Active Data Logger

In the project[1] „Active Data Logger" we investigate the design of embedded data acquisition systems when focusing on mobility, efficiency and coverage of as many use cases as possible. Mobile data acquisition systems are needed in many fields of research and productive environments that require mobility, and they have different requirements than non-mobile data acquisition systems. After all, data collection and analysis is a very important task, as it provides a basis for gaining knowledge and taking decisions based on the results. The feasibility of a universal mobile data acquisition system is demonstrated with a proof-of-concept design and implementation which covers multiple use cases.

Matthias Krebs, Christoph Stamm | matthias.krebs@fhnw.ch

Mobile data acquisition is an integral part of today's research, development and productive processes that take place in a mobile environment. Collecting data from different sources is important in order to perform tasks such as verifying a hypothesis or collecting statistical data.

Imagine a cycle racing team as an example. For the team staff, it is important to know technical and medical data of the cyclists, such as position, speed, cadence and heartbeat rate in real time. The problem is that during a race the team's cyclists can be spread over a large area, therefore, the team's supporting car cannot always be close to the cyclists. *Mobility* is one key factor in this case. In order to get data from the cyclists, the team needs a data acquisition system which is small, light and also *energy-efficient*, because the bicycles should carry as little extra weight as possible. The data acquisition system also has to be capable of communicating with the supporting car or the headquarters in real time, because the staff needs to know changes of data immediately. Because the supporting car can be several kilometers away from the cyclist, a wireless connection is required. Remote access to the data acquisition device might also be considered in case it has to be reconfigured or its status has to be checked.

In order to collect the required data, a data acquisition system that fits the use case is needed. Choosing the best product depends on the requirements of the use case, the data to be collected and, if predetermined, the sensors that are used for measurements. Many commercial data acquisition systems available on the market are tailored to a specific use case and do not perform well when trying to use them for a different purpose. These products have a well-defined set of features and supported input sources, which allows them to perform very well within the boundaries of their designated use case. However, this limited feature set can be a disadvantage if the product should be used in a different context.

In contrast to data acquisition systems intended for specific use cases, universal data acquisition systems are on the market as well. While the former often provide specific inputs for analog or digital sensors, depending on the intended purpose, the latter generally feature a number of simple analog or digital inputs. The reason for this is that analog sensors just need an analog voltage to be measured and digital inputs can be used for binary measurements such as limit switches or photoelectric barriers. The result is a universal data acquisition system that can be used with virtually any analog sensor and any binary output. However, the situation becomes different as soon as advanced digital sensors are taken into consideration. Digital sensors that feature digital communication interfaces such as SPI or I^{2}C are a lot more complex to access than analog sensors or digital inputs. They do not only require specific interfaces, but also complex communication protocols. This makes using digital sensors much more difficult to use in universal data acquisition systems.

Choosing a suitable communication technology is also important for the versatility, as the data acquisition system requires an internet connection to communicate with distant remote stations. Wired or wireless LAN could be integrated easily but requires local infrastructure. GSM-based networking is available in most places, but has limited bandwidth, operational costs for each device and makes remote access difficult due to private IPs generally being used. Such a data acquisition system cannot be accessed remotely unless a relay or push services like SMS are used.

This article describes parts of a project that has been conducted at the Institute of Mobile and Distributed Systems, in cooperation with the Institute of Micro-Electronics and the Institute of Aerosol and Sensor Technologies [GW10], which

1 Parts of this project have been financially supported by Scintilla AG and Förderverein Fachhochschule Nordwestschweiz Solothurn FVFS.

