

Active Data Logger

Matthias Krebs

February 1, 2011

Master Thesis

Advisors:

Prof. Dr. Christoph Stamm

Prof. Hans Buchmann

Institute of Mobile and Distributed Systems
School of Engineering
University of Applied Sciences
Northwestern Switzerland



University of Applied Sciences Northwestern Switzerland
School of Engineering

Contents

| | |
|--|-----------|
| I Introduction | 8 |
| 1 Motivation | 9 |
| 2 Task | 10 |
| II Concepts | 11 |
| 3 Mobile Data Acquisition | 12 |
| 4 Usage Scenario | 12 |
| 5 System Concept | 12 |
| III Interfacing with Sensors | 14 |
| 6 Sensor Interfacing Concept | 15 |
| 7 Hardware Interface | 17 |
| 8 Sensor Communication Protocol | 20 |
| 9 Implementation of the Sensor Interface | 24 |
| IV Data Transmission using GPRS | 52 |
| 10 GPRS Concept | 53 |
| 11 GPRS Hardware | 56 |
| 12 GPRS Device Access | 62 |
| V Centralized Data Collection | 71 |
| 13 Data Collection Concept | 72 |
| 14 Database Structure | 74 |
| 15 Data Collection Transmission Protocol | 76 |

| | |
|---|------------|
| 16 Data Collection Server | 82 |
| 17 Evaluation of the Data | 86 |
| VI Data Logger | 87 |
| 18 Data Logger Hardware | 88 |
| 19 Data Logger Embedded Software | 91 |
| 20 Data Logger Miniaturization | 100 |
| VII Experiments and Results | 102 |
| 21 Testing Hardware | 103 |
| 22 Sensor Interface Experiments | 104 |
| 23 GPRS Experiments | 107 |
| 24 Data Logger Experiments | 109 |
| 25 Conclusion | 112 |
| VIII Future Work | 113 |
| 26 Sensor Interface Improvements and Fixes | 114 |
| 27 Implementation of Miniaturization | 117 |
| 28 Data Evaluation Software | 118 |
| 29 Remote Configuration of the Data Logger | 119 |
| 30 Persistent Local Data Storage | 121 |
| IX Appendix | 122 |
| A Versatile Sensor Protocol Identifiers | 123 |
| B Example XML Protocol Specification | 124 |

C Glossary 127

References 128

Abstract

Mobile data acquisition is an important topic in many situations, as data collection and analysis can be the foundation of gaining knowledge and taking decisions based on the results. The requirements for a data acquisition system change with every scenario it is used in, thus flexibility is a relevant factor.

In this thesis, we discuss what the requirements are for a flexible data acquisition system, what components are part of it and how it can be designed in general. More specific concepts depict the design of a concrete data acquisition system that can be used in different situations.

A proof-of-concept solution based on a real scenario demonstrates how such a data acquisition system can work in reality, based on experiments that have been carried out. It also pinpoints potential and existing problems that arise during the design and development process.

Declaration

I hereby declare that this master thesis has been written exclusively by me, and that no sources other than those specified have been used. Parts of the system that have not been developed exclusively by myself are labeled accordingly.

Windisch, February 1, 2011

Matthias Krebs

Acknowledgement

This thesis has been written at the Institute of Mobile and Distributed Systems, which is part of the University of Applied Sciences Northwestern Switzerland in Windisch, Argovia.

I would like to take the opportunity of expressing my gratitude to the people who supported me during the 18 months of this master studies project, especially my advisors Prof. Dr. Christoph Stamm and Prof. Hans Buchmann.

I would also like to thank Dr. Martin Fierz, Benjamin Wyrsh, Michael Glettig and Michèle Lochiger for the good cooperative work concerning the development of a common data collection system.

Part I

Introduction

1 Motivation

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 the 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 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 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.

In order to collect the requested 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. 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. Depending on the product, the sensors either have a digital interface or just output an analog voltage to be measured.

In addition to specialized data acquisition systems, there are also universal data acquisition systems available on the market. In general, such a system has an arbitrary number of analog and digital inputs that measure an analog input voltage or a single digital signal each.

Such a data acquisition system can be connected to any kind of analog voltage source as well as simple digital sources, such as limit switches. Therefore, the system is very flexible as far as different use cases are concerned.

However, it cannot be used with sophisticated sensors that require a digital interface like SPI, I²C or RS485, because it requires a complex communication protocol to be handled.

When working with different scenarios that cannot be covered by a universal data acquisition system, a specialized system is needed for each scenario. The reason is that it is difficult to find a single system that covers all requirements in terms of functionality, size and connectivity.

In our scenario mentioned before, a mobile data acquisition system designed specifically for bicycles is needed, which transmits the collected data every minute using GPRS for example. In a different scenario, let us imagine a weather station, the data is sent every hour by using SMS instead, and the software to store and evaluate the data is completely different.

This can make the task more difficult, as we are dealing with different kinds of hardware, and not all systems use the same method to access the collected data. Some data acquisition systems are connected to a computer via RS232, USB or Ethernet, or they store the data on a memory card. Additionally, if the data is transmitted over a wireless connection, several possibilities such as WLAN, GPRS or SMS exist.

What we need is a flexible and cost-effective data acquisition system that can be used in mobile scenarios and adapted to the requirements of various use cases, while requiring only minimal modifications. The advantage is that basically the same hard- and software can be used in different scenarios, the only differences being the sensors used and the evaluation of the collected data.

2 Task

The main task of this thesis is the design and development of a mobile data acquisition system that is flexible enough to satisfy different mobile use cases, without the need of developing a separate solution for each use case.

The task is divided into various sub-tasks. First, a flexible yet simple sensor communication interface has to be developed, so that all necessary sensor modules can be accessed in a consistent manner. This also makes it easier to integrate new sensors into the system. The second sub-task is the development of an actual data logger device, which is called an active data logger because it actively accesses the sensors, determines what data to save and independently forwards the data to a server. The data logger incorporates an embedded computer system, the sensor interface and a communication interface to access the server through the internet. The goal of this sub-task is a functional prototype that can be miniaturized later. The third sub-task is the server part of the system. The data acquired by the data logger has to be collected and stored persistently in a database, where it can be accessed for evaluation.

Core aspects of the system are compactness, low power consumption, performance, reliability, and production costs as well as running costs. In order to achieve a good balance between the aforementioned aspects, different technologies have to be examined.

Finally, all components of the data acquisition system should be tested and integrated into the process chain, demonstrating that they are consistent, thus allowing a correct data flow starting at the sensors and ending in the database.

Part II

Concepts

This part of the thesis provides an overview of the fundamental concepts used in the project.

3 Mobile Data Acquisition

Data acquisition can take place in different environments, which leads to different requirements and constraints.

Fixed installations of data acquisition systems, for example a process monitoring system inside a production facility, are often designed to work in an industrial environment. They are built from robust components and use industrial-grade communication interfaces, because wireless technologies are more vulnerable to noise in general. Real-time data evaluation is not very difficult, provided that there is a permanent network connection available on the data acquisition system.

In contrast, a mobile data acquisition system focuses on portability. The components should be lightweight, so they can be carried around easily, and energy-efficient if they need to be battery-powered.

Mobile data acquisition system often need to work independently of operators, for example when they are used in inhospitable areas or in places far away from civilization. In this case, it is advantageous if they transmit their data automatically, eliminating the need of extracting the data on-site.

4 Usage Scenario

Generally speaking, we are developing a data acquisition system which depends as little on its usage scenario as possible. However, we are also following a real scenario, which we use for an implementation of the concepts, and upon which the experiments are based on. The scenario then provides us with a proof-of-concept.

Our scenario is the acquisition and analysis of power tool utilization profiles. Professional portable power tools, such as power drills or jigsaws, are widely used on construction sites. They are used to work with different materials, like wood, metal or concrete. Depending on the material being worked on and the manner the power tool is used, different kinds of strain are put on the tool.

This makes it interesting for the tool manufacturers to know more about the usage of their products, so they can identify the most common utilization profiles and therefore optimize the tool development and production processes.

Scintilla AG, a subcontractor of the Bosch group, manufacturer of a wide range of power tools, would like to improve their product development in order to adapt new products to the customer's needs. To achieve that, *Scintilla AG* would like to integrate a compact data logger into a batch of portable power tools, which are distributed among selected customers. This data logger is supposed to record data like temperature and power consumption whenever the tool is used. Since the tools are portable and thus change their location, collecting the data on-site is tedious. Therefore, the data logger should transmit the recorded data to a centralized database on a regular basis.

5 System Concept

When creating a concept for a mobile data acquisition system, it is important to identify the different components the system is built of.

Each data acquisition system needs *data sources*, in many cases these are sensors measuring certain quantities. A *data logger* is a device that captures data from the data sources, does preprocessing in certain cases and stores it. This is minimal basis for a data acquisition system. An example would be a simple GPS data logger that periodically stores the current GPS position on a flash memory card.

Depending on the concrete case, the sensors could be built directly into the data logger, or there could

be external connectors. For example, an environment monitor would most likely use external sensors, so the sensors can be put in an optimal place for measurements, or if data from different places is to be acquired.

A more sophisticated concept, especially when thinking about mobile data acquisition systems, is a data logger that does not only store data locally, but also has the capability of *communication*, which means that it can independently transfer its data through a communication system, or do so when remotely requested by an operator.

A first rough approach for an independent mobile data acquisition system concept is shown in Figure 1. This concept consists of a data logger that connects one or more sensors over a digital interface, which could be a network-based bus like in [LE] or [CGS] or an industrial bus technology, and is also capable of communicating with a server over a network connection. The data is stored on the server and can be accessed by a client over the network.

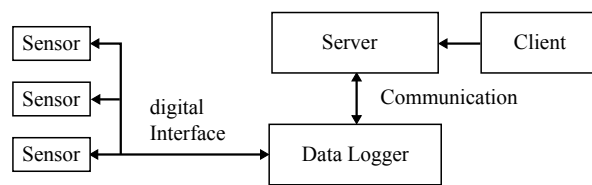


Figure 1: A simple concept of a mobile data acquisition system

The concept which is the base for our data acquisition system is a more fine-granular concept. Like in the simple approach, multiple sensors are connected to the data logger over a digital bus interface. An interface that only supports end-to-end connections is not ideal, as it would only allow one sensor or one sensor module with multiple sensors to be connected.

In our concept, the data logger is thought of as a more modular device. As shown in Figure 2, the data logger component can be seen as core system that connects to the sensors, it can possess a local storage medium to save data and it can be equipped with a communication module that provides a network connection to communicate with a server. This communication module can be built-in or even be exchangeable in case a different networking technology must be used. To be independent, the data logger actively transfers its data to the server, as it is also described by [WL09]. The server collects data from different data logger devices and stores the data in a centralized database. This provides a single access point to an operator who would like to work with the data, in contrast to collecting the data manually from different devices.

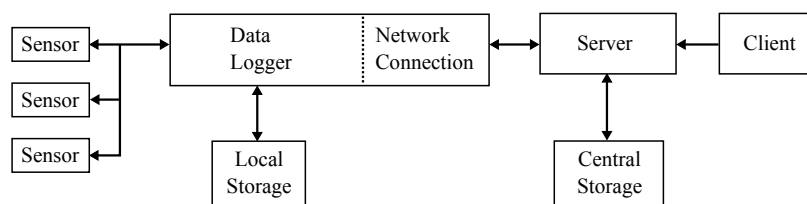


Figure 2: A more detailed view of the data acquisition system's components

Part III

Interfacing with Sensors

This part describes by which means sensors are accessed by a data logger. This includes documentation of the physical connection, the communication protocol and an implementation of the concept.

6 Sensor Interfacing Concept

There are thousands of different integrated sensors available on the market, and particularly for common uses cases like measuring temperature, there is huge choice of products among several manufacturers.

One of the most relevant differences between sensors is the interface through which they can be accessed. Analog sensors for example don't have a real interface, they just output an analog voltage in a specific range, which must be measured using an analog-digital converter (ADC). However, many integrated sensors, especially sensor ICs, feature a digital interface like SPI or I²C. More complex sensor modules also have RS232, RS485 or CAN bus interfaces. In general, those digital interfaces are not compatible among each other, and each technology has its advantages and disadvantages.

Besides the hardware interface, there is also a difference in communication. Most sensors, even if they provide the same hardware interface, have different communication protocols or command sets. This makes their implementation in a flexible data acquisition system more complicated.

When designing a flexible data acquisition system, it is advantageous to provide a single standardized interface between the data logger and the sensors, thus supporting as many different sensors as possible.

Table 1 shows a rough overview of the pros and contras of some of the most widely used interfaces.

In our concept, we use a single hardware interface as an intermediate layer between the data logger and the actual sensors. This means that we are not accessing the sensors directly, instead we access a microcontroller, which implements both our standardized hardware interface as well as the actual sensors connected to it. This combination of microcontroller and sensors forms a sensor module.

Of course, this is more complicated than just directly connecting the sensors to the data logger, but it gives us the best flexibility in terms of sensor choice.

| Interface | Pro | Contra |
|------------------|---|---|
| SPI | <ul style="list-style-type: none"> • simple implementation • multiple slaves possible | <ul style="list-style-type: none"> • needs separate chip-select line for each slave • no error detection possible • precise timing necessary • only suitable for short distances (less than 1m) |
| I ² C | <ul style="list-style-type: none"> • bus interface • multi-master capable • collision and error detection | <ul style="list-style-type: none"> • only suitable for short distances (between 1 and 10m) |
| RS232 | <ul style="list-style-type: none"> • simple point-to-point interface • suitable for long distances (more than 10m) | <ul style="list-style-type: none"> • no bus interface • asynchronous transceivers, needs precise clocking • different voltage level variants |
| RS485 | <ul style="list-style-type: none"> • simple serial bus interface, only two lines necessary • differential signaling • multiple slaves possible • suitable for long distances (more than 10m) | <ul style="list-style-type: none"> • only physical interface is standardized • no collision detection |
| CAN | <ul style="list-style-type: none"> • serial bus interface, only two lines necessary • differential signaling • multi-master capable • advanced collision and error detection • suitable for long distances (more than 10m) | <ul style="list-style-type: none"> • complex implementation |

Table 1: Pros and cons of common digital interfaces

7 Hardware Interface

Considering the advantages and disadvantages of different interfaces mentioned in Table 1, we choose the RS485 interface for sensor interfacing. This interface is relatively simple and therefore can easily be implemented on a small microcontroller. It also gives us great flexibility in terms of wire length. A further advantage is the differential signaling, which makes the interface less vulnerable to noise than if the signals were relative to ground. Noise usually affects all data lines, so it is effectively cancelled out when differential signals are used.

Since the RS485 standard only specifies the physical interface, we can use it to add bus capability to the RS232 interfaces which already exists on almost all microcontrollers and embedded systems. In this case, RS485 is used as the physical connection, whereas data is transferred using the RS232 protocol used by the serial ports on the microcontroller or embedded system.

7.1 How RS485 Works

RS485 is a serial bus interface, therefore it can be used to create a network of devices. There are three standard variants of the physical interface:

- **2-wire, half-duplex, no echo**
One pair of differential bus lines are used for bi-directional communication. Therefore, only half-duplex communication is possible. The transceiver can only either transmit or receive at a given point of time.
- **2-wire, half-duplex, echo**
One pair of differential bus lines are used for bi-directional communication. Therefore, only half-duplex communication is possible. The receiver is always on, thus everything that is transmitted is received again as an echo.
- **4-wire, full-duplex** Two pairs of differential bus lines are used, one for transmitting and one for receiving. Since they are independent, full-duplex communication is possible.

In our data acquisition system, a master/slave bus interface is used, with the data logger as the master device and one or more sensor modules as the slave devices. Therefore, the master communicates only with one slave at a time, so we use the *2-wire, half-duplex, no echo* variant, which is described in [MH07].

RS485 uses differential signal levels, whose *high* level is the same as the supply voltage of the transceiver IC. This is because the transceiver's driver is usually bridged to the supply. If the supply voltage is 5V, the positive bus line has a *high* level of 5V and a *low* level of 0V. These signal levels are inverted for the negative bus line. The minimal voltage difference must be at least 200mV to get a valid signal.

The RS485 standard is designed for a maximum number of 32 devices on the same bus, which is due to the load each device causes. Each device is usually terminated with a 120Ω resistor (a different value might be needed, depending on the number of devices and wire type), and pull-up resp. pull-down resistors on the bus lines in order to force a valid signal level when the bus is idle.

When the output driver of the RS485 transceiver (such as the MAX485 [M485]) is disabled, it goes into a high-impedance tri-state mode, so that the device is passive and has no influence on the bus. This should be the default behaviour, because there can only be one device transmitting at the same time. As long as the receiver is enabled, the device is still able to receive data.

For 2-wire applications, either the master or one slave can transmit at a given point of time. For 4-wire application, two devices on the bus can transmit, but only in opposite directions, since there are two pairs of bus lines.

The RS485 standard provides no means to detect collisions (multiple devices trying to transmit at the same time), unlike I²C or CAN. Theoretically, when using 2-wire connections, the echo mode could be used to detect if data

7.2 Implementation

In our data acquisition system, we use a 2-wire non-echo RS485 interface. Figure 3 shows the setup with the data logger as the master device, and the sensor module(s) as slaves. This forms a bus topology similar to the description in [L485]. Because we use a non-echo mode, the TXEN signal not only enables the transceiver, but also disables the receiver. If the receiver were enabled all the time, all data transmitted would also be received and thus would result in an echo mode.

The MAX485 [M485] is used as a transceiver IC, which includes a driver and a receiver unit that can be enabled or disabled separately, and it is capable of providing a 2-wire connection at up to 2 Mbps. The transceiver IC only acts as a level translator, the actual communication interface at the device is an RS232 UART with 5V or 3.3V signal levels. An additional digital output is needed for enabling/disabling the output driver. Embedded systems that feature a serial port with RTS/CTS signal can use the RTS output pin for this purpose.

