Operating System Support for Data Acquisition in Sensor Networks

Lucas Francisco Wanner, Arliones Stevert Hoeller Junior,
Augusto Born de Oliveira and Antônio Augusto Fröhlich
Laboratory for Software and Hardware Integration — Federal University of Santa Catarina
PO Box 476 – 88049-900 – Florianópolis, SC, Brazil
{lucas,arliones,augusto,guto}@lisha.ufsc.br

Abstract

Due to modularity and heterogeneity in Wireless Sensor Networks sensing devices, a sensor application developed for a given platform will seldom be portable to a different one, unless the run-time support systems on those platforms deliver mechanisms that abstract and encapsulate the sensor platform in an adequate manner.

In this article we propose a software/hardware interface that is able to abstract families of sensing devices in an uniform fashion. We define classes of sensing devices based on their finality (e.g. sensing acceleration, sensing temperature), and establish a common substrate for each class. Each individual device in a class is able to describe itself and its properties, in a similar fashion to the IEEE 1451 standard sensors transducer electronic data sheet. *A thin software layer adapts individual devices to fit the minimal requirements of its sensor class. Software-based self-description allows applications to use individual sensors' extended characteristics. We show that this strategy does not incur in excessive overhead, and presents a significant advantage with relation to solutions found in other operating system for sensor networks.*

1 Introduction

Sensors play a key role in embedded systems from smart appliances to machine monitoring and control. An electronic sensor responds to stimulus such as light, pressure or motion and generates a measurable signal, which can be interpreted in order to extract environmental information. Advances in hardware technology and design have led to reductions in size, power consumption and cost for sensing devices. Micro Electro Mechanical Systems (MEMS) technologies allow mechanical elements, sensors and electronics to be integrated on a common silicon substrate, enabling autonomous operation of sensing devices. The recent convergence of sensing and low power wireless communication technologies has enabled the advent of Wireless Sensor Networks (WSN). In a Wireless Sensor Network, several *sensor nodes*, comprised by a set of analog and digital sensors, a micro-controller, a wireless transceiver and batteries, coordinate and exchange information in order to provide a *macroscopic* vision of a given environment.

In a sensor network, application-specific requirements drive the entire hardware design, from processing capabilities to radio bandwidth and sensor modules, thus requiring the hardware to be modular. However, these requirements have led to a huge variety of hardware components, making Wireless Sensor Networks hardware not only modular, but also heterogeneous. In this scenario, a sensor application developed for a given platform will seldom be portable to a different one, unless the run-time support systems on those platforms deliver mechanisms that abstract and encapsulate the sensor platform in an adequate manner. Architectural differences aside, sensor modules (e.g. temperature, light, motion sensors) present an even wider range of variability. Sensor modules presenting the same functionality often vary in their access interface, operational characteristics and parameters.

A properly designed run-time support system could free application programmers from such architectural dependencies and promote application portability among different sensing platforms. A high-level sensing subsystem could provide transparent access to families of sensing devices. We propose a software/hardware interface that is able to abstract families of sensing devices in an uniform fashion. We define classes of sensing devices based on their finality, and establish a common substrate for each class. A thin software layer adapts individual devices to fit the minimal requirements of its sensor class.

This paper follows up on *Operating System Support for Handling Heterogeneity in Wireless Sensor Networks* [11], and explores sensing technologies and related data acquisition techniques. In the following sections, we introduce the model of *sensing abstractions* used in EPOS, an experimental *application oriented* operating system, and compare it to other relevant solutions.

2 Sensing Components in EPOS

We propose a software/hardware interface that is able to abstract families of sensing devices in an uniform fashion. We define classes of sensing devices based on their finality (e.g. sensing acceleration, sensing temperature), and establish a common substrate for each class. Each individual device in a class is able to describe itself and its properties, in a similar fashion to the IEEE 1451 standard sensors *transducer electronic data sheet* [8]. A thin software layer adapts individual devices (e.g. converts ADC readings into contextualized values, performs calibration) to fit the minimal requirements of its sensor class. Thus, a simple thermistor is exported to an application in the exact same fashion as a complex digital temperature sensor. Software-based self-description allows applications to use individual sensors' extended characteristics.

We implemented this interface for EPOS [3, 10], an application-oriented operating system. The EPOS system framework allows software components to be automatically adapted to fulfill the requirements of particular applications. Thus, an application may use a *Thermometer* abstraction, without having to address a particular temperature sensor.