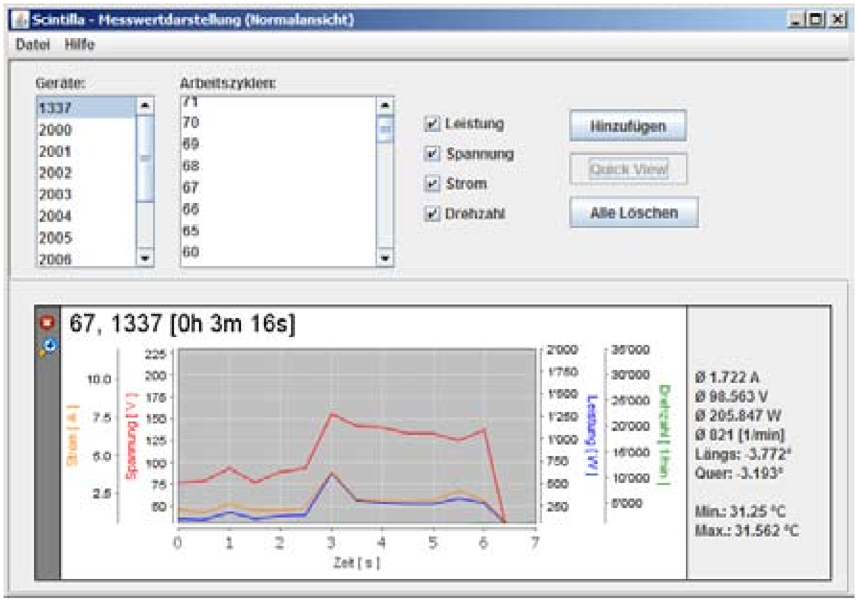

Figure 1: The modified data analysis application

both focus on embedded technology. Our goal is to create a basic framework for universal data acquisition systems that are capable of handling digital sensors and are usable in a mobile context. We take a look at a functional prototype implementing the design of a mobile universal data acquisition system that incorporates digital sensors and mobile communication technologies. The functional prototype includes software as well as embedded hardware components. We only take a brief look at the hardware components and focus on the software components instead.

Use Cases

In order to create a prototype, we need at least one but better several concrete use cases the design will be tested against. Two use cases are covered in this project, demonstrating the flexibility of the design. The first use case is the integration of sensors into portable power tools, which allows the manufacturer to analyze their usage profile. This part of the project is carried out in cooperation with Scintilla AG, a sub-organization of Bosch, and is a successor to the project [SC09]. In the project of 2009, the primary focus was to create the data analysis application *Scintilla Messwertdarstellung*, a Java-based desktop application that accesses a data collection database and displays data captured from power tools graphically. This application, as seen in Figure 1, is also used as part of our new functional prototype and is improved in the process.

The second use case is MiniDISC [MDIS], a device being developed at the Institute of Aerosol and Sensor Technologies, which is actually a mobile data acquisition device, because it is portable and measures fine dust particles. Both use cases share the same data transfer protocol [GW10].

System Design

The design of the data acquisition system is modular, so it can be adapted to different use cases more easily, with as little need to modify components as possible.

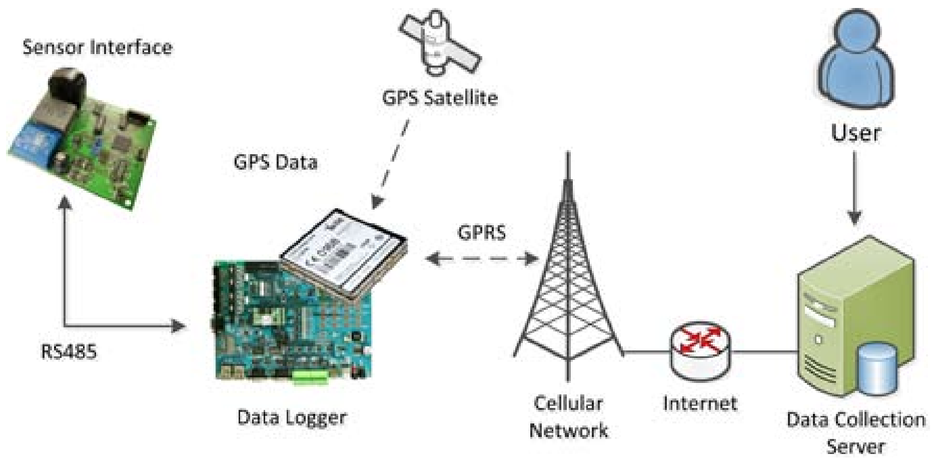

Figure 2: Data Acquisition System Design

The data acquisition system is divided into four main components:

1. a *sensor interface*, which contains sensors suitable for the designated use case, and a serial communication interface (RS485);
2. a *data logger device*, which is an embedded computer, has a local storage (e.g. SD card) and a communication module (GSM) attached, captures data from the sensor interface and transfers the data over the internet;
3. a *data collection server*, a software application that receives data from different data logger devices and stores the data inside a database;
4. a *data analysis application*, a software application that is used to access the database and evaluate the captured data. The design of this application is entirely dependent on the specific use case.