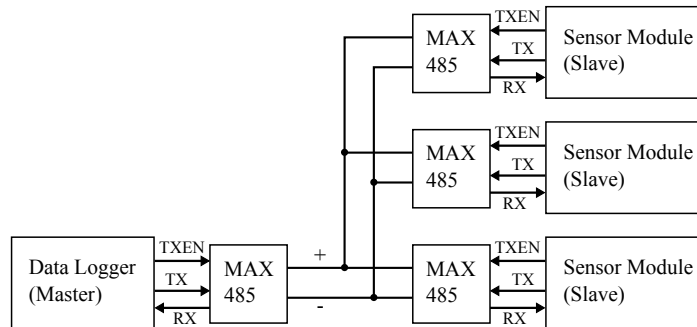


Figure 3: RS485 2-wire bus interface with a single master and multiple slaves

A timing diagram for the transmission of data is given in Figure 4. The critical part is the TXEN signal, which enables the output driver. No data must be transmitted over the TX line, which is the output on the microcontroller, before the driver is enabled. When the driver is enabled, the signals coming from the TX line are turned into two differential bus signals + and -. As soon as the transmission is finished, the driver is disabled again. The delay t_1 prevents the data from being transmitted too early, because the output driver needs a few microseconds to be enabled, and t_2 make sure the UART buffer of the TX is completely transmitted before the output driver is disabled again. Otherwise, a part of the data is not correctly put on the bus. The values for t_1 and t_2 depend on the hard- and software being used. Realtime systems or simple microcontrollers provide interrupts when a transmission over the UART has finished, other systems do not. In that case, a delay of a few milliseconds might be necessary to ensure the UART buffer is completely transmitted.

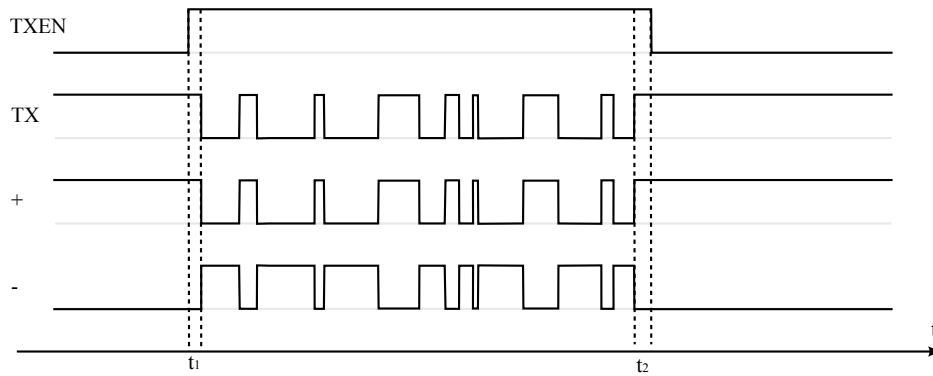


Figure 4: RS485 2-wire bus interface transmission timing diagram

8 Sensor Communication Protocol

The RS485 hardware interface described in section 7.2 provides a physical layer to transmit and receive data bytes over a serial bus interface. Because an RS232 serial port is actually used on the device side, the RS485 interface works just like an RS232 interface, but with differential signals and only half-duplex. Specifically, this means a maximal baud rate of 115 kbps, 8 data bits per frame, 1 stop bit and no parity by default. Flow control is omitted, as it does not exist on RS485.

In order to communicate with a sensor module, we need a communication protocol with the capability to read data from the sensor module, to write configuration data to the sensor module and to do basic error handling.

This communication protocol needs the following capabilities:

- Address a specific sensor module
- Address different sensors on the sensor module
- Command request/response handling
- Checksum validation of received data
- Basic error handling if a transmission error occurs
- Simple and efficient design, usable on small microcontrollers

8.1 Protocol Specification

There are already a few standardized industrial communication protocols which can be used in conjunction with RS485, such as Modbus [MB02] or Profibus [PB98]. However, the command set of Modbus is not very flexible, and both protocols are relatively complex and thus are difficult to implement on small microcontrollers.

Because we need a simple communication protocol that is flexible enough to access all kinds of sensors, we design a protocol of our own. This protocol, called *Versatile Sensor Protocol*, has the following characteristics:

- Data is transferred in frames which all have the same basic layout
- Each data frame consists of a fixed header, data payload of variable length and a checksum
- Incoming frames are buffered until a frame is complete, and then processed
- Frames received by a slave device, which are not addressed to that specific device, are discarded
- The protocol consists of command request frames sent by the master device to a slave, and command response frames, which are sent from the slave back to the master

8.1.1 Frame Layout

Each communication frame of the *Versatile Sensor Protocol* has the same basic layout, as shown in Table 2.

The frame consists of the following fields:

| Section | Header | | | | Payload | Trailer |
|---------|--------|-------------|---------|--------------|--------------|----------|
| Field | Start | Destination | Command | Payload Size | Command Data | Checksum |
| Length | 1 | 1 | 1 | 1 | 0..255 | 1 |

Table 2: Versatile Sensor Protocol Basic Frame Layout

- **Start**
A start byte representing the beginning of every frame. It is used for frame synchronization and has the arbitrary hexadecimal value A5h.
- **Destination**
This is the destination address of the frame. Command requests should have a slave device address between 01h and FEh, whereas responses should carry the master device address. For valid IDs see Appendix A.1.
- **Command**
This is the ID of the command request and response. For valid IDs see Appendix A.2.
- **Command Data**
The command data depends on the command, and whether it is a request or a response. It is of variable length, the maximal length being 255 bytes.
- **Checksum**
This is a simple XOR sum of all the previous frame bytes. This checksum must be valid in order to process the frame.

8.1.2 Checksum Validation

A checksum is computed before a frame is transmitted and after a complete frame has been received. When a frame is received, the computed checksum is compared to the checksum field which is located at the end of the frame. If the two checksums match, the frame can be processed.

The quality of the checksum depends on its computational algorithm. Our XOR checksum does an incremental bit-wise XOR on each frame byte. Therefore, the probability that an error is detected in a specific bit position is the same for all positions within the byte. However, the checksum can only detect an error in a bit position if there is an odd number of errors in the same position. If there is an even number of errors in a position, the XOR operation will cancel them out.

Furthermore, the XOR checksum can only detect errors, but not correct them. Therefore, if an error is detected, the frame must be discarded.

The reason why we use this simple checksum algorithm is that it needs the least computing performance, which is practical when it is implemented on a small microcontroller. Error correction is not so important, as the protocol frames are small, so discarded frames can be retransmitted in case an error occurs.

8.1.3 Commands

The following commands are currently supported by the *Versatile Sensor Protocol*:

Echo (Command ID 00h)

The echo command can be used to ping a specific slave device. The data payload sent with the command request is returned unmodified.

Read Sensor Value (Command ID 01h)

This command reads the current values of a given sensor.

Request:

| | |
|----------------|--------------|
| Section | Command Data |
| Field | Sensor ID |
| Length | 1 |

Table 3: Read Sensor Value Command Request

The command request takes a single byte as payload: the ID of the sensor to read from. For valid sensor IDs see Appendix [A.4](#).

Response:

| | | |
|----------------|--------------|-------------|
| Section | Command Data | |
| Field | Sensor ID | Sensor Data |
| Length | 1 | variable |

Table 4: Read Sensor Value Command Response

The response delivers the ID of the sensor and the sensor data in serialized form, which depends on the sensor type being accessed. If an error occurs, an error command response is returned instead.

Write Sensor Configuration (Command ID 02h)

This command updates the configuration of a given sensor.

Request:

| | | |
|----------------|--------------|---------------|
| Section | Command Data | |
| Field | Sensor ID | Configuration |
| Length | 1 | variable |

Table 5: Write Sensor Configuration Command Request

The command request takes the ID of the sensor to configure and the configuration data as payload. The configuration data depends on the sensor type and is of variable length.

Response:

| | |
|----------------|--------------|
| Section | Command Data |
| Field | ACK |
| Length | 1 |

Table 6: Write Sensor Configuration Command Response

The response delivers an acknowledge flag (ACK). If the value of the ACK field is not zero, the command was successful. Otherwise, the command failed. As an alternative, if an error occurs during the command processing, an error command response is returned.

Error (Command ID FFh)

This command response provides an error code in case a command request fails.

Response:

| | |
|----------------|--------------|
| Section | Command Data |
| Field | Error Code |
| Length | 1 |

Table 7: Error Command Response

The response delivers the error code of the failed command request. See Appendix [A.3](#) for a list of error codes.

9 Implementation of the Sensor Interface

In a proof-of-concept implementation of the *Scintilla* usage scenario, a sensor interface to be built into power tools is developed. This sensor interface uses a microcontroller to access four different sensors. The sensor interface itself will be accessed over an RS485 interface using the *Versatile Sensor Protocol*.

For other scenarios, a similar sensor interface based on this design can be created, possibly requiring different sensor implementations.

9.1 Measurement Categories

For a useful analysis of the power tool usage profiles, *Scintilla* would like to get the following data from the data acquisition system:

9.1.1 Working Cycle

A working cycle is defined as the timespan during which the power tool is turned on. Sensor data is only recorded during a working cycle, and not when the power tool is idle. This reduces the amount of data being collected to the data that is actually relevant for analysis.

In general, the working cycle duration is between 1 and 60 seconds, with a typical duration of 10 seconds.

9.1.2 Temperature

Power tools have strong electric motors which produce heat, especially under heavy load. To measure how much a power tool heats itself up while being used, the temperature inside the case is measured. It is measured at least at the beginning and at the end of a working cycle.

9.1.3 Current

Many power tools do not only have a fixed setting (on or off), but they can set to different speed levels, depending on the material being worked with. When a higher setting is used or the tool is under higher load, the motor draws more current. This current is to be measured twice a second during a working cycle.

9.1.4 Voltage

When a power tool has different speed levels, the motor speed is usually determined by regulating the voltage. The motor voltage is usually regulated by using pulse-width modulation (PWM) for DC motors or phase angle control for AC motors. The motor voltage is to be measured twice a seconds during a working cycle.

9.1.5 RPM

When a power tool is running idle, the motor rotates at a certain rate. This rotation rate is usually the specified RPM in the user manual. However, when under load, for example when drilling a hole using a power drill, the rotation rate decreases due to the material resistance. In order to analyze, at which

rotation rates the power tools are typically used, the rotation rate is measured twice a second during a working cycle.

9.1.6 Orientation

Some power tools, for example power drills, can be held at different angles when working. A hole can either be drilled horizontally or vertically. The orientation of the power tool is to be measured during the working cycle, so it is possible to analyzed at which angles the tools are typically used.

9.2 Microcontroller

An Atmel ATmega16 microcontroller of the AVR series is used in our implementation of the sensor interface. This microcontroller comes in a 44-pin TQFP package and has a footprint of approximately 1 cm³.

The reasons why it is chosen are that AVR's are inexpensive, energy-efficient, have a huge set of features, and they can be programmed easily using a GNU C compiler. An Atmel STK600 developer board is used during the development process.

There are also smaller AVR models which have less I/O pins and especially less flash memory. To be on the safe side, the ATmega16 is chosen. If all the required features can be provided by a smaller model, it can easily be replaced, as the program code is usually portable.

Technical details:

- 5V or 3.3V models available
- Harvard CPU architecture
- max. CPU clock 16 MHz
- 16 KB flash memory
- 1 KB RAM
- 512 bytes of EEPROM memory
- two external interrupts
- two 8-bit and one 16-bit timers or counters
- one serial port UART, 5V signal level
- one hardware SPI interface
- one hardware I²C interface
- JTAG interface

In our design, a 7.3728 MHz crystal is used for the CPU clock. This odd frequency is necessary because of the UART. Whenever the UART is used, it needs these odd frequencies to calculate the baud rate (since UARTs are asynchronous). If a straight clock frequency like 8 MHz is used, the baud rate is calculated wrong, and data transfer cannot work.

9.3 Sensors

In our design, integrated digital sensors are used wherever possible. An SPI interface is used to access them through the microcontroller.

9.3.1 Temperature

The temperature sensor used in our design is a Microchip TC77. It is a very small IC in a SOT23 package which measures temperatures from -55 to +150°C. The temperature is directly measured on the IC, there are no connectors for external temperature diodes.

The current temperature is read digitally using SPI, a 13-bit signed integer value is returned.

SPI Configuration:

- Chip Select: active-low
- Clock Polarity: positive (rising edge)
- Clock Phase: setup-sample
- Data Order: MSB-first

Reading Data:

The returned value is a signed integer value. In order to get a temperature in centigrades, this value has to be multiplied by 0.0625:

$$Temperature = Input \cdot 0.0625^{\circ}C$$

Configuration:

The TC77 has no configuration registers, it is read-only. If required, temperature offset compensation must be done in software.

9.3.2 Current and Voltage

An Analog Devices ADE7753 energy meter IC is used to measure the motor current and voltage. This highly integrated IC is designed to be used in AC power meters and is capable of measuring the most important electrical parameters. It has two analog ADC channels, one for current measure and one for voltage measure. Furthermore, the IC can calculate the active power based on the current and voltage measured.

Reading data registers and writing configuration registers is done via an SPI interface.

SPI Configuration:

- Chip Select: active-low
- Clock Polarity: positive (rising edge)
- Clock Phase: setup-sample
- Data Order: MSB-first

Reading Data:

The RMS current (the voltage across a measurement resistor respectively) is saved in a 24-bit register of unsigned integer type. At a full-scale ADC input voltage of 0.5 V the register value is 2642412. The actual voltage that is supposed to be measured is calculated by the formula

$$U_I = \frac{Input}{2642412} \cdot 0.5V$$

Like the RMS current, the RMS voltage is also saved in a 24-bit register of unsigned integer type. At a full-scale ADC input voltage of 0.5 V the register value is 1561400. The actual voltage that is supposed to be measured is calculated by the formula

$$U_V = \frac{Input}{1561400} \cdot 0.5V$$

Configuration:

The ADE7753 has many configuration registers that let the user compensate ADC offsets, ADC gain and phase shift.

9.3.3 RPM

There is no IC that can directly be used as a rotation rate sensor.

However, the Bosch GST 135 BCE is the only power tool among the tested tools that has an integrated rotary encoder. It outputs impulses when the magnets of the motor induce a small current inside a coil. These impulses are fed into a microcontroller's external interrupt input and then counted.

Tools that don't have a rotary encoder of some kind are more difficult to measure. They need an appropriate rotation rate measuring concept, which could be similar to the GST 135 BCE magnetic sensor. However, such a concept is only feasible if the sensor coil can be placed so that a current is induced when the motor is running. Optical methods could work, too, but they are more vulnerable to dust that could easily get inside the power tool's housing.

9.3.4 Orientation

The orientation is measured using a capacitive accelerometer IC. The Freescale MMA7455L is a 3-axis accelerometer that has a measure range between 2 and 8 g. It has 8- and 10-bit registers to store the acceleration values. There are two interfaces for digital access, an SPI and an I²C interface.

The tiny LGA package of the IC is a slight disadvantage for experimenting. It is barely possible to solder this IC by hand, at least very good soldering skills are required, or the IC is damaged.

It is important to know that the MMA7455L runs only on a 3.3 V supply, while the other components run on 5 V. This is an unfortunate increase in complexity, as it requires an additional voltage regulator, and, which makes it even more difficult, the SPI signals need a proper signal level conversion.

SPI Configuration:

- Chip Select: active-low
- Clock Polarity: negative (falling edge)
- Clock Phase: setup-sample
- Data Order: MSB-first

Reading Data:

The acceleration along each axis is saved in an 8- and a 10-bit register of signed integer type. At full scale, this is equal to an acceleration of 2, 4 or 8 g, depending on the selected precision. To calculate the orientation, the 2 g range is good enough, since we only measure the earth's gravity. The real acceleration of each axis is measured using the formula

$$a = Input \cdot 0.015625g$$

Configuration:

The MMA7455L has an offset register for each axis, which compensates for shifted acceleration values. Furthermore, the measurement range and operation mode can be configured.

Computing the Orientation:

The orientation can be described using two angles: The *pitch* angle describes a rotation around a horizontal axis perpendicular to the longitudinal axis, and the *roll* angle describes a rotation around the longitudinal axis of the object.

[MI03] states that the orientation can be computed using an accelerometer if the only acceleration is gravity.

It can be computed as described in [TU07], using the formulas

$$pitch = \arctan \frac{z}{x}$$

$$roll = \arctan \frac{z}{y}$$

9.4 Original Sensor Interface Design

The original sensor interface design is from an early stage of development, it is designed as a test platform for the different sensors and measurement concepts.

9.4.1 Components

The most important components besides the sensors are described here.

Power Supply

There is no integrated power supply, a 5 V DC supply must be provided externally. In this case, the 5 V supply comes from the Atmel STK600 developer board. Since the STK600 provides no 3.3 V supply, which is needed by the MMA7455L, a 5 V to 3.3 V voltage converter is used. The conversion is done by a LM3940 voltage converter, which can deliver up to 500 mA.

Current Measurement Transformer

The ADE7753 can measure the current by measuring the voltage drop over a shunt resistor connected in series to the motor current. Unfortunately, this eventually connects the sensor interface directly to phase potential, which is dangerous if there is an external connection like RS232 or RS485.

To get an isolated measurement that also does not affect the power tool operation, a current transformer is used. The Triad CST-1015 is a simple current sense transformer which has a hole in the center where the current-carrying wire is routed through. The alternating current in the wire induces a small current in the transformer coil, the current transfer ration being 1:1000. The current can then be measured by measured the voltage drop over a 33Ω resistor connected to the CST-1015. Currents up to 15 A can be measured with this setup, so that a 15 A current results in a 0.5 V voltage drop over the resistor.