2.1 EPOS Sensing Subsystem

Figure 1 presents a simplified overview of the EPOS sensing subsystem. Common methods for all sensing devices are defined by the `Sensor_Common` interface. The `sample()` method provides a single sensor, single channel reading (i.e. enables the device, waits for data to be ready, reads the sensor, disables the device, and returns readings converted into pre-determined physical units). The `enable()`, `disable()`, `data_ready()` and `get()` methods allow the operating system and applications to perform fine-grain control over sensor readings (e.g. performing sequential readings, obtaining raw sensor values). The `convert(int v)` method may be used to convert raw sensor readings (e.g. ADC or duty-cycle outputs) into scientific or engineering units. The `calibrate()` method performs a device and platform specific calibration method, which may require user interaction, depending on the sensor.

Each sensor family may specialize the `Sensor_Common` interface in order to properly abstract specific family characteristics. The `Magnetometer` family may add, for example, method for sampling and reading different axes. A `Thermistor` family, on the other hand, will probably not need to extend the basic common interface. Each family also defines a specific `Descriptor` structure, which defines specific fields for operation, accuracy, timing, calibration data and physical units.

Every sensing device implements one of the defined interfaces, and may provide specific methods for calibration, configuration, and operation. Furthermore, each sensing device fills a family-specific `Descriptor` structure with device-specific values. Default configuration parameters (e.g. frequency, gain, etc.) for each device are stored in a *configuration traits* structure.

Whenever the operating system or an application need to refer to a sensing device, they may either refer to the *specific device* (e.g. `MicaSB_Temperature`) and perform device-specific operations, or refer to the *device class* (e.g. `Temperature_Sensor`) and restrict to operations defined by that class. The *configuration traits* structure lists all the devices in a given class which are present in a given system configuration. A statically metaprogrammed realization of the device class interface aggregates all the devices listed by the *configuration traits*. This realization is concrete when all the devices in a class are of the same type, and polymorphic when different sensor types are present in a class.

2.2 Sample family of sensing devices: Accelerometers

The Accelerometer family of devices extends the basic `Sensor` interface by adding methods for reading different sensitivity axes (See figure 1). Specific devices implement the `Accelerometer` interface fully or partially (e.g. a 2-axis accelerometer will not implement methods for the `z` axis. The family also defines a Descriptor class. Specific devices may have their own `Calibration` structure and user-defined *configuration traits*.

Actual family realization is performed by a metaprogrammed wrapper. A list of devices is defined for each *machine* (e.g. `Mica2`). If all the devices in the list are of the same type, the `Accelerometer` realization will be concrete. Otherwise, it will be polymorphic. The device list also defines the order for the `Accelerometer(int unit)` constructor.

Applications may either use the `Accelerometer` realization or a specific device implementation. In the first case, the application programmer is restricted to the methods defined by the general class. Device-specific methods (e.g. configuration functions) are only available through the actual device implementation (e.g. `ADXL202`). However, the application may use the general `Accelerometer` class and use the `Descriptor` structure in order to perform device-specific operations.

3 Evaluation

In order to illustrate cases of use of our sensor abstraction strategy, we present in this section a series of application configurations and their respective costs in terms of code size and memory usage. We implemented these applications for the Mica2 [6] platform, using standard Mica Sensor Board [2] sensors. In all our configurations, a remote sensor node continually sends sensor data messages to a serial gateway connected to a PC. We used a B-MAC [9] derivate for medium access control, with a global addressing scheme and fixed-size data packets.

