DESCRIBING SENSOR DATA FOR PERVASIVE PROTOTYPING AND DEVELOPMENT

Kristof Van Laerhoven, Martin Berchtold,
and Hans-Werner Gellersen

Department of Computing, Infolab21, Lancaster University,
LA1 4WA Lancaster, United Kingdom
{kristof, hwg}@comp.lancs.ac.uk

TecO, University of Karlsruhe,
Karlsruhe , Germany
{berch}@teco.edu

Abstract
It is currently extremely challenging to bridge the gap between the hardware configuration of sensor-based systems and algorithm implementation. When an algorithm on live sensor data needs to be tried out, preferably in an embedded state, a first barrier to cross would be to get access to the sensor data in a programming environment that would offer sufficient resources for the algorithms. This poster proposes a minimalist description language of how to acquire sensor data from a wide variety of interface protocols, and briefly illustrates how it has been implemented and used in our current research thus far.

1. Introduction

Both research and production of ever more pervasive and embedded devices continue to yield implementations that break away from the traditional desktop computer paradigm. One result of this trend is that the distinction that existed earlier between hardware and software development in this area is fading; no longer do microcontrollers need to be written in carefully prepared assembly language by the engineers that built the hardware, no longer is software assumed to be affected by a single user's input alone.

Various built-in sensors can potentially influence components in the entire system, and previously inaccessible data from transducers are becoming effortlessly available via accepted interfaces, with off-the-shelf products. This paper will cover the protocols that allow software to read and interact with sensor data in particular.

From our own experience, the two biggest obstacles in incorporating sensor modules to the platform where the core software is running (be it a server in the background or a smaller, more embedded machine) are the parameters of the protocol, and the implementation of the protocol. These two combined will be used later to characterize a given sensor unit.

Protocol Implementation. One of the more universal protocols that is still in use nowadays to transfer data from sensor modules to a central unit is RS232, where the sensor is attached to the computer by serial port or USB port. Every type of development has its own way of controlling and accessing these ports: common languages used during development such as C(++), Java,

MATLAB or PERL all have their own favorite modules that can configure the right hardware port and read or write data accordingly. Especially when prototyping, using these modules can be unforgiving due to implementation specific settings, over-simplification, and/or limited capabilities. Other examples include reading data from logged files, internal sensors, or network packages.

Protocol Structure. Every sensor has different properties, and the way its data should be interpreted is therefore likely to be different as well. The structure that dictates what type of data could be expected and in what format this data is transferred, can sometimes be equally tough to get hold of.

We introduce a general approach to facilitate dealing with configurations for the acquisition of sensor data, describing how it is interfaced, in a way that aims to be as efficient and clear as possible. The implementation also *abstracts* the sensor data, as it strips away any protocol-specific elements in the remainder of the development and prototyping phase.

2. Describing Sensor Data and Interface Configurations

The way most algorithms handle sensor data is, by-and-large, identical: information from a sensor module is represented by one or more sample vectors, of which each component represents a particular sensor reading at a particular time. The three sample vectors (23,78), (22,70) and (24, 75) might for instance be readings at three different times from a sensor module that contains a thermometer and humidity sensor. Other information such as the units (degrees Celsius and percent, for those three examples), the timing of the reading, the location of the sensor module, calibration parameters, etc., are most often expected to be prior knowledge, but could also be represented as sensor data on their own if really necessary.

The format in which most embedded sensor units provide their data is generally not standard at all; although several proposals are disseminated [1,2], most manufacturers simply construct their own protocol and provide device drivers or proprietary software to process the data. Also, the method of communication varies a lot: from the use of serial or USB ports, to stand-alone units that send UDP packages over Ethernet. We also include recorded sensor data and internal sensors in this model: most phones, PDA's, and laptops come with a range of sensors that can be used in other applications (one or more light sensors for regulating display brightness, accelerometers for detecting falls, or microphones for example).