A low-pass RC filter cancels out potential current spikes of frequencies higher than 5 kHz.

Voltage Measurement Transformer

In an ADE7753 example application, the AC line voltage is directly connected to the ADE7753 ADC input, there is only a 1:500 voltage divider that turns a 250 V AC line voltage into a 0.5 V AC voltage that can be measured by the ADE7753. This design eventually connects phase potential to the analog

ground plate of the ADE7753, which is also connected to the digital ground plate used by all other components. As a consequence, there is potential danger if an external connection such as RS485 exists.

This is why we use a small transformer that isolates the AC voltage from phase potential. A Block 6 V PCB transformer is chosen, because it is the smallest PCB transformer that could be ordered. It transforms a 230 V AC voltage into a 6 V AC voltage (which has a peak amplitude of 8.4 V), and therefore only needs a 1:18 voltage divider to turn it into a measurable voltage with a peak amplitude of 0.5 V.

The PCB transformer is actually a supply transformer, which is used due to a lack of an appropriate measuring transformer.

Communication

The sensor interface itself has no communication interface, because it is provided by the Atmel STK600 developer board. The only interface available is an RS232 port. In a later development phase, the RS232 port is replaced by an RS485 adapter, so the original sensor interface gets RS485 connectivity instead of RS232.

9.4.2 PCB Layout

A schematic is shown in Figure 5. The resulting PCB is merely used as an add-on board for the Atmel STK600 AVR developer board.

9.4.3 Design Issues

The original sensor interface has quite a few design issues, which is the reason a new sensor interface design is developed.

TC77 SPI Interface

The TC77 temperature sensor claims to have an SPI interface, but in fact it is actually a different interface called 3-wire. SPI and 3-wire interfaces are quite similar, except that the 3-wire interface has a bi-directional data line, while the SPI interface has separate lines for sending and receiving data.

The problem is that this bi-directional data line is not fully compatible with the SPI data lines, even if it is only used as the receiving line. As a consequence, other sensors on the same SPI interface are disturbed by the misbehaving data line of the TC77.

This problem is solved by connecting the TC77 to a separate interface.

MMA7455L SPI Signal Level Conversion

On the original sensor interface, signal level conversion is done by using a Schottky diode on the MMA7455L input pin, which prevents the MMA7455L from receiving a 5 V signal, and pull-up resistor to 3.3 V. This concept works but is rather improvised. It should be replaced by a real signal level converter.

Voltage Measurement Transformer

The transformer used to break down the AC line voltage of the motor to a voltage of maximal 0.5 V to be measured by the ADE7753 is actually a power supply transformer. It is not designed to forward a signal for measurements, as the input signal is not a sine wave, but also contains high frequency components due to the phase angle control most 230 V power tools use.

It should be replaced by a transformer designed for voltage measurements.

No RPM Measurement Capability

The original sensor interface has no means to measure the motor speed or simulate it yet.

9.5 New Sensor Interface Design

Despite the fact that most of the concepts work, the original sensor interface has some design issues. Additionally, it can only be used as an extension to the Atmel STK600 developer board. It cannot be integrated into a power tool because it is missing an internal power supply and, measuring 100 mm x 65 mm, it is simply too big to fit inside a power tool.

For these reasons, a new sensor interface is developed. The goal is to resolve the design issues of the original sensor interface and to create a PCB that contains all necessary components of the sensor interface and which is compact enough to fit inside a power tool.

9.5.1 Components

Most components have been taken from the original design. Some have been changed in order to improve the design.

Power Supply

The new sensor interface has its own integrated power supply, so it is automatically powered as soon as the AC plug of the power tool is plugged in. Powering the sensor interface only when the power tool is turned on would cause the sensor interface to react too slow, because it needs at least half a second to initialize itself properly. It would also make it impossible to configure the sensors remotely.

Power is delivered by a Block 6 V transformer, its output is rectified to a 8.4 V DC voltage using an SMD rectifier. At 6 V AC, the transformer can deliver up to 0.35 W, which means it could deliver approximately 40 mA at 8.4 V DC. To get a regulated 5 V and 3.3 V supply, two LM1117 voltage regulators are used, one is a 5 V type and the other a 3.3 V type. These voltage regulators can deliver up to 500 mA theoretically.

3.3 V Signal Level Conversion

The MMA7455L needs 3.3 V SPI signal levels, whereas the AVR and the other sensors use a 5 V level. To get a cleanly converted signal, a Texas Instruments TXS0104 4-channel signal converter is used.

Voltage Measurement Transformer In the original sensor interface, the same Block 6 V transformer as the one used for the power supply has been used for isolated voltage measurement. In the new sensor interface design, it is replaced by a real voltage measurement transformer, which is also used in high-voltage testing equipment. The transformer has 2300 primary and secondary windings and therefore provides a 1:1 transformation. It also has a very small internal resistance.

Due to the 1:1 transformation, a 1:680 voltage divider is connected to the secondary coil to get the 0.5 V signal level required by the ADE7753's ADC.

The measurement transformer allows a more accurate measurement of the motor voltage, as it should not distort the signal as much as a simple supply transformer.

The filter capacitors are omitted, so higher frequencies can be detected. The ADE7753 samples at 27 kHz, so it should be able to handle them.

9.5.2 PCB Layout

The PCB layout based on the schematic is a two-layered design where most of the components are SMD. Almost all active components such as ICs are placed on the top layer (as shown in Figure 7), except for the RS485 transceiver, which is placed on the bottom layer (Figure 8) due to lack of space.

The new PCB is considerably smaller than the original design, despite the fact that it contains more components. It measures 83 mm x 63 mm. The only problem is that the supply and measurement transformers add about 15 mm of height, the current transformer even 24 mm.

For testing purposes, the sensor interface is attached to the GST 135 BCE. This setup is shown Figure 9, with the the add-on boards mounted as well. Currently, there is no encasing yet, as the sensor interface cannot be made so small that it fits inside the original encasing.

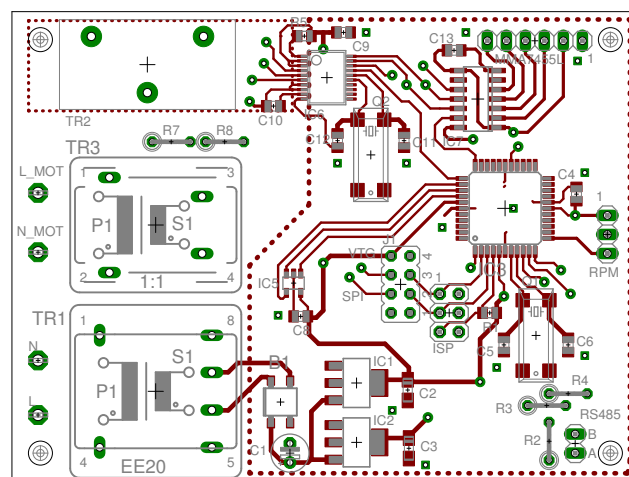


Figure 7: The top layer of the new sensor interface layout

9.5.3 Add-On Boards

Some of the sensor interface components are implemented on separate add-on PCBs, because they are either hard to replace in case they fail, or they might require modifications in case they do not work as expected.

RPM Interface

In order to get the motor speed from the GST 135 BCE, we have to tap into the signal of the motor control PCB. When unmodified, this signal outputs rectangular impulses at a 5 V signal level. The faster the motor rotates, the higher the frequency of the impulses. Since the power supply of the motor control PCB is very simple, the signal could be at phase potential, which might be dangerous.

To be on the safe side, the signal is optically isolated using an optocoupler, as seen in the schematic of Figure 10. The 15kΩ resistor limits the current running through the optocoupler, it has to be chosen carefully because the motor speed signal can only drive a small load. The right side of the optocoupler is connected to an external interrupt pin of the AVR MCU on the sensor interface, which counts the impulses as they arrive.

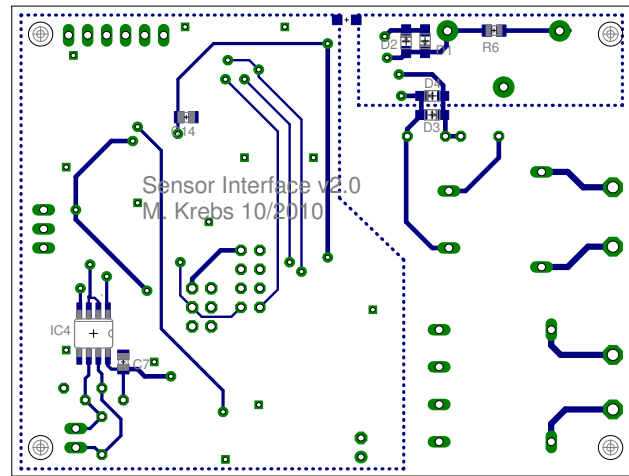


Figure 8: The bottom layer of the new sensor interface layout

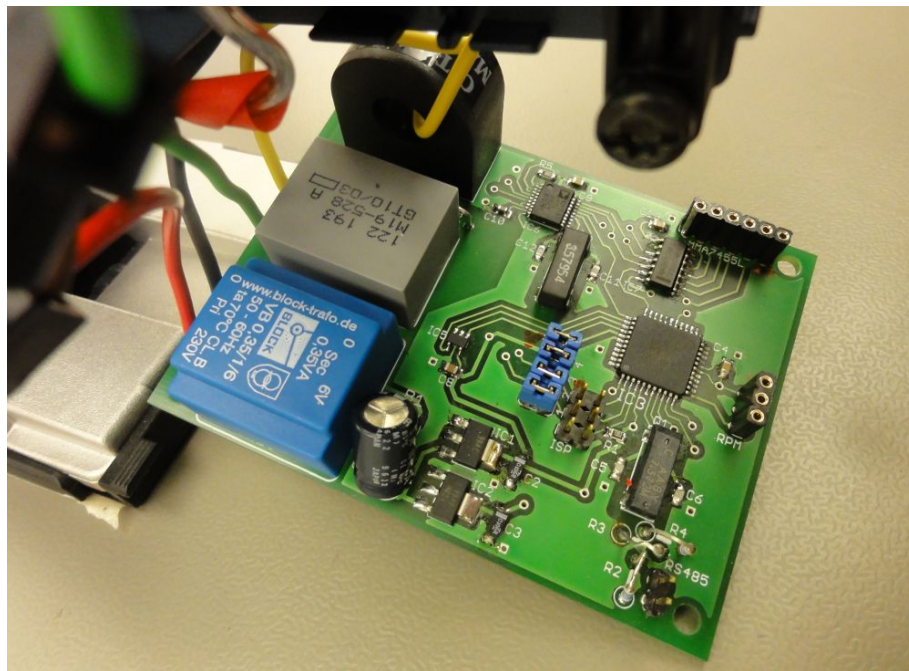


Figure 9: The new sensor interface, connected to a Bosch GST 135 BCE jigsaw

The optocoupler driven by the speed signal does in fact affect its waveform, but the motor speed control still works.

The RPM interface PCB, as shown in Figure 11, can be mounted on top of the sensor interface, and replaced if necessary, for example if a different power tool is tested. The finished PCB is shown in Figure 12, its right side is taped because some of the PCB material has been ground off, and for better

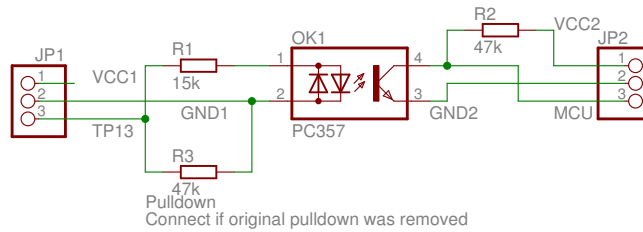


Figure 10: The schematic of the RPM input interface

insulation.

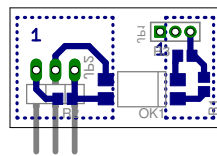


Figure 11: The layout of the RPM input interface

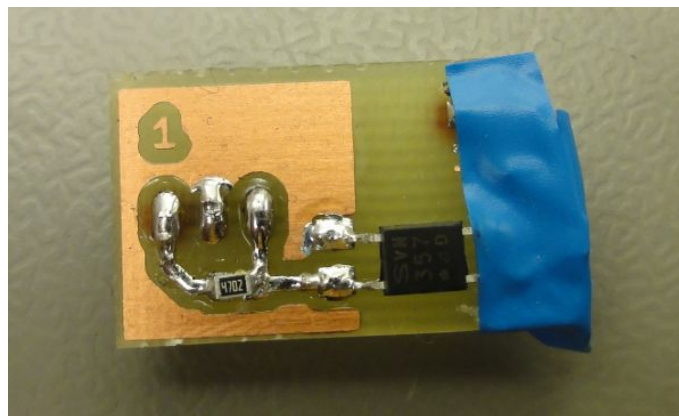


Figure 12: The prototype PCB of the RPM input interface. The right side is taped for better insulation.

MMA7455L Adapter

The MMA7455L accelerometer is not soldered directly to the sensor interface, because it is difficult to solder due to its LGA package. It is also very fragile, and to make replacement easier, a small carrier board is designed.

The schematic in Figure 13 contains only the IC itself and some supporting capacitors, thus it is very simple. The layout in Figure 14 is designed so that the accelerometer lies flat, with the X axis pointing forward, the Y axis to the left and the Z axis downward.

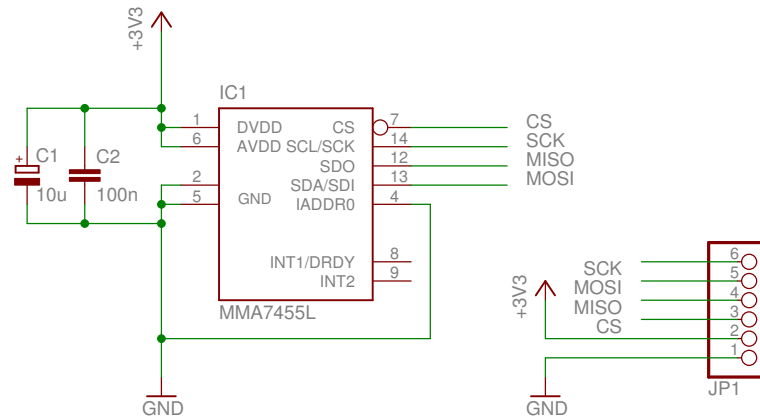


Figure 13: The schematic of the MMA7455L adapter board

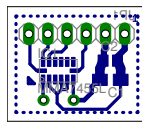


Figure 14: The layout of the MMA7455L adapter board

As seen in Figure 15, the IC is casted in silicone or hot glue to protect it against mechanical damage.

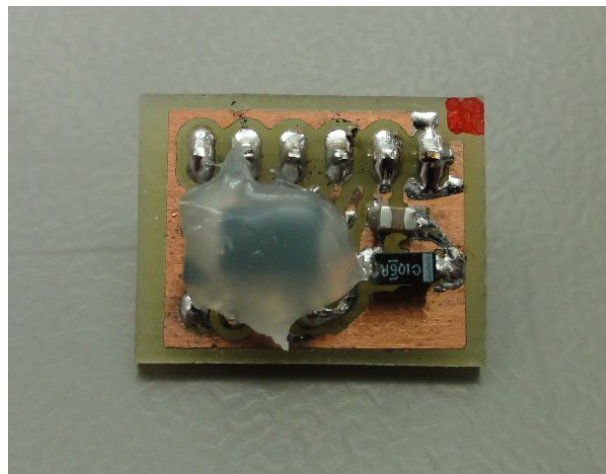


Figure 15: The prototype PCB of the MMA7455L adapter board. The MMA7455L IC is casted in hot glue for mechanical protection.

9.6 Microcontroller Software

The sensor interface software is implemented in C on an Atmel AVR microcontroller.

9.6.1 Structure

A microcontroller software implemented in C on a microcontroller is a low-level software that has to deal with only few system resources like processing performance and memory. It is therefore optimized for small and efficient code and not on a sophisticated software design.

The C software is more difficult to modularize than with higher-level programming languages, as it is not object-oriented. All functionality has to be implemented in functions, data that is needed by different functions must be either passed by reference or defined as a global variable.

The structure of the sensor interface is defined as follows:

The functionality is divided into header and source files at the finest possible granularity. AVR-specific constructs such as interrupt handlers are placed in the appropriate source files. The actual program flow is realized in the main() function, not every step is broken down into separate functions for reasons of efficiency.

A global configuration header file contains all the parameters that can be changed at compile-time, for example the configuration of I/O ports and the explicit inclusion or exclusion of sensors.

Figure 16 shows an overview of the sensor interface software. The different headers are illustrated as UML classes that are grouped into virtual packages, according to their relatedness. Headers that include other headers are illustrated as dependency relationships that point to the included header. An exception is the global configuration, which is included by almost all other headers.