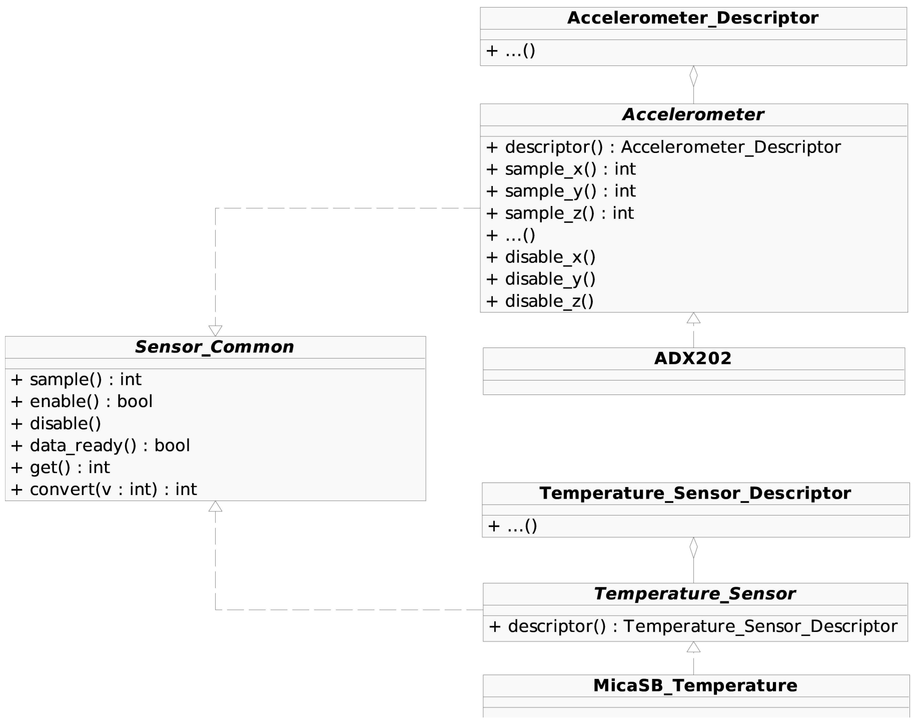

Figure 1. Overview of the EPOS Sensing Subsystem.

Basic Configuration

Figure 2 presents our basic sensing application configuration. In this application, a remote node continuously samples data from two axes of a `Null_Sensor` (a software sensor that always samples the same value) and broadcasts it to the network.

```
1  struct Msg {
2    int x,y;
3  };
4
5  int main()
6  {
7    Msg msg;
8    NIC nic;
9    Null_Sensor sensor;
10
11   while(1) {
12     msg.x = sensor.sample_x();
13     msg.y = sensor.sample_y();
14     nic.send(NIC::BROADCAST, 0,
15              &msg, sizeof(msg));
16   }
17 }
```

Figure 2. Basic Application Configuration

This application was compiled and linked with the `EPOS` operating system using GCC 4.0.2 for AVR. The resulting object code used 256 bytes of data memory (.data + .bss) and 10644 bytes of code memory (.text). These memory footprint values constitute the base cost for communication, time coordination and scheduling for EPOS in the Mica2 platform.

Actual Sensor Sampling

In order to illustrate base cost for sampling an actual sensing device, we changed the basic configuration to sample data from an ADXL202 accelerometer. The implementation for this sensor uses the device's analog output, which is sampled through an ADC Channel. In this configuration, line 9 of figure 2 is replaced with `ADXL202 sensor;`. The resulting object code for this application added 1 byte of data memory and 188 bytes of code memory to the base application configuration. This implementation deals directly with the device implementation (`ADXL202`), as opposed to the device class `Accelerometer`. This allows specific device methods to be used (e.g. specific calibration method), but hinders application portability.

Using Device Classes

In order to evaluate the overhead of addressing device classes instead of specific devices, we replaced line 9 of figure 2 with `Accelerometer sensor(0);`, and configured the device list for the Accelerometer class in the Mica2 platform with a single `ADXL202`. The resulting object code for this application added 0 bytes of data memory and 80 bytes of code memory to the previous configuration. This is due to the method calling indirection added from the general class (`Accelerometer`) to the actual device mediator (`ADXL202`).

Table 1 summarizes overhead for basic configuration (`Null_Sensor`), sensor sampling (`ADXL202`) and using device classes (`Accelerometer`).

Usage	Data Memory		Code Memory	
	Bytes	%	Bytes	%
Null_Sensor (System Base)	256	100	10644	100.0
ADXL202	257	100	10832	101.7
Accelerometer	257	100	10912	102.5

Table 1. Overhead Summary

4 Related Work

TinyOS [7] presents a three-layer design for hardware abstraction [5]. A *Hardware Presentation Layer* is positioned directly over the software/hardware interface. Components in this layer export an interface that is fully determined by the capabilities of the underlying hardware. A *Hardware Adaptation Layer* uses the raw hardware interfaces to build domain-specific components such as `Alarm` or `ADC Channel`. All components in this layer are tailored to specific hardware devices and expose specific features. Finally, a *Hardware Interface Layer* takes the platform-specific components and converts them into hardware-independent interfaces through software adaptation (either *downgrading* or emulating hardware capabilities).

