd</code> is not valid. |
| <code>int mount</code> (<code>char* resource_path, int type,</code> <code>(*pFunc) read, (*pFunc) write</code>) | Creates a resource in the name space. Type and function pointers to their specific read and write operations are given. <code>-1</code> is returned if the <code>resource_path</code> already exists. |
| <code>int umount(char* resource_path)</code> | Removes a resource; <code>-1</code> 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[12]. 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 were 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. 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 | Indicates directories, such as <code>/dev/</code> or <code>/usr/</code> . |
| regular file | Each file containing arbitrary data is considered to be a regular file. |
| special files | These types are defined by a developer for all other resources, e.g. each file identifying a direct resource in <code>/dev/</code> 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, special file types can be defined freely. It enables the developer to define the compatibilities between resources. However, the developer 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[13] 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 micro controller 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 hierarchical name space by serializing a tree structure into a name space table (figure 3). The name space table points to the different device tables that are integrated within a single address space but may have different access methods.

The simplest case is the mount table, which represents the user-mounted resources. Every resource is associated with pointers to a read() and a write() method as well as a type. Another case is the storage table implemented directly by the page based flash memory component. Here the read() and write() methods are shared between all resources. Having static read() and write() functions for storage is important to ensure identically access semantics beyond restarting or even reprogramming of the system.

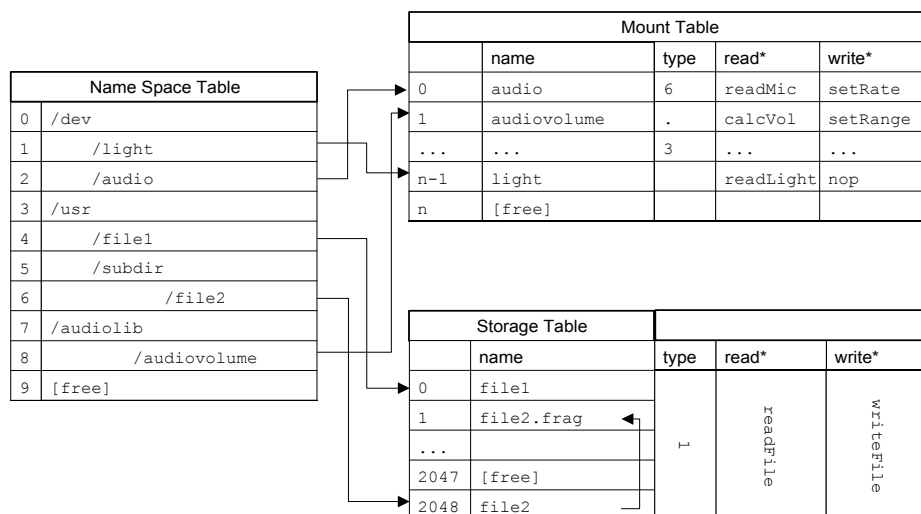


Fig. 3: File system tables of the ParticleFS

Only the mount table is kept in RAM. Its size can be adapted to the needs of the system. Pointers to non-persistent resources are removed from the name space on restart and must be remounted as their function pointers might have changed. When a file is opened first the name space is searched with the path. If the path is found a file descriptor is associated with the read() and write() methods of the resource and a state is initialized. Calling read or write on the file descriptor will result in calling the associated implementation via the function pointer. Read and write operations may to continuously alter the state associated with the file descriptor to reflect the semantics of sequential access

4.2 Functionality of ParticleFS

During the boot up of the hardware when the ParticleFS starts it mounts several resources. The file system mounts the known sensors in the /dev/ directory. 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. the audio sensor). Figure 4 (left) 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 (figure 4: 1.). 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 (figure 4: 2.) and starts sampling the microphone using the MPU's analog-digital-converter (figure 4: 3.). The sampled values are then returned in the given buffer from the read() operation.