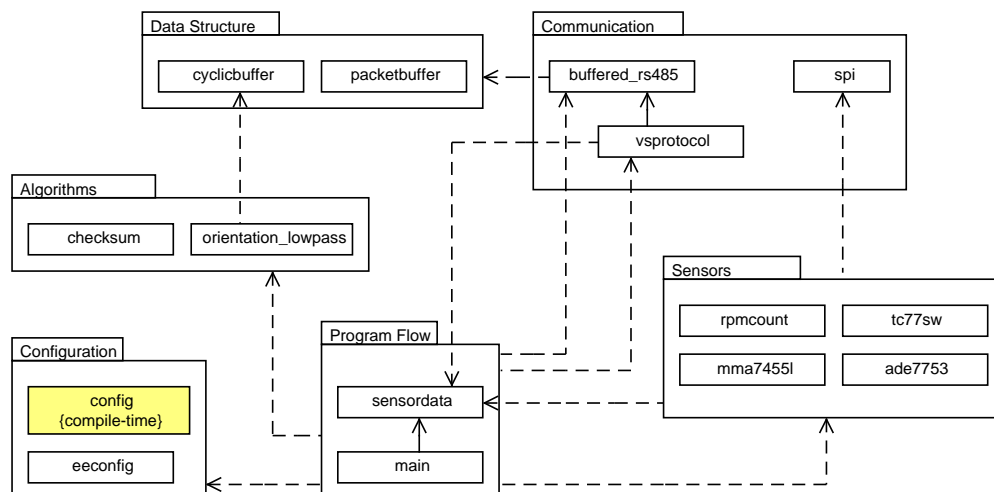


Figure 16: An overview of the MCU software structure on the sensor interface. Parts with similar functionality are grouped into virtual packages.

9.6.2 Code Documentation

This section describes the relevant parts of the sensor interface source code.

defs.h

The header file `defs.h` defines different data types and macros that are used by other components of the software. These custom data types improve the readability of the source code and allow quick changes of a definition without changing much of the source code. They also help understanding data types, as C does not support boolean values, for example. The macros are abbreviated representations of complex software constructs.

Data Types:

- `boolean` defined as `uint8_t`
- `byte` defined as `uint8_t`
- `uint8` defined as `uint8_t`
- `uint16` defined as `uint16_t`
- `uint32` defined as `uint32_t`
- `int8` defined as `int8_t`
- `int16` defined as `int16_t`
- `int32` defined as `int32_t`

Macros:

- `TRUE` defined as 1
- `FALSE` defined as 0
- `bit(b)` represents a specific bit within an AVR register.
- `setBits(reg, bits)` allows setting bits within an AVR register without changing other bits.
- `clearBits(reg, bits)` allows clearing bits within an AVR register without changing other bits.

config.h

The file `config.h` is a global configuration file for all other software components. The macros contained allow the reconfiguration of the source code, for example for redefining I/O ports to adapt them to a different MCU, or for activating and deactivating specific sensors.

Activation of Components (Deactivation by commenting out the macros):

- `USE_DEBUG_LEDS`
Debug LEDs on STK600 are used.
- `USE_SPI`
SPI interface is enabled (required to access sensors).
- `USE_TC77SW`
TC77 sensor with software SPI interface is enabled.

- `USE_ADE7753`
ADE7753 sensor is enabled.
- `USE_MMA7455L`
MMA7455L sensor is enabled.
- `USE_ORIENTATION_LOWPASS`
software-based low-pass filter is used when reading MMA7455L.
- `USE_RPMCOUNT`
speed impulse counter is enabled.

Configuration Macros:

- `SPI_CLOCKDIV`
divisor of SPI clock (relative to CPU clock)
- `SW3WIRE_SCK_PERIOD_US`
half clock cycle of 3-wire interface in microseconds
- `SW3WIRE_SCK_PORT`
register of the clock signal output pin
- `SW3WIRE_SCK_BIT`
register bit of the clock signal output pin
- `SW3WIRE_SCK_DIRREG`
direction register of the clock signal
- `SW3WIRE_SIO_PORT`
register of the data signal output pin
- `SW3WIRE_SIO_PIN`
register bit of the data signal output pin
- `SW3WIRE_SIO_BIT`
register bit of the data signal output pin
- `SW3WIRE_SIO_DIRREG`
direction register of the data signal
- `TC77_CS_PORT`
register of the TC77 chip select pin
- `TC77_CS_BIT`
register bit of the TC77 chip select pin
- `MMA7455L_CS_PORT`
register of the MMA7455L chip select pin
- `MMA7455L_CS_BIT`
register bit of the MMA7455L chip select pin
- `ADE7753_CS_PORT`
register of the ADE7753 chip select pin
- `ADE7753_CS_BIT`
register bit of the ADE7753 chip select pin

- `UART_BUFSIZE`
UART buffer size (RS485)
- `UART_BAUDRATE`
UART baud rate (RS485)
- `FRAME_BUFSIZE`
buffer size for a protocol data frame
- `LOWPASS_SIZE`
number of samples of the orientation low-pass filter

concurrency.h

This header file contains macros to prevent the interruption of the program flow by interrupts. This is important when accessing the same resource from the normal program flow as well as an interrupt service routine.

Macros:

- `ENTER_CRITICAL_SECTION()`
Beginning of a critical section (interrupts are disabled)
- `EXIT_CRITICAL_SECTION()`
End of a critical section (interrupts are re-enabled)

packetbuffer.h

This header file provides a linear byte buffer, which is used for transmission of data packets. The buffer is linear, so the data array is directly accessed when reading. Writing is possible through a convenience function as well.

Data Structure:

packetbuffer defined as `packetbuffer_struct`

- `data: byte*`
pointer to the data array
- `size: uint8`
size of the data array
- `pos: uint8`
current position within the array

Functions:

- `void initPacketBuffer(packetbuffer* buf, uint8 size)`
Initialize buffer with specified size.
- `void freePacketBuffer(packetbuffer* buf)`
Free the buffer memory.
- `boolean addByteToPacketBuffer(packetbuffer* buf, byte b)`
Add a byte to the buffer. If it is not full, *TRUE* is returned. Otherwise, *FALSE* is returned.
- `boolean addBytesToPacketBuffer(packetbuffer* buf, byte* bytes, uint8 num)`
Add multiple bytes to the buffer. If there is enough capacity, *TRUE* is returned. Otherwise, *FALSE* is returned.

- `void clearPacketBuffer(packetbuffer* buf)`
Clear the buffer (position set to 0).

cyclicbuffer.h

This header file provides a cyclic byte buffer, which is used as receive and transmit buffer for the UART. Since the buffer is cyclic, data is appended at the back and removed at the front.

Data Structure:

cyclicbuffer defined as cyclicbuffer_struct

- `data: byte*`
pointer to the data array
- `size: uint8`
size of the data array
- `putIdx: uint8`
index to append the next byte
- `getIdx: uint8`
index to remove the next byte
- `dataLen: uint8`
number of bytes in the buffer

Functions:

- `void initCyclicBuffer(cyclicbuffer* buf, uint8 size)`
Initialize the buffer with the specified size.
- `void freeCyclicBuffer(cyclicbuffer* buf)`
Free the buffer memory.
- `boolean addByteToCyclicBuffer(cyclicbuffer* buf, byte b)`
Add a byte to the buffer. If it is not full, *TRUE* is returned. Otherwise, *FALSE* is returned.
- `boolean addBytesToCyclicBuffer(cyclicbuffer* buf, byte* bytes, uint8 num)`
Add multiple bytes to the buffer. If there is enough capacity, *TRUE* is returned. Otherwise, *FALSE* is returned.
- `boolean getByteFromCyclicBuffer(cyclicbuffer* buf, byte* b)`
Remove a byte from the buffer.
- `void clearCyclicBuffer(cyclicbuffer* buf)`
Empty the buffer (putIdx and getIdx set to 0).

checksum.h

This header file provides a simple checksum algorithm which computes a XOR checksum of a byte array.

Functions:

- `byte computeChecksum(byte* data, uint8 len)`
Compute the checksum of the specified array. The number of bytes in the array must be specified explicitly.

orientation_lowpass.h

This header file contains a simple software-based low-pass filter which takes a certain amount of accelerometer values and calculates the average. A cyclic buffer is used internally for each axis, since the filter works according to the sliding window concept.

Functions:

- `void orientationLowpassInit(uint8 size)`
Initialize the filter with the specified size (per axis).
- `orientationLowpassShift(int8 x_in, int8 y_in, int8 z_in, int8* x_out, int8* y_out, int8* z_out)`
Insert a new accelerometer value. The new average is output.

buffered_rs485.h

This header file provides a buffered UART implementation for RS485. A receive and transmit buffer is used. The transmit buffer is filled by using the cyclic buffer functions and transmitted using interrupts. The receive buffer is filled automatically using interrupts, so it has to be read on a regular basis.

Macros:

- `DATABITS_5`
a macro for setting the UART configuration register to 5 data bits
- `DATABITS_6`
see above, for 6 data bits
- `DATABITS_7`
see above, for 7 data bits
- `DATABITS_8`
see above, for 8 data bits
- `DATABITS_9`
see above, for 9 data bits
- `STOPBITS_1`
1 stop bit
- `STOPBITS_2`
2 stop bit
- `PARITY_NONE`
no parity
- `PARITY_EVEN`
even parity
- `PARITY_ODD`
odd parity

Functions:

- `void RS485Setup(uint32 baudrate, uint8 databits, uint8 stopbits, uint8 parity, ringbuffer* txBuf, ringbuffer* rxBuf)`
Initialize the UART with the specified parameters. The baud rate is given numerically, the other parameters must be set using macros.

- `void RS485Shutdown()`
Disable the UART.
- `void RS485TransmitBuffer()`
Start transmitting the transmit buffer.
- `boolean RS485TransmitComplete()`
Return *TRUE* if transmit buffer is transmitted completely, *FALSE* otherwise.

eeconfig.h

This header file contains functions to save and load sensor configuration parameters in the MCU's EEPROM. All sensors are included in the header file, but only the functions for the sensors enabled in `config.h` are compiled. The definition of the EEPROM variables and their default values are stored in `eprom.c`, because the compiler would throw an error otherwise, due to a double definition.

Functions:

- `uint16 getTimerDelay(uint16 interval_ms)`
Converts a millisecond timer interval into a divisor used in the main program.
- `void tc77LoadConfig(uint16* tc77Delay, int16* tc77Offset)`
Loads the TC77 configuration from EEPROM.
- `void tc77SaveConfig(uint16 timer_ms, int16 tc77Offset)`
Saves the TC77 configuration into EEPROM.
- `void ade7753LoadConfig(uint16* ade7753Delay, int8* ade7753ch1Offset, int8* ade7753ch2Offset, int16* ade7753iRmsOffset, int16* ade7753vRmsOffset)`
Loads the ADE7753 configuration from EEPROM.
- `void ade7753SaveConfig(uint16 timer_ms, int8 ade7753ch1Offset, int8 ade7753ch2Offset, int16 ade7753iRmsOffset, int16 ade7753vRmsOffset)`
Saves the ADE7753 configuration into EEPROM.
- `void mma7455lLoadConfig(uint16* mma7455lDelay, int16* mma7455lOffset)`
Loads the MMA7455L configuration from EEPROM.
- `void mma7455lSaveConfig(uint16 timer_ms, int16* mma7455lOffset)`
Saves the MMA7455L configuration into EEPROM.
- `void pulseCountLoadConfig(uint16* pulseCountDelay)`
Loads the speed impulse counter configuration from EEPROM.
- `void pulseCountSaveConfig(uint16 timer_ms)`
Saves the speed impulse counter configuration into EEPROM.

spi.h

This header file contains basic functions to access the SPI interface. These functions are used in the implementation of the sensors.

Macros:

- `CLOCKDIV_4`
SPI clock of $\frac{1}{4}$ of the CPU clock

- `CLOCKDIV_16`
SPI clock of $\frac{1}{16}$ of the CPU clock
- `CLOCKDIV_64`
SPI clock of $\frac{1}{64}$ of the CPU clock
- `CLOCKDIV_128`
SPI clock of $\frac{1}{128}$ of the CPU clock
- `CLOCKPHASE_SETUPSAMPLE`
Data bit is set at the first clock edge and sampled at the second edge.
- `CLOCKPHASE_SAMPLESETUP`
Data bit is sampled at the first clock edge and set at the second edge.
- `RISING_EDGE`
Clock cycles start with rising edge.
- `FALLING_EDGE`
Clock cycles start with falling edge.
- `MASTER`
SPI interface defined as master.
- `SLAVE`
SPI interface defined as slave.
- `MSB_FIRST`
MSB transferred first.
- `LSB_FIRST`
LSB transferred first.

Functions:

- `void spiSetup(byte divider, byte phase, byte polarity, byte hosttype, byte order)`
Enables the SPI interface and configures clock divider, clock phase, clock polarity, master/slave and bit order. The parameters must be specified using the macros above.
- `void spiShutdown()`
Disables the SPI interface.
- `void spiWrite(byte b)`
Writes a byte to the SPI interface. Non-blocking since interrupts are used.
- `byte spiWriteAndRead(byte b)`
Writes and reads a byte.
- `byte spiWait()`
Waits until a write operation finished.

sensordata.h

This header file provides a data structure containing all sensor data registers. Only the enabled sensors are compiled in. For each sensor, data and configuration registers are provided.

ade7753.h

This header file contains functions to access an ADE7753 sensor.

Functions:

- `void ade7753Init (byte spiClockDiv)`
Initializes the SPI interface with ADE7753 parameters.
- `void ade7753Enable ()`
Enables the chip select.
- `void ade7753Disable ()`
Disables the chip select.
- `void ade7753Configure (boolean DC, int8 phaseCalibration, int16 vRmsOffset, int16 iRmsOffset, int8 ch1Offset, int8 ch2Offset)`
Configures the sensor.
- `uint32 ade7753ReadCurrent ()`
Reads the current.
- `uint32 ade7753ReadVoltage ()`
Reads the voltage.
- `int32 ade7753ReadEnergy ()`
Reads the accumulated energy.

9.6.3 mma7455l.h

This header file contains functions to access an MMA7455L sensor.

Functions:

- `void mma7455lInit (byte spiClockDiv)`
Initializes the SPI interface with MMA7455L parameters.
- `void mma7455lEnable ()`
Enables the chip select.
- `void mma7455lDisable ()`
Disables the chip select.
- `void mma7455lConfigure ()`
Configures the sensor.
- `void mma7455lReadXYZ (int8* x, int8* y, int8* z)`
Reads the axis acceleration with 8-bit precision.
- `void mma7455lReadXYZ10 (int16* x, int16* y, int16* z)`
Reads the axis acceleration with 10-bit precision.
- `void mma7455lCalibrate ()`
Calibrates the axis offsets. The sensor must be in horizontal position.
- `void mma7455lSetOffset (int16 x, int16 y, int16 z)`
Sets the axis offsets to compensate.

tc77sw.h

This header file contains functions to access an TC77 sensor through a software-based 3-wire interface.

Functions:

- `void tc77swEnable()`
Enables the chip select.
- `void tc77swDisable()`
Disables the chip select.
- `void tc77swConfigure(int16 offset)`
Sets the temperature offset.
- `int16 tc77swReadTemperature()`
Reads the temperature.

rpmcount.h

This header file contains functions to access a speed impulse counter.

Functions:

- `void rpmCountSetup()`
Initializes the impulse counter and enables the interrupt handler.
- `uint16 rpmCountGetRPM()`
Reads the current count and sets it to 0.

vsprotocol.h

This header file contains the Versatile Sensor Protocol implementation, based on the specification in Section 8.

Data Structure: `vspdata` defined as `vspdata_struct`

- `frameBuffer`: `packetbuffer*`
pointer to a packet buffer
- `fsmState`: `uint8`
current state machine state
- `byteCount`: `uint8`
bytes read in
- `dataSize`: `uint8`
data size of the current packet
- `sendReply`: `boolean`
ready to send reply
- `frameError`: `uint8`
the frame error code

Functions:

- `void vspInit(vspdata* ds)`
Initializes the protocol.
- `void vspReceiveByte(vspdata* ds, byte b)`
Processes next incoming byte.
- `void vspInterpretFrame(vspdata* ds, sensorData* sdata)`
Processes the data frame received.
- `void vspReplyError(vspdata* ds, uint8 errCode)`
Creates an error reply.
- `void vspReplyCmdAck(vspdata* ds, uint8 cmd, boolean ack)`
Creates an acknowledge reply.
- `void vspReplyReadSensorValue(vspdata* ds, sensorData* sdata)`
Creates a sensor value reply.
- `uint8 vspHandleWriteSensorConfig(vspdata* ds, sensorData* sdata)`
Processes a sensor configuration command.

main.c

This source file contains the `main()` function, which is the main program.

Functions:

- `int main()`
The main program, containing the program flow described in Figure 17.

Program Flow:

In an AVR program, all I/O pins must be configured first, so they are correctly set up as input or output. In the main program, the chip select pins of the sensors are configured first, so they can be accessed.

The second step is loading the sensor configuration from EEPROM. They are placed in the `sensordata` data structure. Also, other data structures and buffers are initialized.

After that, SPI and UART are activated, then the sensors are configured with their previously loaded parameters.

Before the program goes into an endless loop, the main sensor timer is started. The timer is clocked at 100 Hz, which is the base clock for all sensors. The timer is controlled by interrupt, each tick increases a counter. Based on this counter, the sensors are read in individual intervals, which are a divisor of the 100 Hz base clock.

Within each iteration of the endless loop, all sensors are checked for their timer interval being expired. If it is expired, the sensor is read out and its data is put into the appropriate registers of the `sensordata` structure.

After all sensors have been handled, the RS485 interface is checked. If the interface is in transmission mode, which means there is data to be transmitted, the data transmission is started. If the transmission is already complete, the interface is set to receive mode again. If the interface is in receive mode, all data in the receive buffer is processed. When a data frame is received completely, the command it represents is processed. Should the command require a response, it is prepared and the interface is set to transmission mode.

The endless loop should never terminate, because if it does, the AVR goes into an undefined state.