The components, particularly the sensor interface and data logger device, do not necessarily have to be separate. Depending on the actual use case, it could make sense to combine them in order to reduce the complexity or create a more compact design. We keep the sensor interface and the data logger separate due to our power tool use case. The reason is that the functional prototype we develop is simply too big to be attached to a portable power tool without any serious impact on usability.

Interfacing with Sensors

When designing a universal data acquisition system, support for many different digital sensors requires a standardized interface between the sensor interface and the data logger device. The sensors themselves might have different physical interfaces such as SPI or I^2C. This is why we use a small Atmel AVR microcontroller [AAVR] that controls the sensors and implements a serial communication protocol common to all sensor interfaces. The sensor interfaces are physically connected through an RS485 bus interface, which allows a single master (data logger device) and multiple slaves (sensor interfaces).

Using a microcontroller is an advantage, because only little hardware development is necessary, most of the functionality is provided through software. The AVR microcontroller is programmed in C, as the GCC toolchain and an Eclipse plugin are freely available. Configuration values such as data capture intervals are stored inside the AVR's EEPROM, this allows the sensor interface to be configured at runtime. The actual sensors are queried internally by the sensor interface, and the data is preprocessed and cached in memory. When sensor data is queried through the RS485 bus, the cached data is returned. This takes some load off the data logger, as the sensor data is already preprocessed on the sensor interface.

Data Logger Device

The data logger device is the core component of the data acquisition system, as it captures data from the sensor interfaces and either stores it locally or transmits it to a network server. Developing a hardware platform is a time-consuming task, this is why we have evaluated different existing platforms based on ARM processors, as these are widely used in embedded systems due to their flexibility and low power consumption. We have evaluated development boards from Roundsolutions (Aarlogic), Olimex (CS-E9302) and Quickembed (S3C2440SBC). However, they are pure developer board and could not easily be integrated in a custom design. The platform we have finally chosen is the ARM-based Eddy CPU module by SystemBase [SYSB], as it is compact, provides the necessary connectivity and runs a customizable embedded Linux operating system. The module itself can be detached from the developer board and mounted on a custom board. Applications can be programmed in C using the GCC toolchain, and after adding the C++ standard library (libstdc++) to the system, even C++ can be used for development. We choose C++, because it allows an object-oriented approach resulting in a more structured software design, while still being efficient enough to run on an embedded system.

The embedded data logger software is divided into separate threads, as we need concurrent data retrieval and network connectivity. One thread fetches data from the sensor interfaces in defined intervals. The interval duration depends on how often new data is required. To address the problem of a potential lack of a network connection, captured data is stored temporarily and transmitted as soon as a connection is available again. We implement this functionality by pushing the data, which is to be transmitted over the network, into a memory-based FIFO. If no connection is available, the data stays there until the connection is available. This is done in the same thread as the data capture. A second thread, which handles the network connection, takes data from the FIFO when a network connection is available. Of course, the FIFO is limited due to system memory limitations.

Future implementation could also introduce a persistent local storage such as an SD card. This would allow data to be saved until a network connection is available, even if the device is turned off.

GPRS Data Transfer

A mobile data acquisition system that transmits its acquired data automatically and independent of its location needs a mobile network connection. We choose a GPRS connection through the cellular network, because unlike wired or wireless LAN connections, it is available almost everywhere on Earth.

There are different embedded GSM modules on the market. They primarily differ in size and functionality. Our hardware of choice is a Telit GM862-GPS embedded module, since it provides all the GSM/GPRS functionality in a single package and can be controlled through a serial RS232 interface. It also includes a GPS receiver that allows recording the location of the device. Other products need a separate SIM slot or GPS receiver. To find out whether a GPRS connection provides enough bandwidth for the captured data being transmitted, tests using different packet sizes are conducted[2].

Packet size (bytes)	Average upload (bps)
32	1718
64	2850
128	4017
256	5362
512	6374
1024	9500

Table 1: GPRS upload performance in relation to packet size

These test results in Table 1 show that the data transfer is more efficient with increasing packet size. This is primarily due to the overhead produced as a smaller packet size requires more send commands to be executed to transmit the same amount of data. Additionally, the upload bandwidth is limited to 9600 bps when GPRS is used. Fortunately, this is enough for our power tool scenario, as we only capture data when the power tool is switched on.

Acquired sensor data is transmitted using a custom data transfer protocol that uses a TCP/IP connection. We intend to use a custom protocol, because we need as little communication overhead as possible in order to keep communication costs low. The protocol we use has been developed by Michael Glettig and Benjamin Wyrsch during a student project [GW10]. The protocol design is kept as simple as possible, so it can be implemented in embedded systems that have only little resources, and to minimize the amount of data being transmitted in order to keep communication costs low.

