Ubiquitous Ressources Abstraction using a File System Interface on Sensor Nodes

Till Riedel, Christian Decker

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

Abstract. This paper proposes a file system as an abstraction layer for a uniform way of accessing any system resource on wireless sensor nodes. Even functions and libraries can be represented and accessed in this uniform way allowing developers a novel way to design and implement applications. A lightweight implementation on our Particle Computer platform is described and performance measurements indicate only a small overhead for a resource access.

1 Introduction

In ubiquitous computing environments tiny, networked sensor nodes are embedded in a variety of objects. Application programs on the nodes utilize many resources such as different sensors, actuators like LEDs and speakers, memory for data storage and the wireless communication interface. Other computational functionalities like algorithms are encapsulated in libraries. Various approaches were developed to ease the development of applications on wireless sensor nodes. Still, developers struggle with the diversity of different resources and their particular access methods. We propose a file system, which provides a uniform name space and access model for all resources. An implementation on our Particle computer platform (http://particle.teco.edu) proves the feasibility of an implementation for sensor nodes and performance measurements indicate only a small overhead when accessing resources through this abstraction layer.

2 Analysis and Design

It is important for the design of any translation layer is to get an idea of the entities you are interfacing. We chose a very broad aproach that covers all identifiable entities, i. e. resources, in our system view. This system view strives to be adaptable to any abstraction a system designer may choose in a system.

2.1 Resources

In general two kinds of resources can be identified. Direct resources represent the hardware on the platform, like sensors. Mediated resources abstract functional units that access other resources for further processing. Accessing either kind through a uniform interface enables us to abstract from artificial boundaries like hard- and software, or remote and local resources. The system programmer can add, modify and move device driver functionality transparently. At the same time an application may be developed without any knowledge about the underlying system. Inspired by the "/proc" file system introduced by Plan 9 [3] the Particle File System allows to represent operating system functionality along with file storage, devices and

remote resources within in a common name space. Software can access any shared functionality in a system through the stream-oriented functions “read”, ”write”, ”open” and “close”.

2.2 Name Space

The abstraction of name spaces provides an intuitive way of addressing any resources. Hierarchical name spaces prove to be an adequate means to categorize any resources independent of its internal representation. Textual natural language resource identifiers specify a path into the name space that uniquely identifies a resource. Internally this resource can be represented by a fixed size machine-readable address pointing to an arbitrary implementation. New parts of the name space can be built into the existing hierarchy via the “mount” command. This enables us to extend the system to react to dynamic settings and reflect a global context. All resource access is implicitly sensitive to such changes as “open” binds resources in an ad hoc manner.

2.3 File-based Operations

Stream primitives are a scalable and mininalistic, yet powerful abstraction for accessing resources. The interface to streams consists of a pair of POSIX-like I/O functions “read” and “write” [3] as well as a state to enforce the semantics of sequential access. We found that such semantics fit most if not all resources in our system. Streams may invoke other streams to collect or to pass on data for further processing. This stacking of streams can extend to a flexible mechanism for selection and aggregation of data. Streams accessing other streams again are mounted into the file system's name space as mediated resource.

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 close(int fd)	Frees the 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 1. Generic Access Functions of the File System

3 Implementation

Once a name is resolved to an address by calling open, a file descriptor is associated with the stream. A file descriptor table provides a cached call indirection to the driver function. The driver provides this “read” and “write” functionality as well as an initial stream state when called by “open”. Passing arbitrary stream functions to „mount“ a volatile resource may be instantiated. The device driver for special files will associate the function pointers and the resource path with a unique resource address. Calling „open“ on such the resource’s path will copy „read“ and „write“ to the „file descriptor table“ and instantiate a stream.