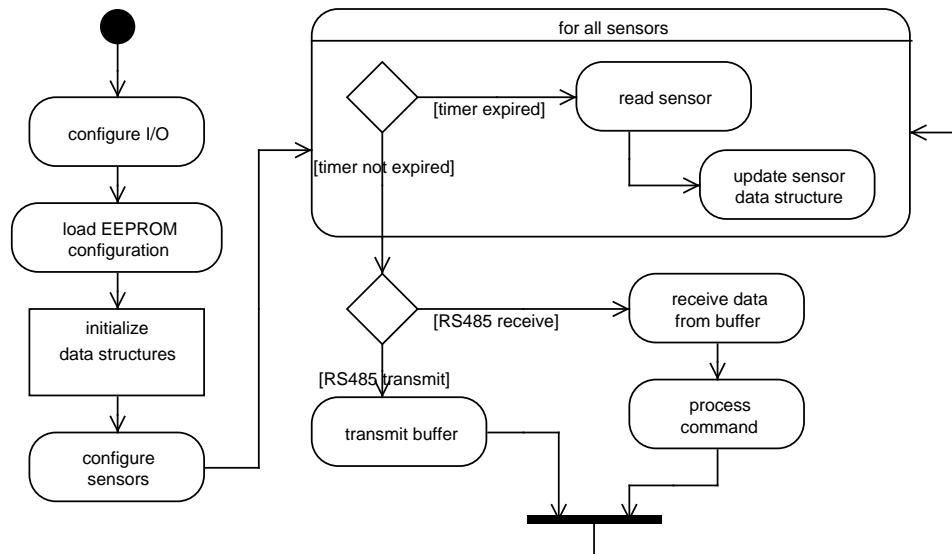


Figure 17: The general program flow of the MCU software on the sensor interface

9.7 A C++ Library for Sensor Access

The *Versatile Sensor Protocol* implemented in the sensor interface MCU software is a slave device implementation. On a data logger device, or any other device accessing sensor through the protocol, a master device implementation is required.

This implementation is developed in C++, so we can profit from an object-oriented design, which allows new sensor implementations to be added more easily than with C functions.

9.7.1 Structure

The structure of the Versatile Sensor Protocol library is based on a command-pattern-like design, as seen in Figure 18.

The central object is the `VSProtocol` class, which performs the basic protocol handling and serial communication. The commands are all implemented as objects derived from the `VSCommand` base class. The command classes contain the methods needed to process the command according to the specification in Section 8, as well as the data that is returned when the command is executed.

The most important sensor commands are the `VSReadSensorValueCommand` and the `VSWriteSensorConfigCommand`. These classes are only base classes, the actual command implementations must be derived from them. The derived commands are sensor-specific, each type of sensor needs its own implementation.

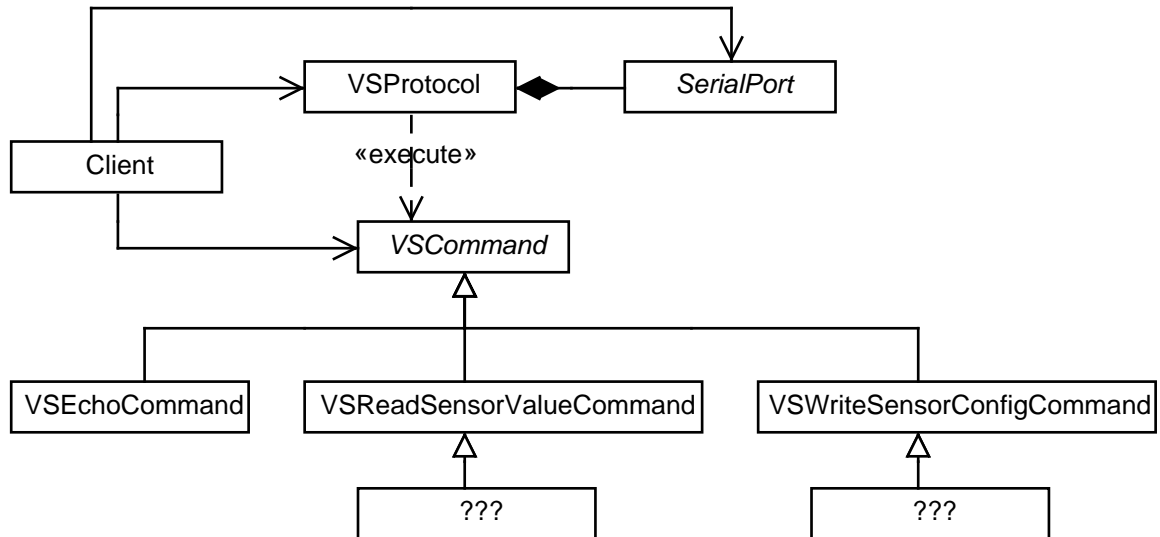


Figure 18: The basic structure of the Versatile Sensor Protocol C++ implementation

9.7.2 Protocol Control Class

The core class is the `VSPProtocol` class. It provides a physical serial connection by sending and receiving data through an attached `SerialPort` object (see Section 19.3 for more details).

Type Definitions

- `byte` defined as `char`
- `uint8` defined as `unsigned char`
- `uint16` defined as `unsigned short`
- `uint32` defined as `unsigned int`
- `int8` defined as `signed char`
- `int16` defined as `signed short`
- `int32` defined as `signed int`

The types are explicitly defined as *signed* or *unsigned*, because certain C++ compilers, for example the GNU compiler for the ARM architecture, define the `char` type as an 8-bit unsigned integer, whereas other compilers define it as a signed 8-bit integer.

Interface

The following public methods are available on the `VSPProtocol` class:

- `VSProtocol(SerialPort& serial)`
The constructor, which takes the serial port object to use as parameter.
- `~VSProtocol()`
The default destructor. Frees up allocated buffer memory.
- `void Initialize()`
Initializes the protocol and resets the data frame handling.
- `int32 ExecuteCommand(VSCommand* cmd, int32 timeout_us)`
Executes the given command. An error code according to Appendix A.3 is returned. A timeout in microseconds can also be specified.
- `void AbortCommand(VSCommand* cmd)`
Aborts the given command. Mostly used internally.
- `static byte ComputeChecksum(byte* array, int32 length)`
Computes a XOR checksum on a byte array of the given length.

State Machine

The `ExecuteCommand` method handles the command requests and responses on a frame level, which means it processes the frames but not their content.

A state machine with two state variables is used to handle the data frames. The *communication state* has the three states `COM_IDLE`, `COM_SENDING` and `COM_RECEIVING`. When executing a command, the state must be `COM_IDLE`, otherwise the interface is busy already. When transmitting the command request frame, the state goes to `COM_SENDING`. If it is transmitted successfully, the state goes to `COM_RECEIVING` and the receiver is waiting for the command response.

When receiving a command response frame, the receiver uses a second state machine with the following states:

| | |
|--------------------------------|--|
| <code>STARTBYTE_WAITING</code> | The receiver is waiting for the start byte of the response data frame. Other bytes are just discarded. |
| <code>HEADER_RECEIVING</code> | The receiver reads the three remaining header bytes, which contain the data payload size. If the data payload size exceeds the receive buffer, the state goes into <code>FRAME_TOOBIG</code> . |
| <code>DATA_RECEIVING</code> | The receiver reads a number of bytes according to the data payload size given in the header. |
| <code>FRAME_TOOBIG</code> | The frame is too big to fit inside the receive buffer, therefore the state machine is reset. |
| <code>CHECKSUM_WAITING</code> | After the data payload has been received, a final byte is read, which is the checksum byte. This checksum is verified against the previously received data, if it matches, the frame is considered complete. |
| <code>FRAME_COMPLETE</code> | When the frame is complete, the sender ID of the frame is verified. If it comes from the expected device, the frame can be processed by the executing command's methods. |

9.7.3 Commands

The commands are not processed in the `VSProtocol` class, but actually within the command objects. They contain the methods that are used to create command request frames and process received command responses.

VSCommand

This is the base class for all commands. All command implementations inherit these methods and override them.

- `VSCommand(uint8 cmdID, uint8 devID)`
The constructor, which takes the command ID and the slave device ID.
- `~VSCommand()`
The default destructor.
- `int32 CreateRequest(byte* buf, int32 bufSize)`
Copies a command request frame into a buffer, and returns its size.
- `void ProcessReply(byte* buf, int32 frameSize)`
Processes a command response and updates the command data if required.
- `uint8 GetCommandID()`
Gets the command ID.
- `uint8 GetDeviceID()`
Gets the slave device ID.
- `uint8 GetReplyCode()`
Gets the error code after command execution.

VSEchoCommand

This just sends a single data byte, which is returned by the slave device. Just implements the `CreateRequest` and `ProcessReply` methods.

VSReadSensorValueCommand

This command is a base class for reading sensor data. It implements the `CreateRequest` and `ProcessReply` methods. New methods are:

- `uint8 GetSensorType()`
Gets the sensor type ID according to Appendix [A.4](#). The sensor ID is defined by a derived class.

Actual sensor implementation inherit from this class and add methods to access the sensor data.

VSWriteSensorConfigCommand

This command is a base class for updating sensor configuration. It implements the `CreateRequest` and `ProcessReply` methods. New methods are:

- `uint8 GetSensorType()`
Gets the sensor type ID according to Appendix [A.4](#). The sensor ID is defined by a derived class.
- `bool GetAck()`
Gets the command response acknowledgement. If true, the configuration was updated successfully.

Actual sensor implementation inherit from this class and add methods to provide sensor configuration.

Part IV

Data Transmission using GPRS

This part describes how data is transmitted using a GPRS connection. In our concept, GPRS is the technology of choice for live transfer of acquired sensor data. We will discuss GPRS and what means are needed to use it, as well as alternative technologies that could be used.

10 GPRS Concept

10.1 About GPRS

GPRS is an abbreviation for *General Packet Radio Service* and stands for a cellular networking technology used in 2nd generation GSM networks. GPRS has been introduced in order to provide permanent internet connections and other data services for mobile phones. Before GPRS was available, the only way to get a data connection was CSD, which is basically a dial-up connection using GSM.

CSD connections are limited to 9.6 kbps and are priced by the time the connection is open. GPRS provides a permanent internet connection instead, which is not priced by time but by the amount of data that has been transferred. GPRS provides a variable bandwidth, depending on how many time slots are available and the bit encoding used. The maximal number of time slots used in a connection is eight, resulting in a downstream bandwidth of 171.2 kbps. However, this is an ideal situation, as most GSM cells limit the number of time slots used to four for downstream and one for upstream.

GPRS is a technology built upon GSM, so it uses existing GSM infrastructure as shown in Figure 19. However, it needs additional infrastructure to provide a connection to the internet. The SGSN (Serving GPRS Support Node) is the bridge between GSM and GPRS and provides GPRS access within a GSM cell. The GGSN (Gateway GPRS Support Node) is a gateway between GPRS and the internet.

GPRS itself provides only a data connection to the GPRS network, whereas internet communication protocols like TCP/IP are implemented on top of it.

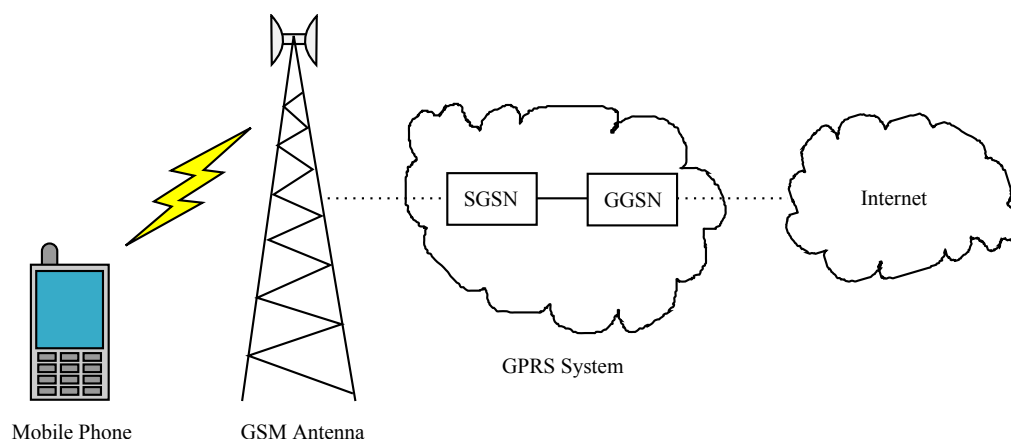


Figure 19: How the GPRS system works

Packet-oriented vs. connection-oriented

A GSM network cell divides its available bandwidth among the registered subscribers using time division multiplexing. This means that time is divided into time slots which are assigned to the subscribers. Within a time slot, the assigned subscriber can send and receive data.

Each GSM time slot is 576.92 microseconds long, eight time slots form a *frame*. Therefore, each frame is 4.61538 ms long. 26 frames form a GSM multiframe of 120 ms, which is the basic unit in GSM systems.

When using a standard GSM connection for voice calls, SMS and CSD calls (modem connection using GSM), it works in a connection-oriented fashion. Connection-oriented means that the network subscriber reserves its time slot permanently, even if no data is being transferred. This is a disadvantage from the provider's point of view, because the subscriber, even when idle, consumes some of the network cell's capacity, so it cannot be used by other subscribers.

GPRS, on the other hand, works in a packet-oriented fashion. Unlike standard GSM connections, GPRS does not permanently reserve a time slot. Instead, data is transferred in packets, thus a time slot is only reserved when data has to be transferred. To the subscriber however, a GPRS connection still looks as if it were a permanent connection.

Other technologies

Nowadays, GPRS is no longer the latest technology for mobile internet. Several improvements have been made to allow faster mobile internet connections.

EDGE (Enhanced Data Rates for GSM Evolution) is an extension to the 2nd generation GSM networks, which introduces a more efficient modulation of the data. This allows faster GPRS connections up to 473 kbps, in contrast to the 171.2 kbps standard GPRS is capable of.

UMTS (Universal Mobile Telecommunications System) is a 3rd generation cellular technology, which replaces or extends existing GSM networks. It features even more efficient data modulation and can provide connections up to 14.4 Mbps. But the improved performance comes at a price: UMTS devices have a considerably higher power consumption than GSM or GPRS devices.

10.2 GPRS and Mobile Data Acquisition

When a mobile data acquisition system requires the acquired data to be accessible in real time, an internet connection is necessary. Especially for mobile systems, a wireless connection is preferable, because the data acquisition device can be put in an arbitrary place and does not depend on the existence of wired network infrastructure.

Wireless technologies that are feasible for data acquisition include WLAN, industrial wireless technologies like Zigbee, Bluetooth and cellular network technologies like GSM and GPRS.

The disadvantage of most technologies is their limited range or the need of additional infrastructure. WLAN can provide internet access easily, but it is not available everywhere. Its existence is normally limited to populated areas, and is not available otherwise. Wireless technologies like Bluetooth and Zigbee have a limited range of less than 100 meters, and they need a base station and gateway in order to provide an internet connection. The only technology that is available almost everywhere is a cellular network.

We still have the possibility to choose among SMS, GPRS and UMTS for the data transfer. [FS06] discusses a feasibility study of using SMS in data acquisition applications, [WX09] discusses the usage of GPRS. They both state that either technology is suitable for mobile data acquisition.

In the end, the decision which technology to be used should be based on cost estimation and performance.

When using SMS, each message is limited to a maximum of 160 characters. Also, a single message is quite expensive compared to the amount of data it contains. In Switzerland for example, a single SMS costs between CHF 0.10 and 0.25, depending on the service provider. When large amounts of messages need to be sent, special contracts might be possible to reduce the costs per message. Still, SMS is only effective when only small amounts of data need to be transferred in infrequent intervals.

GPRS and UMTS provide permanent internet connections, with the only differences being the available bandwidth and the power consumption of the device. The cost for such mobile internet connections depends entirely on the amount of data being transferred. In Switzerland, mobile internet that includes 1 GB of traffic per month is available starting at CHF 15 per month. For commercial applications on a large scale, better condition might be negotiable. Whether to use GPRS or UMTS depends on the bandwidth needed to transfer all required data. If the bandwidth GPRS provides is sufficient, the use of GPRS is to be preferred, because it consumes less power than UMTS, and it has a higher availability

since the coverage of UMTS networks is smaller than the coverage of GSM/GPRS networks.

In the Scintilla usage scenario, we need to transfer small amounts of data frequently, therefore GPRS is the technology of choice.

11 GPRS Hardware

11.1 Hardware Choice

GPRS devices suitable for embedded applications can be purchased as complete integrated modules which only need external parts for power supply and communication. Most modules available on the market do not only provide GPRS functionality, but they basically have the same feature set as a mobile phone, including GSM voice calls, sending and receiving SMS and GPRS connectivity.

Some embedded modules have special features, therefore several variations of embedded GSM/GPRS exist:

- Modules that only provide basic GSM/GPRS functionality
- Modules with integrated GPS receiver
- Modules with a built-in embedded computer system (CPU, RAM and flash memory)

The embedded module of our choice is the Telit GM862-GPS, which is an embedded GSM module with integrated GPS receiver. There are even more compact modules available, such as the Telit GM863-GPS which has the same functionality but a smaller footprint. However, the smaller modules have BGA solder joints, which makes it impossible to solder them by hand, and the module cannot be replaced once soldered on. This is why we choose the GM862-GPS for prototyping.

11.2 The Telit GM862-GPS GPRS Module

The Telit GM862-GPS is an embedded quad-band GSM module which can be mounted on an SMD PCB. It includes all basic GSM features such as GSM voice calls and SMS, plus an integrated TCP/IP stack for use with GPRS, and an integrated GPRS receiver.

The GM862-GPS, which is shown in Figure 20, has a footprint of ca. 44 mm x 44 mm x 7 mm and can be mounted on a PCB using a 50-pin SMD Molex connector as shown in Figure 21.