The basic concept of the communication protocol is the distinction of use cases. Each protocol configuration stands for one use case, which is identified by a *protocol ID*. Additionally, a *revision ID* allows different versions of the configuration on the same server. The reason different versions

2 The GM862-GPS module and a Swisscom SIM card are used for all GPRS tests. Results may vary when other providers are used.

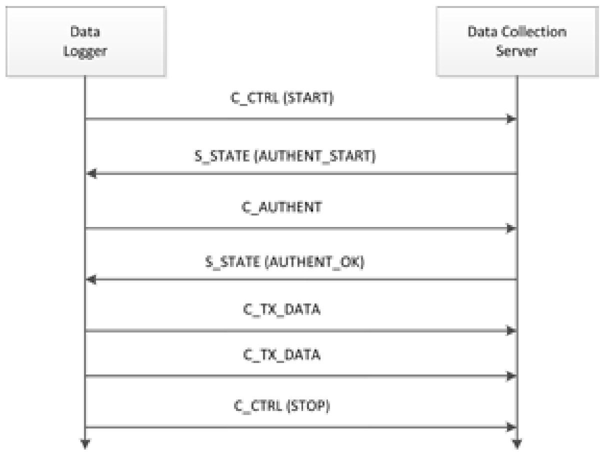

Figure 3: A typical conversation between a data logger and the data collection server

need to be distinguishable is that a relational database is used to store the data, and the database layout may change with protocol revisions, which could result in conflicts with existing data inside the database tables. Sensor data is divided into data *channels* and *sub-channels*, where each data channel consists of a group of one or more sub-channels. A sub-channel is a single data value of one of the common numeric data types, such as integers of different size and floating point numbers. We use this structure because there are data values that are related to each other, for example, a GPS position always contains a latitude and a longitude value. When data is transmitted, it is up to the data logger which channels are transmitted in a single packet, because the channels can have different data capture intervals, and the amount of data transmitted should be kept to a minimum. Nevertheless, when a channel is to be transmitted, all its sub-channels must be transmitted along with it because their data values are related. Each channel has a unique ID within a specific protocol configuration.

Data logger devices are identified through a unique device ID, which is a 64-bit integer. Additionally, each device uses a password to authenticate itself before transmitting data. This provides basic security against unsolicited data collection.

The communication protocol uses binary data packets of the following structure:

- a header (5 bytes) containing the packet ID (1 byte), the data header size (2 bytes) and the data payload size (2 bytes)
- an optional data header of variable size which describes the structure of the data payload
- a data payload of maximal 65’535 bytes

There are four types of communication packets:

- The *C_CTRL* packet is sent to the server to control the connection. It contains a single payload byte 00h (start a connection) or FFh (stop connection). The start packet is acknowledged by a status response. The stop packet is not

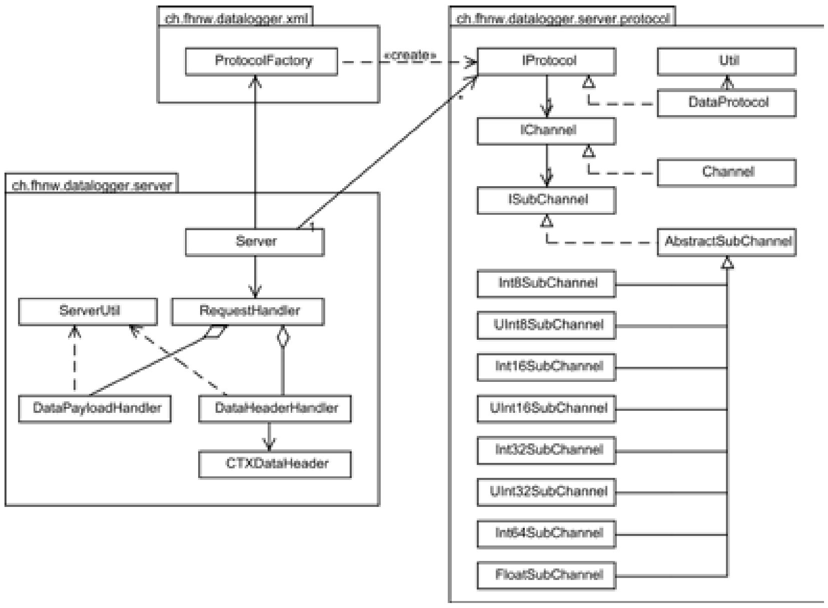

Figure 4: The software design of the data collection server

acknowledged as the TCP socket is closed immediately by the server.