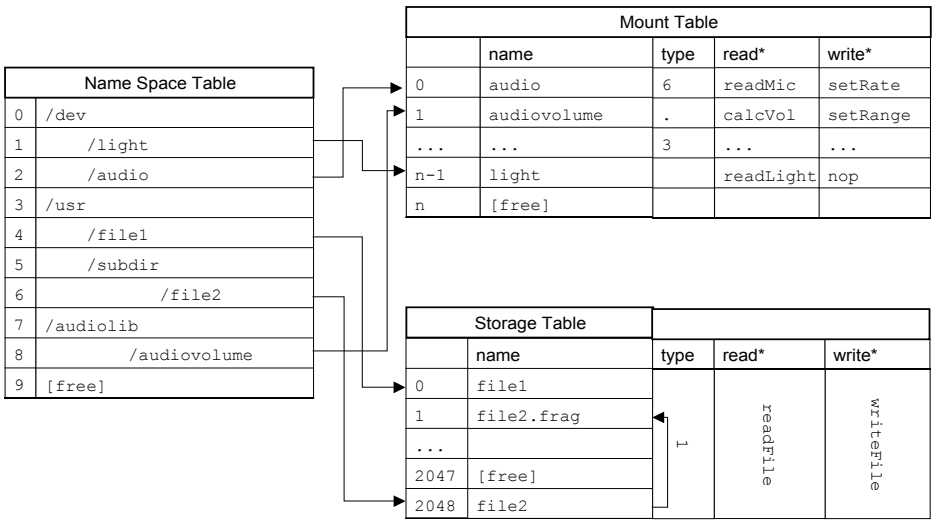

Figure 1: File System Tables of the ParticleFS

Necessary name resolution is done via a flattened tree data structure (see figure 1) . Each tree element consumes only 2 byte of memory. Encoding implicit type information into the resource address allows us to dispatch the correct driver. Being able to address 2^{13} distinct resources medium size storage can be integrated directly into the addressing scheme. The flash driver translates stream semantics to sequential page-based file structures on a 4-MBit flash chip. It supports append-only sequential files that can be used as persistent storage.

3.1 Discussion of ParticleFS

If we want to use the file system as a resource interface on the particle itself performance becomes a predominant issue. After resolving the name the actual overhead for using the file system for resource access is basically reduced to a call indirection for calling read and write functions on file descriptors via a function pointer. Additional overhead is only generated by passing length and state arguments to the driver function in. We measured a minimum of 100 processor cycles for a file system resource access on a PIC18 processor. This compares to 26 cycles when calling a static interface. Both values relate to compiler-generated code reading a one byte of instantly available sensor data. For details on the internal communication overhead refer to table 2.

Operation	**cycles**	**pic18F6720**
Table look up function	15 cycles	3 µs
Dereferencing state	4 cycles	0.8 µs
Passing Parameters	10 cycles	2 µs
Function pointer call	26 cycles	5.2 µs
Accessing Parameters	10 cycles	2 µs
Writing the buffer	15 cycles	3 µs
Returning	13 cycles	2.6 µs
Overhead of file system read	**93 cycles**	**18.6 µs**
Overhead of simple Library call	26 cycles	5.2 µs
Relative overhead	**67 cycles**	**13.2 µs**

Table 2. Call overhead of file system sensor read.

In contrast to asynchronous message passing mechanisms our synchronous approach has the advantage of being able to manage message buffers easily along with the call stack. The semantics of synchronous message passing implicate cooperative scheduling avoiding scheduling overhead. All internal communication cost between functional units is thus reduced to the call overhead mentioned above.

The sequential access to the page based flash memory translates well to the stream semantics. When writing data to the flash through a single file system stream the page allocation and buffering mechanism achieves minimal blocking times. Write overhead is only generated by the inferior erase strategy that is necessary to keep the file system consistent. Because no restrictions are made on the number of open files (unlike e.g. [4]), buffer extrusion may lead to a maximum amount of internal page fragmentation and I/O blocking times. A priorization scheme would be nessesary at this point. Reading the flash, even concurrently, needs no flash buffer or page allocation and imposes no additional overhead on the application.

4 Applications 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 an 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.