Mantis OS [1] provides the application with a POSIX-like interface without incurring in significant runtime overhead (through textual substitution performed by pre-compilation tools). The application reads sensor's associated ADC values by providing the desired, platform-specific sensor name to hardware access methods such as `dev_open()` and `dev_read()`.

SOS [4] implements system with loadable kernel modules support through which an analog sensor driver binds itself to an ADC channel and registers a sensor type such as `PHOTO`. When the application requests for data from a sensor type, the kernel forwards the request to the registered driver and receives the appropriate ADC reading. The registering of drivers incurs in some memory overhead, since the operating system has to keep a table of function pointers indexed by sensor type.

Taking EPOS' approach into account, the reviewed systems fall short of providing the same level of sensor abstraction. While EPOS contextualizes values according to sensor type, all 3 sensor abstraction architectures only provide a hardware-independent interface for performing raw sensor readings. Furthermore, EPOS' self-description structure provides additional device-specific information that may be used by the application as a means to fully utilize its platform's capabilities.

5 Conclusion

We presented a high-level sensing subsystem that provides transparent access to families of sensing devices. Our software/hardware interface is able to abstract families of sensing devices in an uniform fashion, relying on classes of sensing devices defined based on their finality. Our software-based self-description mechanism allows applications to address individual sensors' specific characteristics. We showed that this strategy does not incur in excessive overhead, and presents a significant advantage with relation to solutions found in other operating system for sensor networks.

References

[1] H. Abrach, S. Bhatti, J. Carlson, H. Dai, J. Rose, A. Sheth, B. Shucker, J. Deng, and R. Han. Mantis: System support for multimodal networks of in-situ sensors. In *2nd ACM International Workshop on Wireless Sensor Networks and Applications*, pages 50 – 59, San Diego, CA, 2003.

[2] Crossbow Technology. *MTS/MDA Sensor and Data Acquisition Board User's Manual*. Crossbow Technology, Inc, San Jose, CA, April 2005.

[3] A. A. Fröhlich. *Application-Oriented Operating Systems*. GMD - Forschungszentrum Informationstechnik, Sankt Augustin, 2001.

[4] C.-C. Han, R. Kumar, R. Shea, E. Kohler, and M. Srivastava. A dynamic operating system for sensor nodes. In *MobiSys '05: Proceedings of the 3rd international conference on Mobile systems, applications, and services*, pages 163–176, New York, NY, USA, 2005. ACM Press.

[5] V. Handziski, J. Polastre, J.-H. Hauer, C. Sharp, and A. W. D. Culler. Flexible hardware abstraction for wireless sensor networks. In *Proceedings of the Second European Workshop on Wireless Sensor Networks (EWSN '05)*, Istambul, Turkey, 2005.

[6] J. Hill, M. Horton, R. Kling, and L. Krishnamurthy. The platforms enabling wireless sensor networks. *Communications of the ACM*, 47(6):41–46, 2004.

[7] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for networked sensors. In *Proceedings of the ninth international conference on Architectural support for programming languages and operating systems*, pages 93–104, Cambridge, Massachusetts, United States, 2000.

[8] K. Lee. Ieee 1451: A standard in support of smart transducer networking. In *Proceedings of the IEEE Instrumentation and Measurement Technology Conference*, pages 525–528, Baltimore, MD, 2000.

[9] J. Polastre, J. Hill, and D. Culler. Versatile low power media access for wireless sensor networks. In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 95–107, New York, NY, USA, 2004. ACM Press.

[10] F. V. Polpeta and A. A. Fröhlich. Hardware Mediators: a Portability Artifact for Component-Based Systems. In *International Conference on Embedded and Ubiquitous Computing*, volume 3207 of *Lecture Notes in Computer Science*, pages 271–280, Aizu, Japan, Aug. 2004. Springer.

[11] L. F. Wanner, A. S. H. Junior, F. V. Polpeta, and A. A. Frhlich. Operating System Support for Handling Heterogeneity in Wireless Sensor Networks. In *10th IEEE International Conference on Emerging Technologies and Factory Automation*, Catania, Italy, Sept. 2005.