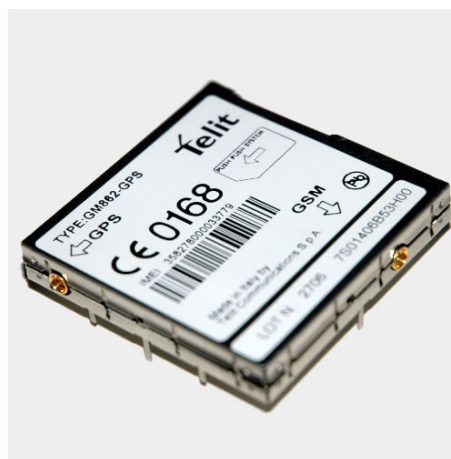


Figure 20: The Telit GM862-GPS embedded GSM module

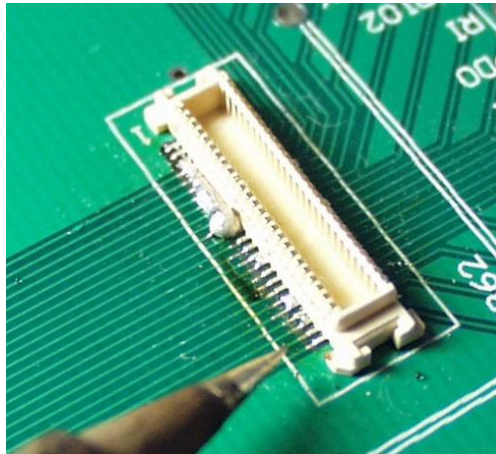


Figure 21: A 50-pin SMD Molex connector the GM862-GPS can be mounted on

11.2.1 Features

- GSM voice calls, headset connectivity
- sending and receiving SMS
- GPRS data access
- SIM card slot integrated
- one RS232 serial port at 2.8V CMOS level for GSM modem access and general control
- one RS232 serial port at 2.8V CMOS level for NMEA GPS data retrieval
- MMCX connector for GSM antenna
- MMCX connector for GPS antenna
- access through AT command set

11.2.2 Required external parts

Even though the Telit GM862-GPS is a highly integrated GSM module, it needs a few external parts in order to function properly:

- a 3.7 V power supply that can deliver a constant 500 mA to power the GSM module
- a 2.8 V (resp. < 3.3V) power supply for the serial ports, if signal level converters are used
- external GSM and GPS antennas
- an external switch or digital GPIO pin to turn the module on and off

11.2.3 Cost

The Telit GM862-GPS is available for ca. 70 € per piece when bought in small quantities, for example while developing a prototype. When bought in large quantities, a lower price is to be expected.

11.3 A Custom-Made Evaluation Board

There are several evaluation boards available for the Telit GM862-GPS, like the GM862 Evaluation Kit from SparkFun Electronics, and the EasyGSM/GPRS from Mikroelektronika.

The SparkFun evaluation board is available with either RS232 or an USB-serial converter, and it comes with the voltage regulators needed to power the GM862-GPS. Unfortunately, it does not have a second serial port for NMEA GPS data retrieval.

The EasyGSM/GPRS board from Mikroelektronika is a very simple evaluation board, which provides only an SMD connector to mount the GM862-GPS on and two antenna holders. All I/Os of the GM862-GPS are routed to two 26-pin pin headers for easier access. We use these pin headers to mount the evaluation board on top of a base board which provides a 3.7 V and 3.0 V power supply and RS232 signal converters for the two serial ports. This design allows us to connect the GM862-GPS modem or NMEA port to a PC or embedded system using standardized RS232 signals.

11.3.1 The Base Board PCB

The base board is a add-on board for the EasyGSM/GPRS evaluation board featuring power supplies for the GM862-GPS module and external RS232 D-Sub connectors.

The GM862-GPS module needs a 3.7 V supply, whereas the RS232 signal converters need a 3.0 V supply. The 3.7 V supply is provided by a LM317T voltage regulator which is capable of delivering up to 1.5 A, while the 3.0 V is provided by a LM317L, a smaller regulator providing up to 100 mA.

A MAX3237 signal converter is used to provide an external RS232 connector for the GM862-GPS modem port. The MAX3237 is a special RS232 converter supporting all modem signals used by the GM862-GPS modem port. The NMEA port only uses the RX and TX signals, therefore a simpler MAX3232 converter is used in this case.

Besides integrated circuitry, the base board contains a power and a reset switch for the GM862-GPS as well as power and status LEDs.

A schematic of the base board is shown in Figure 22, and the matching PCB layout is shown in its original size in Figure 23. Figure 24 shows the evaluation board and the base board next to each other, the base board is already modified to fix some design issues.

11.3.2 Errata

The base board design has a few design issues that can be fixed with some modifications.

- **3.0 V Voltage Regulator**

The LM317L used in the original design gets quite hot during operation. This is compensated by using a LM317T instead.

- **RX/TX Lines of the NMEA Port**

The RX and TX lines of the NMEA serial port are inverted due to contradictory documentation. The lines are easily swapped by rewiring the solder bridges in the center of the layout.

- **Switch Location**

The power and reset switches are placed too close to the evaluation board, so they are hard to reach when the evaluation board is on top of the base board. This could be fixed by wiring the switches externally.

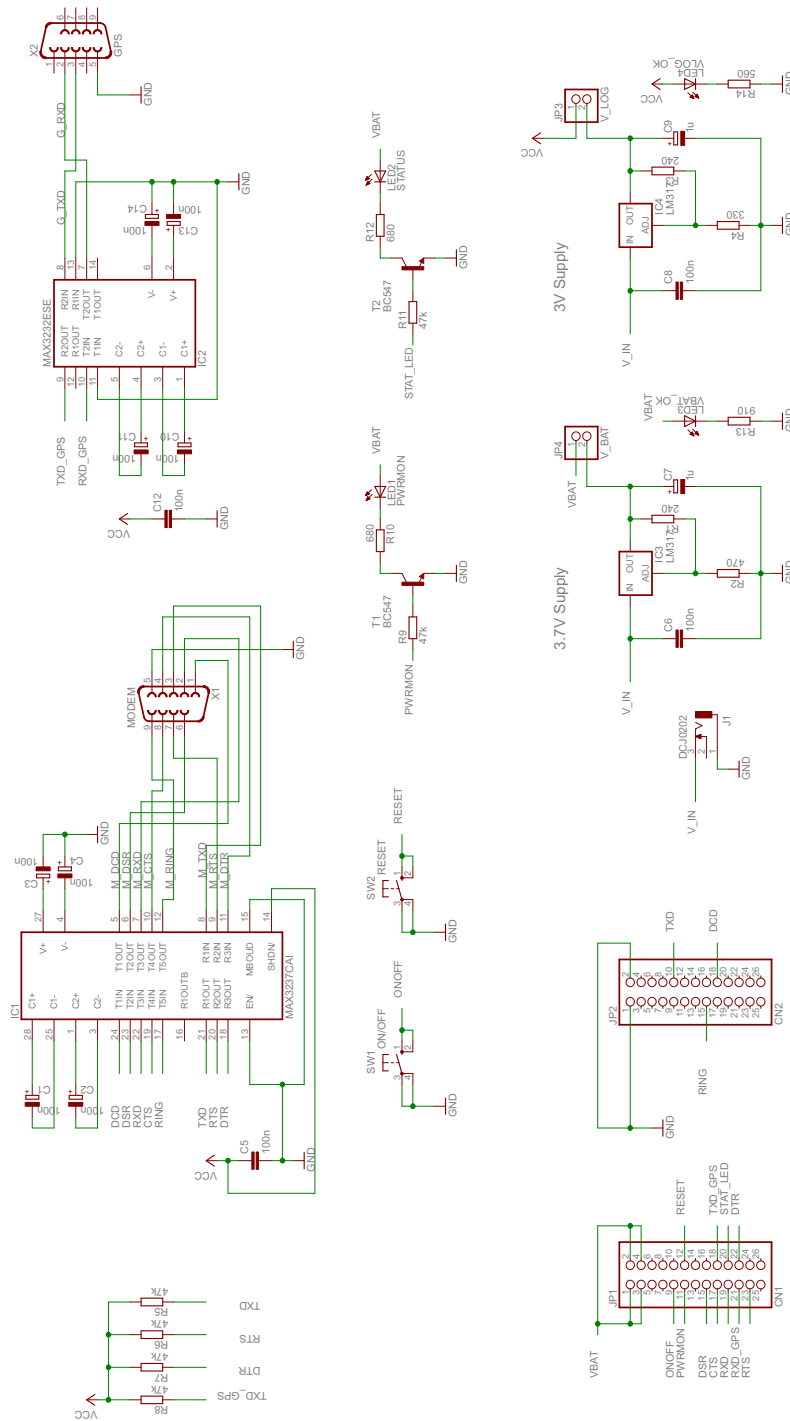


Figure 22: The schematic of the custom-made EasyGSM/GPRS base board

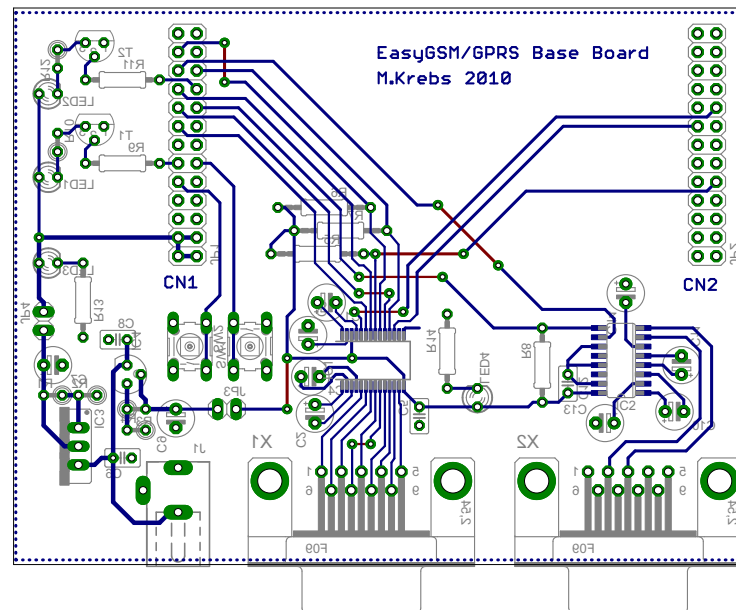


Figure 23: The layout of the custom-made EasyGSM/GPRS base board

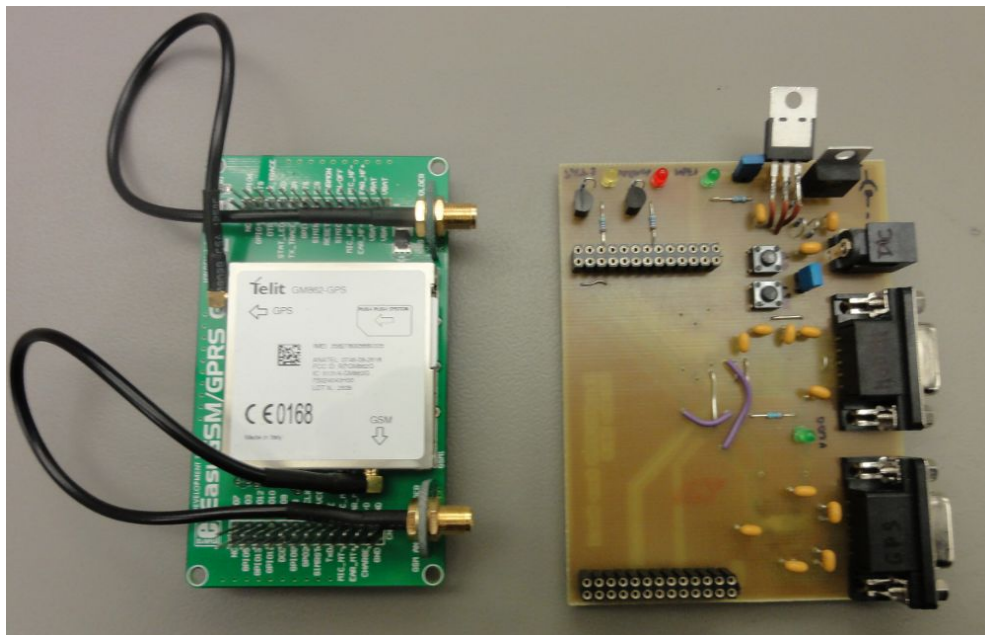


Figure 24: The EasyGSM/GPRS evaluation board and the custom-made base board

- **DC Jack Pinout**

The DC jack connects the ground plane to the center contact instead of the outer contact, which is normally the opposite. This is not a big problem, as many power supplies can be configured

appropriately for this setup.

12 GPRS Device Access

12.1 AT Command Set

The Telit GM862-GPS must be accessed using AT commands. For the basic functionality, standardized AT commands can be used. Advanced functions like the integrated TCP/IP stack and GPS however need Telit-specific commands.

Due to the excessively large AT command set available on the GM862-GPS, only the AT commands that are actually used in our implementation are mentioned here. A complete AT reference manual can be downloaded from the manufacturer's website.

The command syntax specified here is based on the assumption that the enhanced AT command set (#SELINT mode 2), the enhanced SMS command set (#SMSMODE mode 1) and SMS text mode (+CMGF mode 1) are used.

12.1.1 AT Requests and Responses

All AT command requests are sent on a single line and terminated with a *[CR]* character (0Dh). The response depends on the command that has been sent, it can consist of one or more lines of text. Response text lines are terminated with *[CR][LF]* (0D0Ah). Usually, the last token of the response is the word *OK* if the command is successful, or *ERROR* if an error occurs.

12.1.2 Initialization Commands

`ATE{on}`

Enables or disables the local echo on the serial port. If the echo is enabled, all data sent over the serial port is read back right away. Enabling echo is only useful when accessing the GM862-GPS over a serial terminal like Hyperterminal or GTKTerm.

`{on}`: 0 is disabled, 1 is enabled

`AT#SELINT={mode}`

Sets the AT command instruction set to be used.

`{mode}`: 0: old GM862-GSM or GM862-GPRS command set
 1: command set for modules with Python interpreter
 2: command set for newer products and GPS models

In our implementation, we use mode 2 since we have a GM862-GPS module. It is also recommended by the manufacturer.

`AT#SMSMODE={mode}`

Sets the SMS instruction set.

`{mode}`: 0: normal instruction set
 1: enhanced instruction set

In our implementation, we use mode 1, which is recommended by the manufacturer.

12.1.3 GSM Commands

AT+CPIN={*pin*}[, {*newpin*}]

Enters the PIN code of the SIM card. If the parameter *newpin* is optional, if it is set, the PIN code will be changed to the value of *newpin*.

{*pin*}: the current PIN code for the SIM card
{*newpin*}: the new PIN code to set

12.1.4 SMS Commands

AT+CMGF={*mode*}

Sets the SMS format when sending and receiving SMS.

{*mode*}: 0 is PDU mode, 1 is text mode

In our implementation, we use text mode. If PDU mode is used, the command syntax for all SMS commands is different.

AT+CMGL={*filter*}

Lists SMS stored in the module's memory. The parameter *filter* is used to display only a certain type of SMS.

{*filter*}: "REC UNREAD" displays unread SMS from the inbox only
 "REC READ" displays already read from the inbox SMS only
 "STO UNSENT" displays unsent SMS from the outbox only
 "STO UNSENT" displays sent SMS from the outbox only
 "ALL" displays all stored SMS

The response is of the form

+CMGL: {*index*}, {*stat*}, {*number*}, {*address*}, {*time*}[CR][LF]
{*text*}[CR][LF].

{*index*}: the message store index
{*stat*}: "REC UNREAD" SMS is unread
 "REC READ" SMS was read
 "STO UNSENT" SMS not yet sent
 "STO UNSENT" SMS was sent
{*number*}: the phone number the SMS was received from or sent to
{*address*}: an address book entry if the the phone number is in the address book
{*time*}: a timestamp in the format ` `YY/MM/DD, HH:mm:ss+04` `

If more than one message is found, there are also multiple responses.

AT+CMGD={*index*}

Deletes an SMS from the message store.

{*index*}: the message store index of the message to delete

AT+CMGS={*number*}

Writes a new SMS and sends it to the given destination number.

{number}: the destination phone number

After sending this command, a prompt `)` appears. After that, the SMS text can be submitted. To finish the text submission and send the SMS, the text submission must be finished by sending the character `!Ah`.

12.1.5 GPS Commands

`AT$GPSACP`

Retrieves the current GPS data, if available.

The response is of the form

`$GPSACP : {time}, {lat}, {lng}, {hdop}, {alt}, {fix}, {cog}, {spkm}, {spkn}, {date}, {nsat}[CR][LF]`

{time}: the UTC time in the format `'hhmmss'`
{lat}: the latitude in the format `'ddmm.mmmN/S'` (degrees/minutes)
{lng}: the longitude in the format `'dddmm.mmmE/W'` (degrees/minutes)
{hdop}: the horizontal dilution of precision
{alt}: the altitude in meters
{fix}: the fix quality (0 is no fix, 2 is 2D fix and 3 is 3D fix)
{cog}: the course over ground in degrees/minutes
{spkm}: the speed in km/h
{spkn}: the speed in knots
{date}: the date in the format `'DDMMYY'`
{nsat}: the number of visible satellites

If no GPS data is available, most of the comma-separated fields are just empty. When using this AT command, the response is different than when the separate NMEA serial port is used. This response does not comply with the NMEA standard.

12.1.6 GPRS Commands

`AT+CGDCONT=1, 'IP', {apn}, {address}, 0, 0`

Configures the GPRS PDP context. There are 5 user-definable contexts, context 0 is reserved for GSM. We only use context 1 for GPRS, no other contexts.

{apn}: the GPRS access point name (given by the provider)
{address}: the IP address to use. For DHCP enter "0.0.0.0".

`AT#SGACT=1, {on}[, {user}, {password}]`

Enables or disables the GPRS PDP context. When the context is to be enabled, a GPRS username and password (given by the provider) can be specified.