- The *C_AUTHENT* packet is sent after initiating a connection and a server response that notifies the client that the server is ready for authentication. It is acknowledged by a status response telling the client that either the authentication was successful, or it failed either due to wrong credentials or an unsupported protocol ID. There is a 1-byte data header specifying the device password length. The payload contains the device ID, the password as well as the protocol ID and revision (1 byte each). The password supports ASCII format only, which circumvents problems with Unicode handling.
- The *C_TX_DATA* packet contains channel data and is sent by the client when the connection is established and the client has been authenticated. Data packets are streamed by the client without receiving an acknowledgement. The data header contains an array of the channel IDs whose data is present in the payload and the size of each channel payload. The channel order is arbitrary, but it must be the same as the order of the channels in the payload. The order of the sub-channel values is fixed.
- The *S_STATE* is a server status response that acknowledges *C_CTRL* and *C_AUTHENT*.

An example of a typical communication sequence is shown in Figure 3. The communication protocol is described in more detail in [GW10] and [MK11].

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<protocol>
  <title>Example data acquisition protocol</title>
  <protocol_properties>
    <id>0x12</id>
    <revision>0x00</revision>
    <description>Example Data Acquisition</description>
  </protocol_properties>
  <data_channels>
    <channel>
      <id>1</id>
      <name>Temperature</name>
      <preserve>true</preserve>
      <subchannel>
        <name>T1</name>
        <bytes>4</bytes>
        <format>float</format>
      </subchannel>
      <subchannel>
        <name>T2</name>
        <bytes>4</bytes>
        <format>float</format>
      </subchannel>
    </channel>
  </data_channels>
</protocol>
```

Listing 1: An example of a protocol configuration

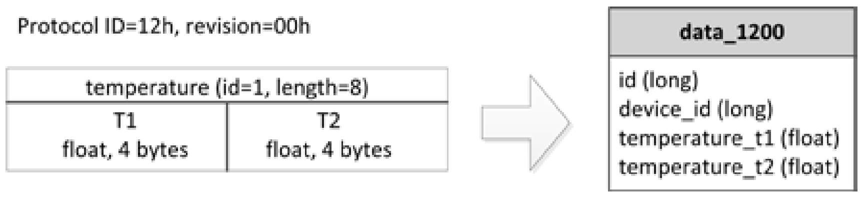

Figure 5: An example of the mapping of a data channel to the database table

Data Collection

Collection of captured data is performed on a centralized system. We implement this system in the form of a Java based stand-alone server software that uses a TCP/IP socket to listen for incoming data. Incoming data is processed and persistently stored in a PostgreSQL database.

The data collection server is designed as a truly use-case-independent tool, which means it can be configured to match any data acquisition scenario without the need of source code modification. The class diagram of the server software is shown in Figure 4. The main server class contains a list of protocol configurations that are loaded on start-up and the communication handler classes that process the client connections in a multi-threaded fashion. For each connection, a new thread is created. This method limits the scalability of the system, but this is no issue here, as the primary goal is to create a basic functional prototype that demonstrates the concepts.

Support for different use cases without code modification is provided through protocol configurations that are loaded on start-up. These protocol configurations are described in XML format, which makes them easy to edit by hand and parse. An example with a channel containing two sub-channels is provided in Listing 1.

For each protocol configuration, a separate file is placed in the *protocols* subdirectory of the application directory. All protocol configurations in this directory are loaded by the *ProtocolFactory* upon start-up.

The *ProtocolFactory* possesses the method *IProtocol loadFromXml(String filename)*, which parses a protocol configuration and creates a protocol object structure according to the class diagram in Figure 4. Each protocol object contains a list of channel objects, which each contain a list of sub-channel objects. Their class depends on the data type. Supported types are all signed and unsigned integer types between 8 and 64 bits, as well as 32-bit floating point numbers. Each protocol object also contains a channel map that allows to find channels by ID quickly, and it provides a method *setupSQL()*, which generates appropriate CREATE and INSERT statements according to the protocol object structure.