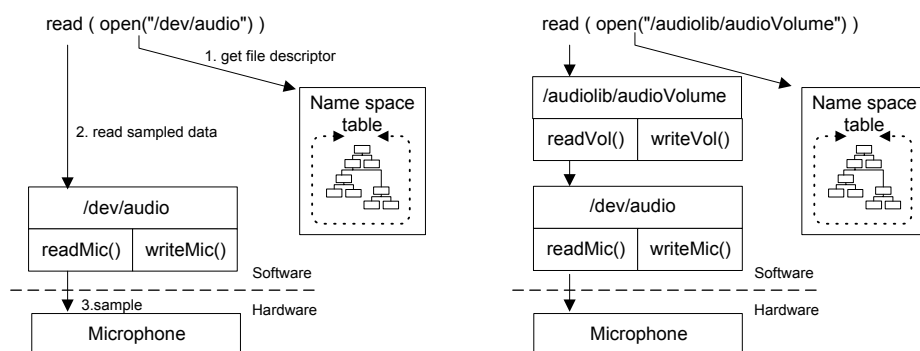


Fig. 4. Reading a sensor directly (left) and via a mediated resource (right)

Access a mediated resource (e.g. the audio volume). In figure 4 (right) the file system's read() operation will call the specific read() function of /audiolib/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/audio like in the previous scenario, but n times. The resource descriptor of /dev/audio can be re-used minimizing the effort to find the entry in the name space 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 /audiolib/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

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 re-used for further uses. As a result, 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' PIC18F6720 MPU the delay for the lookup and the additional call is below 20 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. In table 4 we present the exact overhead for resource access via the file system. The result is then compared against a regular library call for a resource access. The numbers relate to compiled C code and can be improved since the code can still be further optimized.

| Operation | Cycles | PIC18F6720 |
|-------------------------------------|------------------|-------------------------------|
| Table look up function | 15 cycles | 3 μ s |
| Dereferencing state | 4 cycles | 0.8 μ s |
| Passing Parameters | 10 cycles | 2 μ s |
| Function pointer call | 26 cycles | 5.2 μ s |
| Accessing Parameters | 10 cycles | 2 μ s |
| Writing the buffer | 15 cycles | 3 μ s |
| Returning | 13 cycles | 2.6 μ s |
| Overhead of file system read | 93 cycles | 18.6 μs |
| Overhead of simple Library call | 26 cycles | 5.2 μ s |
| Relative overhead | 67 cycles | 13.4 μs |

Table 4. Call overhead of file system sensor read.

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. Synchronous messaging avoids the necessity of handling message buffers implicitly within the system. Furthermore in a system where task isolation is no issue calls provide an implicit dispatching mechanism at no additional costs.

Both read() and write() operations are character oriented. We found, that explicitly passing arrays of bytes between resources is the most generic way to deal with their diversity. Mediated resources can transform those raw data to structured ones and can further work with them. Types are associated with resources in order to ensure the semantic meaning during these transformations. They enable mediated resources and client applications to check for compatibility. Arbitrary integer based typing systems can be implemented.

The name space table can hold up to 129 entries including directories in one 264-byte memory page. If more resources are needed, a name space table can be split into many sub-tables. To increase the locality of changes to those tables, however, an internal fragmentation of up to 50% has to be taken into account. The gained write locality decreases update times and increases the lifetime of persistent memory.

The tree structure can by default be 7 levels deep and can address multiple storage and mount tables with up to 8192 elements. This is needed to directly address files stored on arbitrary pages of a flash memory. For user-mounted resources file name

length is only limited by available memory. For persistent files it must fit 256 bytes.

Persistent storage is implemented via sequential append-only files on the Flash memory component. Multiple files can be opened simultaneously for reading and writing. Maximum throughput and minimal RAM consumption is achieved when continuously writing a single file by using both hardware buffers of the external flash memory. Multiple writers, however, may lead to a maximum of fragmentation because of mutual displacement from the Flash’s internal RAM buffers. In order to keep the interface simple in the current version no priority scheme is implemented, that could work against this behavior.

5 Application Cases on Smart-Its Particles

In this section we describe how to map typical tasks of an application to the file system API. We chose the example of an interactive command shell as a well-known application generic enough to make use of most features of the ParticleFS. Each command of the shell is implemented as a separate application and may therefore stand for any application executed in a non-interactive context.

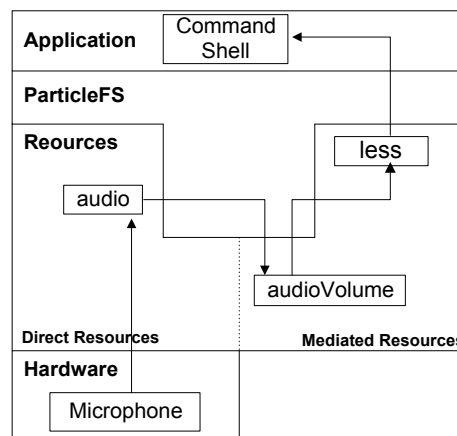


Fig. 5. Shell execution of “less /audiolib/audioVolume”

Accessing sensors. The “less /audiolib/audioVolume” command (figure 5) prints the current detected audio volume on the console. “less” does this by opening “/audiolib/audioVolume” and reading from it and writing the returned string to the console. However, if “less” was called on “/dev/audio” the command would return a failure, because a call to `getType()` will tell the application that this

resource has the type RAW and returns raw data which may contain non-printable characters.