2.1. XML Descriptions for Sensor Data

To describe these possible configurations, and express how sensor data can be extracted from any given sensor unit, we set up an XML format that can easily be parsed in systems with limited resources (with little memory or processing power like PDAs or phones). Although tools have been written to create and edit these files in a graphical user interface, they can also easily be created manually, and generally occupy less than a kilobyte. As can be seen from the examples in Figure 1, its syntax is fairly straightforward and human readable. It can also be opened up in any browser for inspection, especially since support exists for creating these files with their Document Type Definition (DTD) section, which can then automatically be used for validation.

```
- <input id="xbow adxl202 evaluation board">
  - <rs232 port="/dev/ttyS0" baudrate="38400" buffsize="4" databits="8" stopbits="1" parity="no">
      <poll wait="1500" command="G"/>
    - <packet>
        <channel id="0" name="AccX" bits="16" sign="unsigned" format="integer"/>
        <channel id="1" name="AccY" bits="16" sign="unsigned" format="integer"/>
      </packet>
    </rs232>
    <inputcolumn id="0" channel="0" name="AccX" bits="14" sign="unsigned" format="integer"/>
    <inputcolumn id="1" channel="1" name="AccY" bits="14" sign="unsigned" format="integer"/>
  </input>

- <input id="internal iPAQ light sensor, Familiar">
  - <proc filename="/proc/hal/light-sensor" timeout="100" mode="ascii">
    - <packet>
        <channel id="0" name="ipaqLight" bits="8" sign="unsigned" format="integer"/>
      </packet>
    </proc>
    <inputcolumn id="0" channel="0" name="light" bits="8" sign="unsigned" format="integer"/>
  </input>

- <input id="udp from DrDAQ client">
  - <udp port="9910" timeout="100" mode="ascii">
    - <packet>
        <channel id="0" name="light" bits="8" sign="unsigned" format="integer"/>
        <channel id="1" name="temperature" bits="8" sign="unsigned" format="integer"/>
        <channel id="2" name="humidity" bits="8"/>
        <channel id="3" name="soundlevel" bits="8"/>
      </packet>
    </udp>
    <inputcolumn id="0" channel="1" name="thermo" bits="8" sign="unsigned" format="integer"/>
    <inputcolumn id="1" channel="0" name="light" bits="8" sign="unsigned" format="integer"/>
  </input>
```

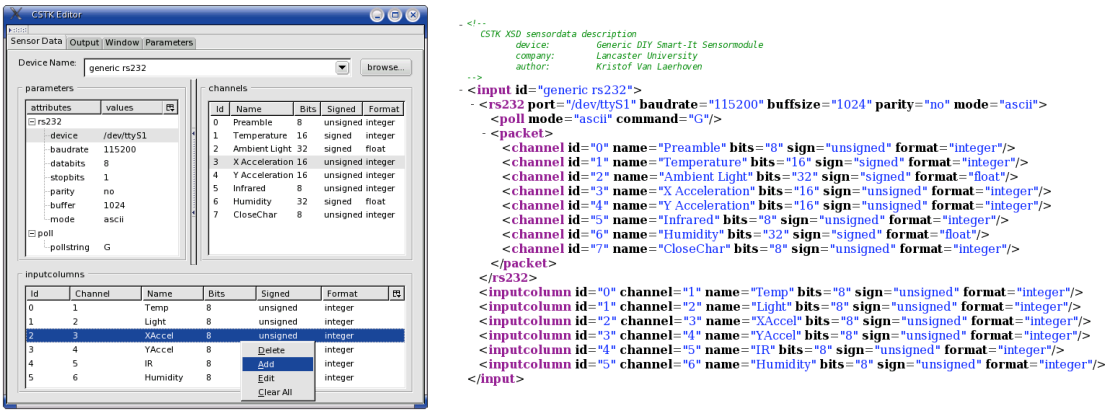
Figure 1. Some examples of the XML format used for expressing how and from where sensor data needs to be acquired.