{on}: 0 is disabled, 1 is enabled
{username}: the GPRS username
{password}: the GPRS password

`AT#SGACT?`

Checks if the GPRS context is activated.

The response is of the form #SGACT: 1, {on}.

{on}: 0 is disabled, 1 is enabled

12.1.7 TCP/IP Socket Commands

AT#SCFG={id}, {cid}, {psize}, {ito}, {cto}, {txto}

Configures an IP socket.

{id}: the socket ID (1..6)
 {cid}: the PDP context ID
 {psize}: the packet size in bytes (1..1500)
 {ito}: the inactivity timeout in seconds (0..65535)
 {cto}: the connection timeout in tenths of seconds (10..1200)
 {txto}: the transmission timeout in tenths of seconds (0..255), flushes the buffer even if no complete packet is inside

AT#SCFGEXT={id}, {cid}, {srmode}, {rxmode}, {ka}, {lar}, {txmode}

Configures an IP socket (extended configuration, only with current firmware).

{id}: the socket ID (1..6)
 {cid}: the PDP context ID
 {srmode}: unsolicited RING mode (signal if data has arrived), 0 is normal (just socket ID), 1 is socket ID and number of bytes, 2 is socket ID, number of bytes and data preview
 {rxmode}: data receive mode, 0 is text mode, 1 is hexadecimal mode
 {ka}: keepalive timeout in minutes (0..240)
 {lar}: listen auto-response, 0 is disabled (default), 1 is enabled
 {txmode}: data transmit mode, 0 is text mode, 1 is hexadecimal mode

We use the hexadecimal transmit/receive mode in our implementation, because non-printable characters are easier to handle.

AT#SD={id}, {prot}, {rport}, {ip}, {ct}, {lport}, {mode}

Opens an IP socket.

{id}: the socket ID (1..6)
 {prot}: the protocol, 0 is TCP and 1 is UDP
 {rport}: the remote IP port
 {ip}: the remote IP address or DNS hostname
 {ct}: TCP closing type, 0 means immediately when remote host disconnects, 255 means only after +++ sequence
 {lport}: local IP port (UDP only)
 {mode}: connection mode, 0 is data mode and 1 is command mode

When data mode is chosen, the module goes into data mode, which means all characters sent will also be sent through the socket. When the +++ sequence is sent, the module goes back into command mode and the socket is suspended. We use the command mode in our implementation, so the socket stays open and data can be transmitted using AT commands.

AT#SRECV={id}, {length}

Reads data from the socket.

{id}: the socket ID (1..6)
{length}: the number of bytes to reads

The response is of the form

#SRECV: *{id}*, *{length}*[CR][LF]
{hexdata}[CR][LF].

{id}: the socket ID (1..6)
{length}: the number of bytes read
{hexdata}: the data as a hexadecimal string

AT#SSEND=*{id}*}

Writes data to the socket.

{id}: the socket ID (1..6)

After sending this command, a prompt `>` appears. After that, the data can be submitted in form of a hexadecimal string. The data submission must be finished by sending the character `1Ah`. A maximum of 1024 bytes can be written to the socket in one attempt, if there is more data it will be discarded.

AT#SH=*{id}*}

Closes the socket.

{id}: the socket ID (1..6)

AT#SS=*{id}*}

Gets the socket status.

{id}: the socket ID (1..6)

The response is of the form

#SS: *{id}*, *{state}*, *{lip}*, *{lport}*, *{rip}*, *{rport}*[CR][LF]

{id}: the socket ID (1..6)
{state}: the socket state, 0 is closed, 1 is active, 2 is suspended, 3 is suspended with data pending, 4 is listening, 5 is an incoming connection attempt
{lip}: the local IP
{lport}: the local port
{rip}: the remote IP
{rport}: the remote port

12.2 A C++ Library for the GM862-GPS

The GM862-GPS needs a software that handles the serial connection and processes the AT commands. Some operating systems, for example Ubuntu 10.10, detect the GM862-GPS as a GPRS modem and use PPP to establish an internet connection. This is practical because in this case, the connection works on the IP stack of the OS, and there is no need for additional software or drivers. However, there is no access to SMS and specific functionality that goes beyond the standardized GPRS commands.

Since we are using an embedded system, which does not have the capability of using the GM862-GPS as a GPRS modem directly, we need a software library to access the GM862-GPS. Unfortunately, Telit does not provide a software API, but only an AT command reference.

A few freely available implementations exist, for example a C# library has been found at <http://www.microframework.nl>. A free C or C++ API could not be found. Some companies have made their own APIs which are used in commercial embedded application, but none of them is freely available.

As a consequence, we make our own C++ API, which implements all the essential functionality and can be ported to different operating systems.

12.2.1 Design

The C++ API is designed as a C++ class `GM862GPS`. This class processes the AT commands described in Section 12.1. A `GM862GPS` instance also implements a custom TCP socket interface `ISocket`, so it can be used with applications that require sockets but should stay independent of the socket implementation. For serial connectivity, an external `SerialPort` object is used (see Section 19.3 for more details). An UML notation is shown in Figure 25

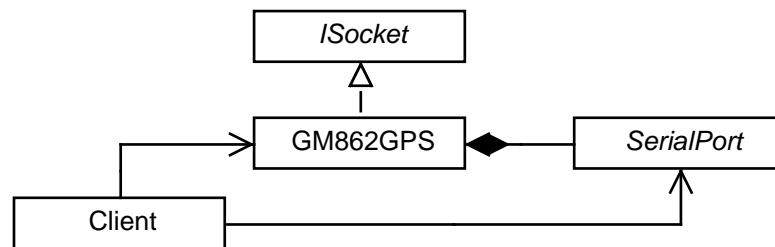


Figure 25: The GM862-GPS library design

The `ISocket` interface provides the pure virtual methods:

- `bool Open(std::string address, int port)`
- `void Close()`
- `bool IsOpen()`
- `std::istream* const GetInputStream()`
- `std::ostream* const GetOutputStream()`
- `bool Write(const char* data, int len)`
- `int Read(char* data, int offset, int len)`

Note: The `GM862-GPS` class does not implement the C++ stream methods, because it cannot always guarantee that enough data is available, and this would block the application when trying to read from the stream.

12.2.2 Command Processing

The command processing is done using a polling mechanism. When executing a command, the method blocks until the response is received or a timeout occurs. A command is only executed when the GM862-GPS is in its idle state, because it can only process one command at a time.

Before an AT command request is sent, the input buffer of the serial port is cleared. The reason for this is that the GM862-GPS occasionally sends unsolicited responses that represent certain events, such as the arrival of an SMS. These events are more difficult to handle than normal AT commands, because they are asynchronous. Handling the events means that the serial port has to be polled regularly, which can either be done in a separate thread or polled manually using a method. Using threads means increased complexity and the need for synchronization objects.

Since it is not absolutely necessary to handle the unsolicited responses, they are just discarded. They might however be handled in a future implementation.

The main command processing method is the method

```
CommandResponse ExecuteCommand(const std::string& cmd, char endlLine,
    int cmdTimeout, int dataTimeout, std::deque<std::string>& cmdOutput).
```

It processes all basic AT commands and outputs the response text as a list of strings if available.

The returned `CommandResponse` can be

| | |
|---------------------------|--|
| <code>OK</code> | the command completed successfully |
| <code>Error</code> | the command returned an error |
| <code>TimeOut</code> | the command returned no response within the timeout period |
| <code>NotConnected</code> | the serial port is not connected |
| <code>NotIdle</code> | a command is already being executed |
| <code>NoGPSData</code> | no GPS fix available (when reading GPS over the modem port) |
| <code>ReadyToSend</code> | an interactive command (e.g. sending SMS) consisting of two basic commands is ready to take the data to send |

The parameter `cmd` is the AT command string to execute.

The endl character `endlLine` is to be specified because not all commands are executed after sending a CR character (0Dh). The endl character is appended to the command string automatically.

The parameter `cmdTimeout` defines the command timeout period in microseconds. This is the maximal time command can take to complete.

The parameter `dataTimeout` defines the timeout period for receiving the response. When this period is over without receiving new data, a response is considered to be complete.

The response is a list of strings (one for each line of the response) and is saved in a double-linked list `cmdOutput`. Only the response lines relevant for parsing are returned, the rest is stripped away.

The `ExecuteCommand` method processes the commands as follows:

First, the command string is sent over the serial port, then the method tries to read data from the serial port in a loop. During each iteration, the method checks if the data timeout period has expired in case no data has been received. Every time a newline character is received, a new response string is appended to the output list.

When the data timeout has expired, the response lines are tested for the `OK` or `ERROR` keyword, and an appropriate `CommandResponse` is returned.

12.2.3 Enumerations and Data Structures

PINState:

| | |
|--------------|--|
| OK | PIN is correctly entered and active |
| PIN_Required | PIN must be entered |
| PUK_Required | PIN is locked out, PUK must be entered |
| PIN_Unknown | unable to check PIN state |

GPRSState:

| | |
|-------------------|----------------------------|
| GPRS_Disconnected | GPRS disconnected |
| GPRS_Connected | GPRS connected |
| GPRS_Unknown | unable to check GPRS state |

SMSType:

| | |
|------------|-----------------|
| Unread_SMS | unread SMS only |
| Read_SMS | read SMS only |
| Unsent_SMS | unsent SMS only |
| Sent_SMS | sent SMS only |
| All_SMS | all SMS |

SMSMessage:

| | |
|-------------|---------|
| Index | int |
| Type | SMSType |
| PhoneNumber | string |
| Timestamp | time_t |
| Text | string |

12.2.4 Interface

The following public methods are available on the GM862GPS class:

- GM862GPS ()
A default constructor
- ~GM862GPS ()
The default destructor. Frees up allocated buffer memory, and disconnects the serial port if necessary.
- bool ConnectSerial (SerialPort* serial)
Connects the GM862-GPS to a given serial port object. The serial port is automatically opened and configured for the GM862-GPS (115200 baud, 8 data bits, flow control enabled).
- void DisconnectSerial ()
Detaches the GM862-GPS from the attached serial port.
- bool SerialConnected ()
Determines if a serial port is attached to the GM862-GPS.
- CommandResponse InitModule ()
Sends basic module configuration commands (disable echo, enable new instruction set, enhanced SMS instructions).
- CommandResponse EnterPIN (const std::string& pin)
Enters the SIM PIN code. Entering extended codes like PUK is not supported.
- CommandResponse ChangePIN (const std::string& oldPin, const std::string& newPin)
Changes the SIM PIN code to a new one.

- `CommandResponse CheckPIN(PINState& state)`
Checks the state of the PIN code and outputs it. The state is either `OK`, `PIN_Required`, `PUK_Required` or `Unknown` (if command fails).
- `CommandResponse InitSMS()`
Sends SMS initialization commands (enable SMS text mode). The PIN must be entered prior to sending this command.
- `CommandResponse ListSMS(SMSType filter, std::deque<SMSMessage>& smsList)`
Queries for received SMS and outputs them as a double-linked list. If only certain kinds of SMS should be listed, a filter can be specified.
- `CommandResponse DeleteSMS(int index)`
Deletes an SMS at a given message store index.
- `CommandResponse SendSMS(const std::string& destination, const std::string& text)`
Sends an SMS text to the given destination.
- `CommandResponse SendSMS(const SMSMessage& message)`
Same as above, but instead an SMS message object is passed.
- `CommandResponse CheckGPRS(GPRSState& state)`
Checks the GPRS state and outputs it.
- `CommandResponse AttachGPRS(std::string apn, std::string address, std::string username, std::string password)`
Sets up a GPRS connection using the given APN, IP address, username and password.
- `CommandResponse DetachGPRS()`
Disables the GPRS connection.
- `CommandResponse GetGPSData(GPSInfo& gps)`
Retrieves the current GPS position (if available) and outputs it.

Part V

Centralized Data Collection

This part describes how the data collection over the internet is performed on the logical layer. It is basically independent of the physical layer described earlier. The logical layer describes how the data that is transmitted over the network is structured.

13 Data Collection Concept

13.1 Client / Server

Our concept of centralized data collection is based on a single server with multiple clients.

Within this concept, the data loggers act as independent clients which connect to the server whenever they have acquired data to transmit. [WL09] give an example why this concept is feasible. The health monitor described calls the monitoring center by itself in case of an emergency, it does not have to be queried actively from the monitoring center.

This is also practical because the data loggers can acquire data on their own, even if a connection to the server is not possible at the moment. Especially in a mobile environment, for example when cellular network technology like GPRS is used, there is a certain probability that no network is available. This can happen when the data logger is used in an environment that blocks cellular reception, for example inside buildings with thick walls or places under ground. Depending on the provider being used, there might also be a lack of network coverage in lesser populated areas.

As an alternative, the data loggers could act as servers which are contacted by the centralized server on a regular basis. The reason why we don't use this concept is the uncertainty of a connection availability. If a data logger cannot establish a network connection or, which is also a fair possibility, is just not active at the moment, the server might have to try establishing a connection over and over before succeeding.

13.2 Server Access

There are several possibilities to access a remote server. Today, many services are HTTP-based web services. One reason for that is that they can be implemented easily, since there are already existing standards for data exchange, such as XML (*extensible markup language*) or JSON (*JavaScript Object Notation*).

However, HTTP services have a disadvantage: they produce a relatively large data overhead, because they are stateless. This means that everytime a service is accessed, a full HTTP request has to be sent to the server, which can quickly result in several hundreds of bytes or even a few kilobytes of data that has to be sent. This does not even include the actual user data that is to be transmitted. Furthermore, if the user data is transmitted a "convenient" format like XML, there is an additional overhead.

This overhead is negligible considering today's high-speed internet connections. On the other hand, this is a different story when mobile connections such as GPRS are taken into account. First, a GPRS connection has an upstream bandwidth of a few KB/s at best, and the running costs depend on the amount of data being transmitted. As a consequence, it makes sense to transmit data as efficiently as possible, because it saves money and allows more data to be transmitted over a slow connection.

Therefore, our concept incorporates a server application which operates directly on TCP/IP connections and uses a custom transmission protocol specifically designed for the data acquisition system. Despite the fact that this decision makes the task of designing and implementing the solution more complex, it allows us to reduce the communication overhead to a minimum.

13.3 Data Storage

Our data acquisition system needs to store two kinds of data: first, the data collected by the data loggers has to be stored. Second, information about the data loggers and their owners is also important to provide access to the collected data.

Since the data acquisition system must support different scenarios, the database configuration must be flexible enough to support the data sets required by the scenarios.

In the proof-of-concept implementation, PostgreSQL 8.4 is used as the database system. It is an open-source database system which natively supports transactions (unlike MySQL with MyISAM tables for example). Future productive implementations should also support different database systems.

13.4 Data Evaluation

The data evaluation is difficult to generalize, because it depends entirely on the usage scenario. The only operation which can be generalized is reading the values of a specific scenario and device and filtering them by specific columns.

Therefore, only a few general mechanisms for fetching data will be discussed.

14 Database Structure

The database is divided into two parts: a general, device-independent part, and a more specific part which depends on the device or the usage scenario.

14.1 General

The general part of the database contains information about registered data logger owners, such as name, login data and which data logger devices are assigned to the owner. Information about each registered data logger device is also included.

The database tables are specified as follows:

Table “owner”:

| Column | Type | Comment |
|----------|--------------|----------------------------|
| id | INT8 | Owner ID (primary key) |
| name | VARCHAR(200) | Owner name |
| login | VARCHAR(20) | Login name |
| password | CHAR(40) | Login password (SHA1 hash) |

Table 8: database table for device owners

Table “device”:

| Column | Type | Comment |
|-----------|----------|---|
| id | INT8 | Entry ID (primary key) |
| device_id | INT8 | Device Unique ID (unique) |
| owner_id | INT8 | Owner ID (references id from table “owner”) |
| password | CHAR(40) | Device password (SHA1 hash) |

Table 9: database table for data logger devices

14.2 Specific

The scenario-specific database tables contain the data collected from the data logger devices. There is exactly one table per scenario, so each row is a data set whose columns are the different data values.

The naming scheme for the table name is **data_{ProtocolID}{ProtocolRevision}**, where *ProtocolID* and *ProtocolRevision* are hexadecimal representations of the protocol ID and the protocol Revision.

Example: `data_0201` is the table for protocol `02h` with revision `01h`.

The data columns of the data table are specified according to the data channels and sub-channels defined in the XML protocol specification in section 15.2.

The naming scheme for the column name is **{Channel}_{SubChannel}**, where *Channel* is the channel name and *SubChannel* is the sub-channel name. The data types are mapped appropriately, since the XML specification does not specify SQL data types.

There is a constraint when using certain database systems such as PostgreSQL, which don't support unsigned data types. In this case, data types that are specified as unsigned in the XML specification

must be mapped to a signed data type of higher order. For example, an unsigned 16-Bit integer will be mapped to a signed 32-Bit integer in order to allow the same range of values.

15 Data Collection Transmission Protocol

The transmission protocol handles the communication between the data logger client and the server. It consists of a set of binary data packets. Also, for each protocol there is a specification that provides the server with information how incoming data has to be processed.¹

15.1 Concept

The concept of the transmission protocol is based on two requirements:

It should be as compact as possible in order to reduce the amount of data necessary for communication to a minimum, yet the protocol must be flexible, so that the client and the server can be configured for a specific usage scenario.

In our concept, we divide the data to be acquired by the data logger into *channels* and *sub-channels*. Each channel represents a bunch of sub-channels grouped together. You could also imagine a channel as an object, where each sub-channel is a single attribute value.

Ideally, a channel consists of sub-channels which bear data that belongs together, for example the latitude and longitude of a GPS position. Only full channels will be transmitted, because the data of a single sub-channel would not be useful for evaluation. Thinking of GPS for example, there are few situations where the latitude or longitude alone would be useful.