The database layout is quite simple. There is a global device table called *devices*, which bears the two fields *device_id* and *password*. Besides the device table, only one table is required per protocol configuration, this is the data table. Data channels are mapped to the database as shown in the example in Figure 5. The data table is named *data_<protocolID><protocolRevision>*, this provides a unique name for each protocol ID and revision. The table of the example in Listing 1 would be named *data_1200*. Common to each data table is a unique *id* field and a *device_id* field that references the device table. The remaining data fields depend on the specific protocol configuration. Their naming convention is *<channelname>_<subchannelname>*, and the order of the fields is equal to the order of channel and subchannel objects inside the protocol object. There is one caveat with certain programming languages and database systems, which also concerns Java and PostgreSQL: They do not support unsigned integer data types. This is why unsigned integers are represented by a signed integer of twice the width in the database, for example, an 8-bit unsigned integer is represented as a 16-bit signed integer. For this reason, there is no unsigned 64-bit integer, as a signed 64-bit integer is the largest integer type available to JDBC, unless database-specific big integers are used. Integers are interpreted this way so no additional conversion is necessary and the original number range is available.

When a data logger opens a connection to the data collection server, the appropriate protocol configuration is chosen through its ID and revision. As soon as a C_TX_DATA packet is received, the payload is processed in the method *void processData()* of the protocol object. The binary payload data is deserialized according to the channel IDs and their length of each channel in bytes. Deserialization of the sub-channel values is done by just converting from the binary little-endian representation of integer and floating point numbers. By default, Java uses big-endian, but this behavior can be changed by using a little-endian *ByteBuffer*. If there is a mismatch between the transmitted channel length and the length defined in the protocol configuration, the current channel is skipped and processing is synchronized to the beginning of the next channel.

As mentioned before, not all channels have to be transmitted in every C_TX_DATA packet. This requires a well-defined behavior to handle omitted channels. In this implementation, each channel has a *preserve* flag, which is specific to the protocol configuration on the server side. If the flag is true, the sub-channel values that have been received last are kept in memory and inserted into the database, even if no data for this channel is received. If the flag is false, or if no data has ever been received since server start-up, NULL is inserted for all sub-channel values of the omitted channel. This shows explicitly that no data has been received from this channel. There is no other way than to insert NULL or a valid value, as we always have to insert a full row into the database.

Conclusion

The functional prototype described in this article demonstrates that the fundamental concepts of the design are a suitable basis for a working data acquisition system. Parts of the design, mainly the GPRS communication protocol and an implementation of the data collection server, are already actively used in a larger scale test of the MiniDISC device [MDIS]. The functional prototype we have developed is capable of capturing data from different sensors and propagating the captured data to a centralized data collection server without user interaction, therefore covering the entire process chain. Furthermore, it fulfills the requirement of a location-independent and mobile system.

The sensor interfacing and data collection concepts are already proven to be flexible and adaptable to different use cases. The configurability of the data logger device, however, still needs to be improved in order to allow an adaptation without changing the embedded software on the device. Also, the general stability, performance and reliability of the system are not optimal yet.

Future Research

The research conducted during the project *Active Data Logger* has given us insight into the topic of data acquisition and general embedded hardware and software development. There is still a lot of potential for improvements that could turn the universal data acquisition system into a product that could be usable in a productive environment.

The Institute of Mobile and Distributed Systems is planning to acquire future projects with industrial partners in order to continue research on this topic. These projects could be based directly on the current research, or focus on completely new goals, while still benefiting from the results of this research.

References

[AAVR] Atmel AVR: http://www.atmel.com/avr

[GW10] Feinstaub-Messnetzwerk. Michael Glettig, Benjamin Wyrsch, Fachbericht HS2010, Institut für Aerosol- und Sensortechnik, Fachhochschule Nordwestschweiz, 2011.

[MDIS] MiniDISC Feinstaubmessgerät: http://fierz.ch/minidisc/

[MK11] Active Data Logger. Matthias Krebs, Master Thesis, FS2011, Institut für Mobile und Verteilte Systeme, Fachhochschule Nordwestschweiz, 2011. http://webapache.imvs.technik.fhnw.ch/~christoph.stamm/reports/P9_2011_ActiveDataLogger.pdf

[SC09] IP209 – Projekt Scintilla. Michael Bodmer, Dominic Feer, Harun Gezici, Roland Kappeler, Adrian Roth, Michael Schneider, Thomas Weber, Institut für Mobile und Verteilte Systeme, Fachhochschule Nordwestschweiz, 2009.

[SYSB] Eddy CPU Module by SystemBase: http://www.embeddedmodule.com