The layout of the input section contains three distinct sections: one that describes the protocol and protocol settings (such as rs232, udp, proc, or logfile, and their parameters), a second (as a list of 'channels') that describes the protocol structure (what sensor data to expect, in which order), and a third (a list of 'inputcolumns') describing what data to retain for further processing. The first two remain almost completely identical for every sensor device, whereas the list of 'inputcolumns' is often application specific.

2.2. Generating and Editing the Descriptions

An application example is shown in Figure 2 next to its XML source: a graphical tool was implemented in XUL [3], which allows it to be run on any platform via the Mozilla, Netscape, or Firefox browsers, by loading it as a webpage or installing it locally as a plug-in. By parsing the XML into this tool, a more intuitive representation is established, where the user can quickly generate and copy information about sensors and the protocols. Note that the three sections from the XML description are again emphasised.

This is exploited additionally by linking the tool to a central database of common sensor device descriptions, so that the section for these sensor units can be retrieved straight away (depicted in the three steps of Figure 3). On-line XML descriptions can be downloaded and inserted in the current XML file via the *browse* button, allowing a fast way to acquire the data without having to enter the descriptions manually.




```
<!--
    CSTK XSD sensordata description
            device:         Generic DIY Smart-It Sensormodule
            company:        Lancaster University
            author:         Kristof Van Laerhoven
-->
<input id="generic rs232">
  <rs232 port="/dev/ttyS1" baudrate="115200" buffsize="1024" parity="no" mode="ascii">
    <poll mode="ascii" command="G"/>
    <packet>
      <channel id="0" name="Preamble" bits="8" sign="unsigned" format="integer"/>
      <channel id="1" name="Temperature" bits="16" sign="signed" format="integer"/>
      <channel id="2" name="Ambient Light" bits="32" sign="signed" format="float"/>
      <channel id="3" name="X Acceleration" bits="16" sign="unsigned" format="integer"/>
      <channel id="4" name="Y Acceleration" bits="16" sign="unsigned" format="integer"/>
      <channel id="5" name="Infrared" bits="8" sign="unsigned" format="integer"/>
      <channel id="6" name="Humidity" bits="32" sign="signed" format="float"/>
      <channel id="7" name="CloseChar" bits="8" sign="unsigned" format="integer"/>
    </packet>
  </rs232>
  <inputcolumn id="0" channel="1" name="Temp" bits="8" sign="unsigned" format="integer"/>
  <inputcolumn id="1" channel="2" name="Light" bits="8" sign="unsigned" format="integer"/>
  <inputcolumn id="2" channel="3" name="XAccel" bits="8" sign="unsigned" format="integer"/>
  <inputcolumn id="3" channel="4" name="YAccel" bits="8" sign="unsigned" format="integer"/>
  <inputcolumn id="4" channel="5" name="IR" bits="8" sign="unsigned" format="integer"/>
  <inputcolumn id="5" channel="6" name="Humidity" bits="8" sign="unsigned" format="integer"/>
</input>
```

Figure 2. A graphical representation of the sensor data, next to its XML representation.