Executable commands. To keep our shell application extensible it makes sense to represent all commands like “less” as resources themselves. We give them the type EXEC in order to reflect the executable permission of such files. The shell application calls write() on the executable file to pass the command line arguments to the command implementation, which was mounted as the associated specific write() method.

Browsing and file operations. The standard set of shell commands includes “dir”, “copy”, “delete”, “write” for listing directories, copy, deletion and writing of files. The commands allow interactively inspecting the file system and all available resources on the embedded sensor node and reflecting basic file system functionality as separate executables.

For instance, the “dir” command simply has to call read on a directory.

Directories are resources having the type DIR telling the caller that they provide a directory listing, which is a sequence of name/type tuples for every resource that is mounted directly below the directory. These tuples are parsed and printed in a human readable way.

Standard I/O. Up to now we assumed that we have standard way to return output to the user by printing it back to the interactive shell. The runtime environments for most programming languages therefore provide the notion of standard output. A program simply has to call functions like “printf” in order to write onto it. The file system enables us to support this standard set of I/O functions for any type of resource that was mounted. In our shell we preset the standard output with the resource that does the output for us before executing a new command. This might be a serial line type of output, a display or a wireless network protocol.

Pipeline resources. Selecting or reformatting data for output is normally done by specifying different selection or transformation criteria. This may be done by parsing each data set and applying these rules manually in a loop. However it is advantageous to have those transformations applied automatically by some query

processor. We can provide a generic way of stream processing on top of the file system by using pipes. Transformations are applied one after the other by modifying and passing on the byte streams that are written to the pipe head until they are finally output by the pipe end.

Each transformation is a executable command whose standard output points to the next command in the pipe. To maintain the pipe structure the shell pushes the file descriptors of all commands onto a pipe stack. Invoking the first pipe element, which was on top of this stack, starts the pipe. On invocation the new top element of the stack is pushed and set as standard output. This way each command will always pass its buffer on to the next pipe stage. For example:

```
cat /voltage/battery | case 'high' -lt 1200 'ok' -lt 1000 'low' | uniq
```

informs the caller of major changes of the battery voltage as it drops below 1200mV (“ok”) and 1000mV (“low”). Voltage information is transformed into context specific human readable data, here ‘high’, ‘ok’ or ‘low’, with the command “case” and then duplicates filtered out with “uniq” to reflect only a context change.

Sensor logging. Redirection mechanism can also easily be exploited for logging data to a file. By choosing “write /usr/logfile” as the last element of your pipe all data is collected in persistent storage and can be accessed later by “read /usr/logfile”.

Remote access. In the case of sensor node a shell mostly makes sense if you have telnet style remote access, as commonly no other input method is available. For this purpose a RPC server externalizes the file system functionality. A write RPC can then be used to invoke a shell command remotely.