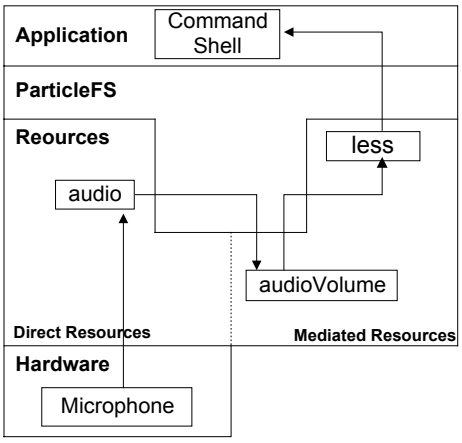

Fig. 2. Shell execution on Particle of "less /audilolib/audioVolume"

Accessing sensors. The "less /audiolib/audioVolume" command (figure 2) prints the current light sensor value on the console. "less" does this by opening "/audiolib/audioVolume" and reading from it and writing the string to the console.

Executable commands. To keep our shell application extensible it makes sense to represent all commands like "less" as resources them self. The shell application now only calls write on the command to pass the command line arguments to the implementation, which was mounted as the 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 to interactively inspect the file system and all available resources on the embedded sensor node and reflect basic file system functionality as separate executables.

The “dir” command as an example simply has to call read on a directory. Directories are simple resources, which is a sequence of names for every resource that is mounted directly below the directory. These tuples are parsed and printed.

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. To write it the program simply has to call functions like “printf”. 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 the standard output before executing a new command with the resource that does the output for us. 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 that apply to the data. This may be done by parsing each data set and applying these rules manually in a loop. However it is nice 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 outputted 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. The pipe is started by invoking the first pipe element which was on top of this stack. On invocation the new top element of the stack is pushed and set as standard output. This way each command will always passes its buffer on to the next pipe stage. For example “cat /voltage/battery| case 'high' -lt 1200 'ok' -lt 1000 'low' | uniq” can inform the caller of major changes of the battery voltage as it drops below 1200mV and 1000mV. To do so voltage information is transformed into context specific data with the command “case” and then duplicates filtered out with “uniq”.

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 the file system functionality is externalized by an RPC server. A write RPC can then be used to invoke a shell command remotely.

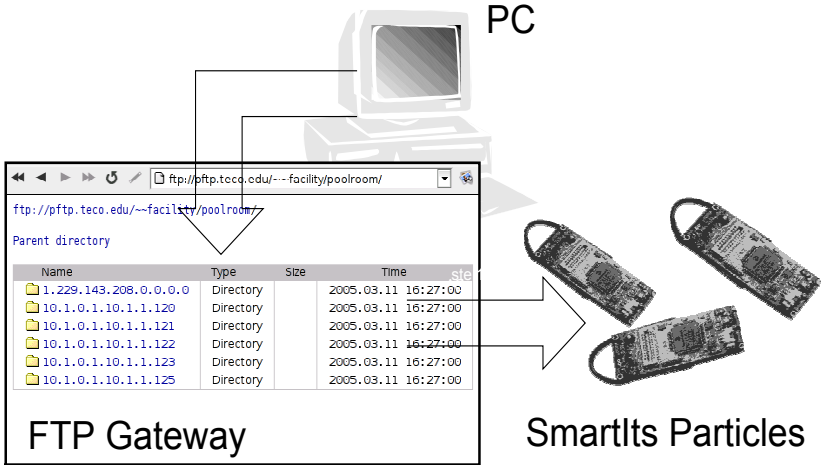

Fig. 3. Reprogramming via FTP using ParticleFS RPCs

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 light-weight 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 SmartIts Particles. This way the target node a can be selected and the program image can be transfered conveniently using a web browser (see figure 3).

5 Conclusion and Future Work

We believe that file system functionality proves to be a powerful tool to decouple layers of software and hardware. The hierarchical name space and the standardized I/O system can be used to express many high level abstractions as system resources, while keeping the performance penalty for indirectly accessing hardware small. The file system strives to reduce design time, while maximizing the design space of the Particle Computer platform.

6 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.Presotto, D., Trickey, H., Thompson, K., Winterbottom, P. and Pike, R.. The use of Name Spaces in Plan 9. Operating Systems Review 27, 1999

3.The OpenGroup. IEEE Std 1003.1, 2004 Edition. Available online: http://www.unix.org/single_unix_specification/ [Accessed: 09/2004]

4.Gay, D. The Matchbox File System. Available online: http://webs.cs.berkeley.edu/tos/tinyos-1.x/doc/matchbox-design.pdf [Accessed: 01/2005]