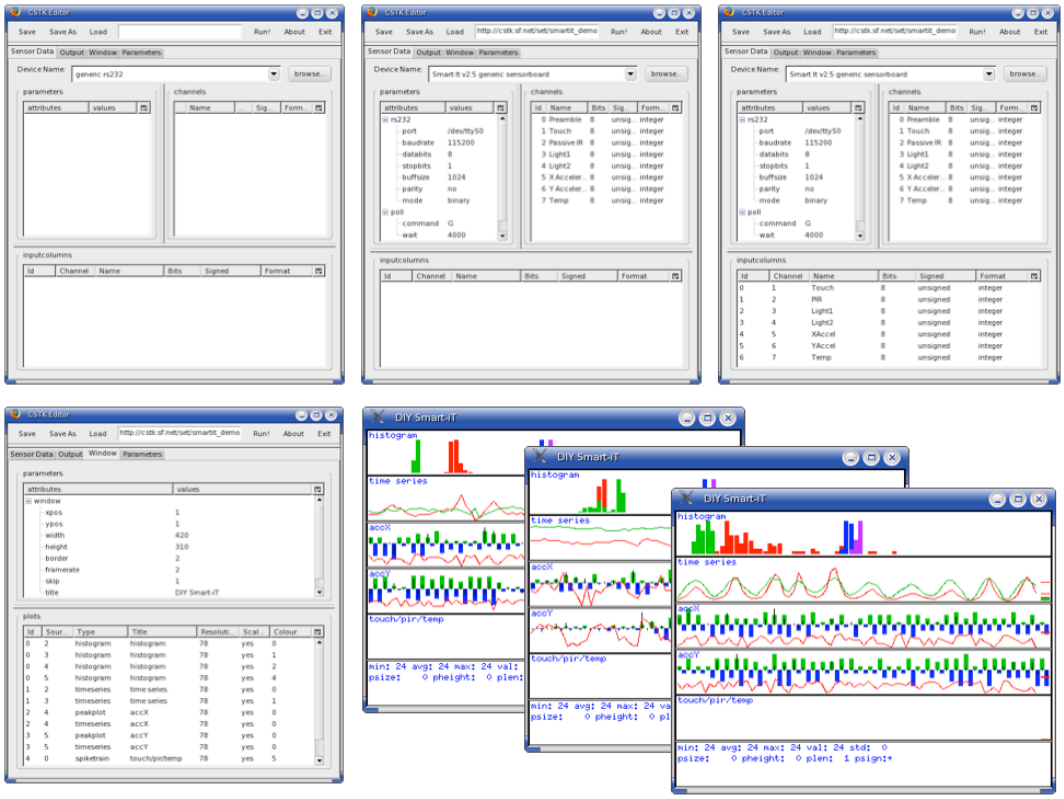


Figure 3. Reading and modifying the XML description: The top row shows how sensor data is described by downloading the device-specific XML section into the application (second), and specifying what sensors will be used (third). The lower row gives an example of how the rest of the XML file can describe tool-specific parameters (here: window attributes and plotting routines).

3. The Supporting Code

The description format that is proposed in this poster is merely an interface for dealing with sensor data in a uniform way. These XML descriptions are used, however, by a large library of code that acquires, processes, visualizes, clusters and classifies sensor data.

An extensible yet minimal parser has been written in C++, providing support for reading the XML descriptions from the most common modalities, such as serial ports, UDP packages, process files, and logged files. On top of that, supporting classes are also provided to actually open, read, and write to these, according to what has been parsed. Only POSIX compliant code is used, and it has been tested on various Linux distributions, Mac OS X, and MS-Windows (the latter via cygwin).

The code is organised in such a way that the parser encapsulates the true nature of the sensor data: after reading the settings from the XML file, it automatically opens the corresponding port or device, and gives back a vector in an abstract form as described in the beginning of section 2. We feel this is a valuable methodology as it allows the programmer to concentrate on the sensor data in the algorithm's perspective rather than in hardware configuration terms.

The whole architecture has proven to be adequately fast on small, embedded systems, with the XML files being generated on a desktop and used as a command line argument on the target system. Starting a visualization of incoming sensor data for instance, such as shown in Figure 3, requires launching the tool with the XML file, e.g.: '*rtplot light.xml*'.

All code is open source and can be downloaded from http://cstk.sf.net.

4. Acknowledgements

This work was carried out with the support of the CommonSense project, funded by the UK's Engineering and Physical Sciences Research Council and the UbiMon project, funded by the Department of Trade and Industry's NextWave Initiative.

5. References

[1] BOTTS, M. SensorML website: http://vast.uah.edu/SensorML/

[2] NMEA: http://www.kh-gps.de/nmea.faq

[3] XUL: http://www.mozilla.org/projects/xul/