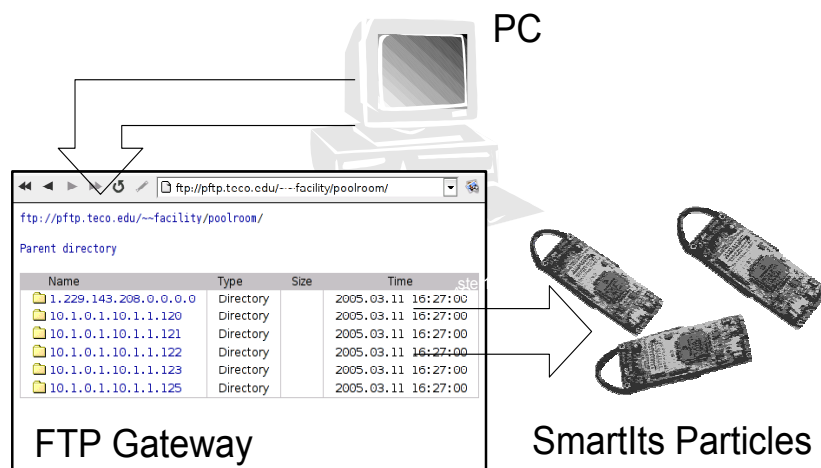


Fig. 6. Reprogramming via FTP using ParticleFS RPC

The same RPC interface can be used to access file from a backend system using web protocols. In this way resources in a sensor node network can be easily integrated in Unified Resource Identification (URI) scheme via lightweight protocol gateways. A semantic location scheme can be used to compose the name spaces of all sensor node on semantic level. An URI then looks like “ftp://pftp.teco.edu/~facility/poolroom/1.1.1.2.3.4/usr/my_file”. We use the FTP protocol for over-the-air programming of Smart-Its Particles. This way the target node a can be selected and the program image can be transferred conveniently using a web browser (figure 6). Also log file data can easily be accessed from any backend application supporting solely via the FTP protocol.

6 Related Work

UNIX operating systems and various derivatives 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 these 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[15], 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) [16] 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[17] from the ETH implement sensor node file systems. Motes follow with MatchBox[18] 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 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. Developers are provided with a compact, easy-to-understand and familiar interface. The uniform access model

and the consistent integration of all resources in one name space support developers in creating an appropriate level of abstraction which is suitable for their development tasks. 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.

Despite the use of synchronous function calls, the file system decouples application and lower system functions. This will be exploited in future work as this approach will be included in an operation system. We will also integrate a mechanism for presenting and handling events caused by hardware interrupts and event-based applications. A potential concept we are currently investigating are services. The latter are small software components which can be moved from the application layer down to the system to be executed and accessed via the file system from the application. 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 shell and remote access capabilities we have implemented an evaluation environment for testing these new file system capabilities.

8 Acknowledgements

The work presented in this paper was partially funded by the European Community through the project CoBIs (Collaborative Business Items) under contract no. 4270 and by the Ministry of Economic Affairs of the Netherlands through the BSIK project Smart Surroundings under contract no. 03060.

References

1. Kidd, C.D., Orr, R.J., Abowd, G., Atkeson, C.A., Essa, I., MacIntyre, B., Mynatt, E., Stamer, T. and Newstetter, W.: The Aware Home: A Living Laboratory for Ubiquitous Computing Research. In the Proceedings of CoBuild'99. October 1999.
2. Neustaedter, C. and Greenberg, S. The Design of a Context-Aware Home Media Space: The Video. Video Proceedings of UBICOMP 2003, Seattle, USA (2003)
3. 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)

4. Hill, J., Szewczyk, R., Woo, A., Hollar, S., Culler, D., and Pister, K.: System Architecture Directions for Networked Sensors. ASPLOS-IX, 2000.
5. Ember – Wireless Semiconductor Solutions. Available from: <http://www.ember.com/>
6. Tapia, E.M., Intille, S., Larson, K.: MITes: Wireless Portable Sensors for Studying Behavior. In Adjunct Demo Proceedings of UBIComp 2004, Nottingham, UK (2004)
7. TecO Particle Computer: <http://particle.teco.edu> [Accessed: 7/2005]
8. Kasten, O., Römer, K. Beyond Event Handlers: Programming Wireless Sensors with Attributed State Machines. The Fourth International Conference on Information Processing in Sensor Networks (IPSN), pp. 45-52, Los Angeles, USA, April 25-27, 2005
9. Decker, C., Krohn, A., Beigl, M., Zimmer T. The Particle Computer System. Proceedings of the ACM/IEEE Fourth International Conference on Information Processing in Sensor Networks (IPSN) 2005, Los Angeles, USA
10. 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)
11. 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
12. The OpenGroup. IEEE Std 1003.1, 2004 Edition. Available online: http://www.unix.org/single_unix_specification/ [Accessed: 09/2004]
13. Beigl, M., Krohn, A., Zimmer, T., Decker, C., Robinson, P.: AwareCon: Situation Aware Context Communication. Proceedings of UBIComp 2003, Oct. 12-15, Seattle, USA
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. Lamarca, A., Rodrig, M. Oasis: An Architecture for Simplified Data Management and Disconnected Operation. ARCS 2004, Augsburg, Germany
16. 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
17. BTnodes - A Distributed Environment for Prototyping Ad Hoc Networks. Available online: <http://www.btnode.ethz.ch> [Accessed: 01/2005]
18. Gay, D. The Matchbox File System. Available online: <http://webs.cs.berkeley.edu/tos/tinyos-1.x/doc/matchbox-design.pdf> [Accessed: 01/2005]