A File System for Resource Abstraction in Ubicomp

Till Riedel, Christian Decker, Albert Krohn, Michael Beigl, Tobias Zimmer

Telecooperation Office (TecO)/University of Karlsruhe
{riedel,cdecker,krohn,beigl,zimmer}@teco.edu

Abstract. This poster 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 prove small overhead for 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, because hiding resources behind numerous APIs only shifts those problems. 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 prove small overhead when accessing resources through this abstraction layer.

2. Uniform Resource Access

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, as the example of an audio sensor in Fig. 1 shows. 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 [2] 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”.

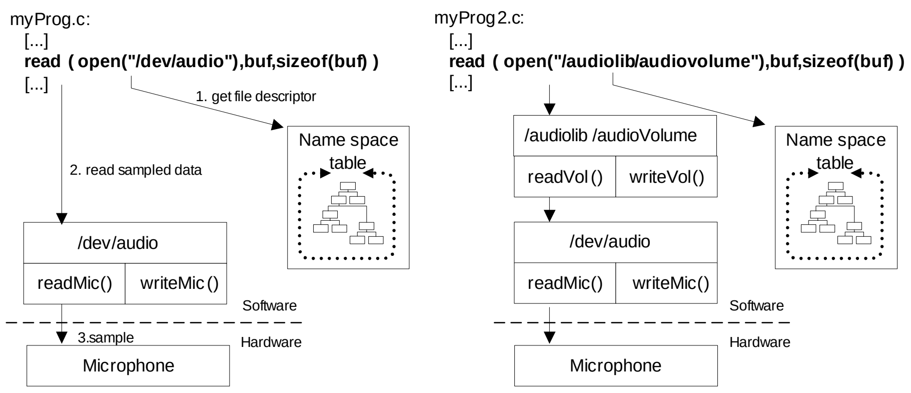

Fig. 1. Reading a sensor directly (left) and via a mediated resource (right)

Name Spaces. 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.

Streams. Stream primitives have proven to be a scalable and powerful abstraction for accessing resources. Streams consist of a pair of I/O functions “read and “write” as well as a state to enforce 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. Each stream end again is mounted into the file system's name space as a mediated resource. By adding an understanding of a standard output stream to the local context of each stream, we can construct a dynamic pipelining mechanism without mounting or changing the stream drivers. The stream data is transparently pushed towards the end of the pipeline by writing to the standard output, which is connected to the next pipeline stage.

3. Implementation and Application

Once a name is resolved to an address by calling open, a file descriptor is associated with the stream. The 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.

Necessary name resolution is done via a flattened tree data structure. 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 213 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.

The file system serves as a runtime environment for programs running on the particle platform. Especially many existing programs written in the C programming language rely on this functionality by using the IO functionality of the standard C Libraries. The functions “fprintf” or “getchar” are examples. We can provide full support for arbitrary resource access using such stream functions. A program call to “printf” e.g. will pass the data to the current standard output, which may be a communication channel via a network protocol as well as a mediated resource, that does further processing.

Although our system is primarily designed for simplifying resource access within the nodes accessing the file system via a telnet gateway allows interactive access to particle computer sensor nodes with any telnet client. By accessing the file system on the particle computer the telnet gateway runs “executable“ resources and passes the resulting output to the telnet console.

The command line “*/bin/head -n200 /voltage/log | /bin/filter -u2 -lt 1000*“ would for example select voltages below 1000mV within the first 100 2byte samples from the sensor log file. This processing is done by first opening the resources “bin/filter“ and “bin/head“ denoting each file descriptor in a “pipe stack“. The pipe’s bottom element defaults to an RF connection back to the telnet gateway. Writing the command line arguments to each file descriptor initializes the pipe. After receiving the command line the streams await further data. The standard output is set within the context of each pipe element's “write“ function by the popping the uppermost stack element. When “/bin/head“ starts outputting data it will be passed through the pipe by subsequently calling all elements of the remaining “pipe stack“.

Using the same server on the Particle via an ftp gateway allows us to program Particle nodes via a standard Internet by writing binary code via a file handle into the program memory. The ftp gateway also allows a direct mapping of the resource hierarchy on the sensor node to an URI scheme for any other resource. Ftp “get” and “put” commands can be directly translated to the file system’s “read” and “write”.

4. Performance

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. As shown in Table 1 we measured a minimum of 93 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.

Table 1. Call overhead of file system sensor read.

Operation	**Cycles**	**PIC18F6720**
Table look up for function and state	19 cycles	3.8 µs
Function pointer call	36 cycles	7.2 µs
Accessing Parameters	10 cycles	2 µs
Writing the buffer via parameter	25 cycles	5 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**	13.4 µs

The semantics of synchronous message passing implicate cooperative scheduling avoiding any 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. Reading the flash needs no allocation and imposes no additional overhead on the application.

5. Conclusion

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. 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.
2. Decker, C., Beigl, M., and Krohn, A.. A file system for system programming in ubiquitous computing, LNCS 3432, 2005.
3. Presotto, D., Trickey, H., Thompson, K., Winterbottom, P. and Pike, R.. The use of Name Spaces in Plan 9. Operating Systems Review 27, 1999