On the server side, we would like to be independent of the server implementation, no matter what data is to be acquired. Therefore, we use a configuration mechanism that describes the structure of the data the data logger sends to the server, so that the server can process this data without the need of modifying the server's source code.

The format of choice for the configuration mechanism is XML, because it can be parsed easily, can be validated using a schema, and it is human-readable.

15.2 XML Protocol Specification

The data packets exchanged between data logger client and server are defined in the transmission protocol specification, but the payload is not. Each data logger can have its own configuration of data channels, therefore the server needs to know how to interpret the data received from the client.

The data channel configuration is specified in an XML file, which is placed in the `protocols` subdirectory of the server installation directory. Each configuration must be placed in a separate file.

15.2.1 XML File Structure

- `title` – the configuration title
- `protocol_properties` – general properties of the protocol
 - `id` – the protocol ID
 - `revision` – the protocol revision
 - `description` – a short description of the protocol
- `data_channels` – data channel instances (up to 255)

¹The transmission protocol has been developed in cooperation with the Institute of Aerosol and Sensor Technology and is developed by Benjamin Wyrsh, Michael Glettig and Matthias Krebs.

- channel – a channel definition
 - * id – the channel ID (0..255)
 - * name – the channel name
 - * preserve – data preservation (true / false)
 - * subchannel – a sub-channel instance
 - name – the sub-channel name
 - bytes – the data size
 - format – the value data type

An example XML file is given in appendix section [B](#).

15.2.2 Identification

The protocol specification is identified by its protocol ID and revision. The protocol ID is supposed to be unique, but if the channel configuration changes over time, the database tables, which might already contain data, should not be modified. This is why a revision number is added to the protocol ID, so that multiple versions of the same protocol can co-exist.

Whenever a channel or sub-channel is modified, it is mandatory to increase the revision number. Otherwise, already existing database tables do not match the configuration. If the revision number is kept, the corresponding database table has to be removed and re-created.

15.2.3 Supported Data Types

Currently, only a few standard numerical data types are supported:

- int8 – signed 8-Bit integer
- uint8 – unsigned 8-Bit integer
- int16 – signed 16-Bit integer
- uint16 – unsigned 16-Bit integer
- int32 – signed 32-Bit integer
- uint32 – unsigned 32-Bit integer
- int64 – signed 64-Bit integer
- float – 32-Bit floating point number

Additional data types can be implemented if needed.

15.2.4 Data Preservation

In order to save bandwidth, data loggers don't always have to transmit all data channels in a data packet. In this case, the default behaviour is to insert a NULL value into the database. However, a NULL value is not always a good choice, since there might be a need for a valid value to be able to evaluate the data. Imagine a NULL timestamp, this would result in data rows not shown when selecting them by timestamp.

To circumvent this problem and force a valid data value, the *preserve* flag is introduced in each channel definition. If the flag is set for a channel, all sub-channel values will be cached when the channel data is transmitted. Until new data is transmitted, the cached values are put into the database. Otherwise, the sub-channel values are set to NULL when a channel is omitted.

15.3 Specification of the Transmission Protocol

This section describes the structure of the data packets used in the transmission protocol.

15.3.1 Packet Structure

Each packet is divided into three sections:

1. **Packet Header**
The packet header provides packet identification and describes the size of the packet
2. **Data Header**
The data header is a optional header containing information about the data payload, if necessary
3. **Data Payload**
The data payload contains the user data

| Part | Header | Data Header | Data Payload |
|--------|---------|-------------|----------------------|
| Length | 5 bytes | variable | variable |
| Offset | 0 | 5 | 5 + data header size |

Table 10: General communication packet structure

Packet Header Layout

| Field | Packet ID | Data Header Size | Data Payload Size |
|--------|-----------|------------------|-------------------|
| Length | 1 byte | 2 bytes | 2 bytes |
| Type | u_int8 | u_int16 | u_int16 |
| Range | 0..255 | 0..65535 | 0..65535 |

Table 11: the packet header layout

The packet header layout is the same for all communication packets. The packet ID identifies the packet type. The data header and payload lengths are variable and depends on the packet type. Since they are variable, their length is specified in the header in order to determine the correct offset within the packet.

The data header and payload have a maximal length of 64 KB each.

Data Header Layout

The data header has no predefined layout, it is basically just an array of bytes. Its length is defined in the *data header size* field of the packet header. Some packet types use the data header to describe the structure of the data payload, so the receiver can process it correctly.

Data Payload Layout

Like the data header, the payload is packet-specific and has no predefined layout. Its length is defined in the *data payload size* field of the packet header.

15.3.2 Communication Packets

The following communication packets are currently specified.

C_CTRL

The *C_CTRL* packet is sent by the client to the server in order to control the connection (start, stop). It has a packet ID of 80h. There is no data header, so the data header size is always 0. The data payload consists of exactly one byte, which is the control command ID.

Valid control command IDs are:

| Value | Name | Description |
|-------|-------|--|
| 00h | START | Initializes the connection with the server. It will respond with an <i>Authent_Start</i> status response to signal the signal it is ready to receive the authentication. |
| FFh | STOP | Terminates the connection with the server. The client is logged off. |

Table 12: C_CTRL command IDs

S_STATE

The *S_STATE* packet is sent by the server to the client to show its communication state. It has a packet ID of 00h. There is no data header, so the data header size is always 0. The data payload consists of exactly one byte, which represents the server state.

Valid server state codes are:

| Value | Name | Description |
|-------|---------------|---|
| 00h | Prot_Error | The protocol used by the client is not supported |
| 01h | Authent_Error | The authentication of the client failed. |
| 02h | Busy_Error | The server is currently unable to receive data. |
| 80h | Authent_Start | The server is ready to receive the client's authentication. |
| 81h | Authent_Ok | The client was successfully authenticated. |

Table 13: S.STATE response codes

C_AUTHENT

The *C_AUTHENT* packet is sent by the client to the server in order to authenticate itself. It has a packet ID of 81h. The data header is 1 byte long, this byte defines the length of the password in the payload.

The payload is structured as follows: The device ID can be an arbitrary serial number, but it must be unique within the service provided by the data collection server. The device password can be up to 255

| | | | | |
|--------|-----------------------|------------------------|-------------------|-------------------|
| Field | Device ID | Password | Protocol ID | Protocol Revision |
| Length | 8 bytes | variable | 1 byte | 1 byte |
| Offset | 0 | 8 | 8+Password Length | 9+Password Length |
| Type | int64 | string | u_int8 | u_int8 |
| Range | $-2^{63}..2^{63} - 1$ | ASCII codes 32..127 | 0..255 | 0..255 |

Table 14: C_AUTHENT payload structure

characters long. It is compatible to C strings, but not zero-terminated. In order to provide maximal compatibility, only printable standard ASCII characters are supported. This avoids problems with systems that use variable character sets such as UTF-8, which might interpret special characters differently.

C_TX_DATA

The *C_TX_DATA* packet contains the actual data acquired by a data logger, which is sent to the server. It has a packet ID of 82h. The data header is used to specify which channels are contained in the payload, whereas the payload contains the serialized channel data.

The data header length is a multiple of 2 bytes, because for each channel its ID and data length are specified. Since these fields are each 1 byte long, there is a maximal number of 256 channels, and each channel can have up to 255 bytes of data. This pattern is repeated for each channel that is contained in

| | | |
|--------|------------|-------------|
| Field | Channel ID | Data length |
| Length | 1 byte | 1 byte |
| Type | u_int8 | u_int8 |
| Range | 0..255 | 1..255 |

Table 15: C_TX_DATA data header structure

the packet.

The payload is basically unstructured serialized data. The channel data is put in the same order as the channels specified in the data header. The order of the sub-channel data within each channel is ordered according to the XML specification described in section 15.2.

15.3.3 Example Communication

The following diagram (Figure 26) shows an example communication sequence between a data logger client and a server.

In this example, the client authenticates itself and sends three data packets before disconnecting.

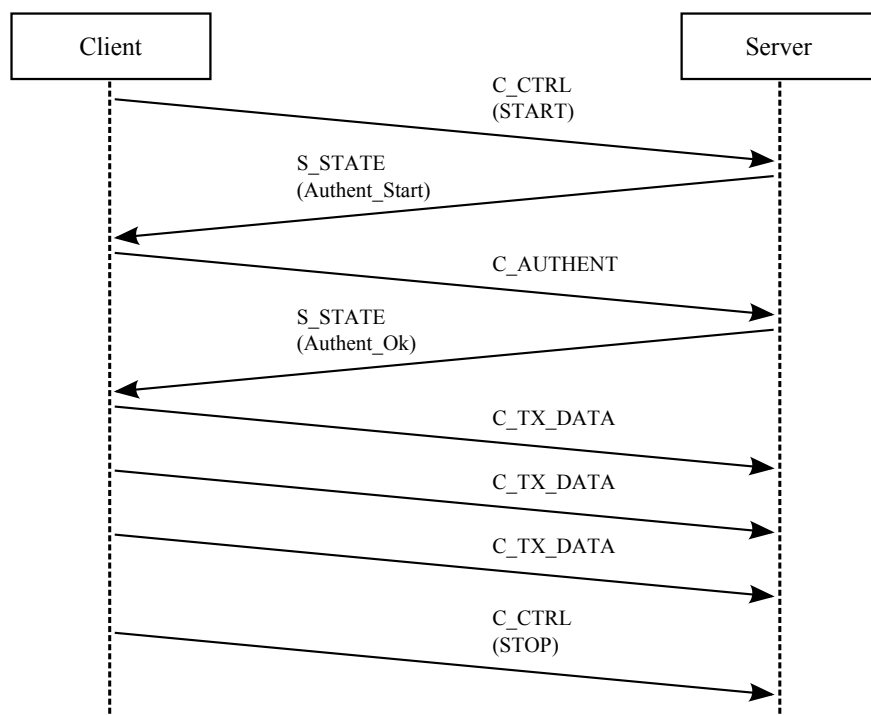


Figure 26: An example of the data logger transmission protocol

16 Data Collection Server

The data collection server handles connections from the data logger clients and processes the data acquired by them. The server puts the received data into an SQL database.

16.1 Concept

The data collection server is an application listening on a TCP port.

Whenever a data logger connects to the server, the connection runs in a new thread while it is open. This is a simple approach of allowing multiple concurrent connections, since there might be a number of data loggers trying to connect at the same time. This approach is good enough for a small number of clients, as mentioned by [YAL10]. As soon as a large number of connections is required, different concepts have to be thought about.

Within each connection thread, the client and server use the transmission protocol specified in Section 15.

To be flexible, different protocol specifications can be loaded at boot time, upon which the database layout is generated. The different protocol specifications are transformed into a data structure residing in memory, which is accessed every time a data logger transmits sensor data. The data is then processed according to the protocol specification matching the protocol the data logger uses.

The advantage of this flexible approach is that the server implementation does not have to be modified when a new data logger device is developed or the data it acquires is changed.

16.2 Protocol Data Structure

The protocol data structure is defined like in the XML specification and implemented according to the UML diagram in Figure 27.

Protocols, channels and sub-channels are defined as separate objects, with the following relations:

- the server knows a list of protocols
- each protocol contains a list of channels
- each channel contains a list of sub-channels
- each sub-channel represents a single data value

The object attributes correspond to the attributes of each XML tag in the protocol specification.

It is important that, when a client connects to the server, the correct protocol instance that is supposed to process the incoming data is found quickly. This is why the protocol list in the server is implemented as a hashtable, whose key is calculated based on the protocol ID and revision:

$$key = ID \cdot 256 + revision$$

Because the protocol ID and revision are 8-bit integers, this results in a unique key for each protocol.

The channel list inside a protocol instance is also implemented using a hashtable, the key being the channel ID. The reason for this is that the data header of the C.TX.DATA packet does not demand a specific channel order.

This is different for the sub-channels, as their data is serialized exactly in the order of their appearance in the XML specification. The order is mandatory, so a list of sub-channel instances suffices.

Each protocol instance has a database connection object attached, as well as prepared SQL statements for inserting new data. The sub-channel instance know which parameter within the statement belongs to their data value, so they update all statement parameters when they fetch data. When all sub-channels have been processed, the data is inserted into the database. For data preservation, each sub-channel contains a hashtable whose key is a device ID and the value an instance of the sub-channel's data type. It is implemented as a hash table because each device must preserve its last data value.

16.3 Implementation

The data collection server is designed as a stand-alone application implemented in Java.

16.3.1 Server Software Structure

The data collection server software is structured as seen in Figure 27. The classes are grouped into packages:

- The `ch.fhnw.datalogger.server` package contains the basic server classes needed to handle connections.
- The `ch.fhnw.datalogger.server.xml` package contains the factory for loading XML protocol specifications.
- The `ch.fhnw.datalogger.server.protocol` package contains the protocol data structure classes.

16.3.2 Configuration and Start-Up

The server configuration is stored in a Java property file named `server.properties`. This file must be placed in the server installation directory and is loaded at server start-up.

The properties available are:

| | |
|-------------------------|---|
| <code>TCPPort</code> | the TCP port number the server listens on |
| <code>Timeout</code> | the server socket timeout in milliseconds |
| <code>DBHost</code> | the database host name |
| <code>DBPort</code> | the database port |
| <code>DBDatabase</code> | the database name |
| <code>DBUser</code> | the database user name |
| <code>DBPass</code> | the database password |

The protocol specification XML files must be placed in the `protocols` subdirectory of the server installation directory, they are loaded at start-up like the server configuration.

The `ProtocolFactory` class loads and parses each XML file in the `protocols` directory using a simple DOM parser. There is no schema validation yet, it will be included in a future implementation.

For each protocol loaded successfully, a database table is created. Note that the table is only created if it does not exist yet. This is the reason why the revision of a protocol must be increased when the channel configuration is modified, as modifying the original XML specification would result in a conflict with the database. Also, the table cannot be modified as there might already be data inside, and altering the table

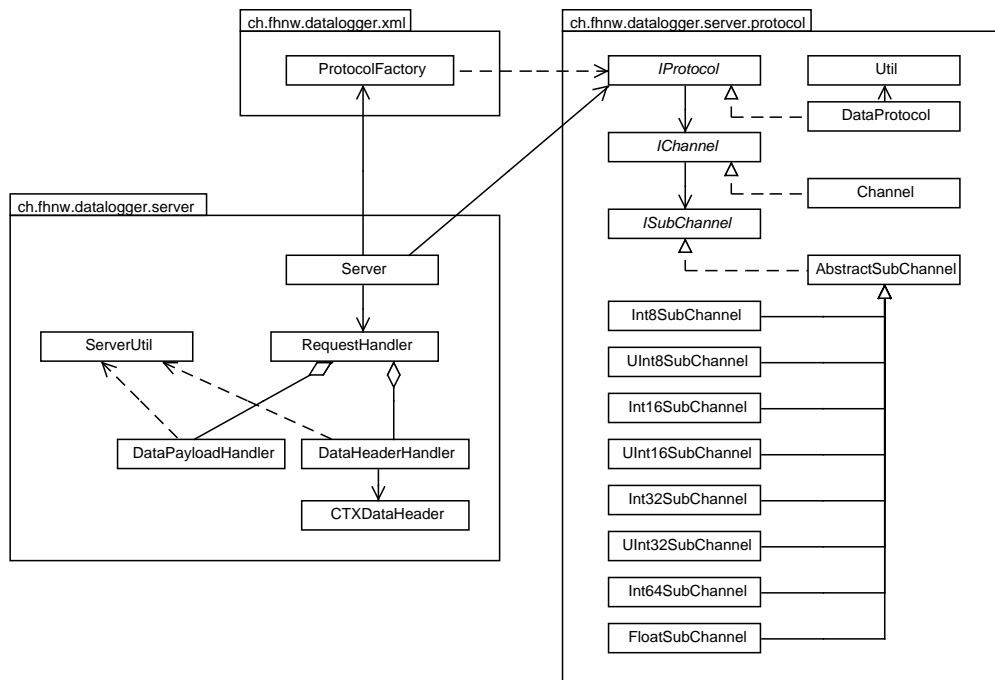


Figure 27: The UML class diagram of the data collection server

structure might result in data rows lacking essential data afterwards. In a future implementation, the tables could be verified against the protocol specification in order to avoid mismatching specifications.

16.3.3 Connection Handling

In the main application class `Server`, the server socket waits for connections in an endless loop.

Each time a client opens a connection, a new `RequestHandler` instance is created and executed within a new separate thread. The request handler tries to read a full communication packet (header and data). After reading the packet, its packet ID is checked and the appropriate command is handled. The data consisting of a data header and data payload is then passed to a `DataHeaderHandler` and `DataPayloadHandler` which further process the packet.

When the client initializes the connection, it must first announce which protocol ID and revision it is going to use. If the protocol is not supported by the server, the connection is terminated by the server and the thread is destroyed. Otherwise, the client can send its authentication request. If the authentication fails, the connection is also terminated.

If the authentication is successful, the client can start sending `C_TX.DATA` packets, until there is a connection failure or the client wishes to end the connection.

16.3.4 Data Processing

When a `C_TX.DATA` packet arrives, its data header and payload are passed to the appropriate protocol instance.

First, the protocol instance iterates through all channels contained in the data header and updates the data values of all sub-channels. Channels not found in the data header have their sub-channel values set to NULL, or, if data preservation is enabled for the channel, the last data value is kept.

The order of channel IDs in the data header also represents the order of the channel data in the data payload. The data size specified for each channel determines the number of bytes to read from the data payload. If this size does not match the server's protocol specification, the channel is skipped for safety reasons.

After all channels have been processed, the prepared insert statement is executed on the database and the data is inserted.

17 Evaluation of the Data

The evaluation of the collected data is highly specific to the usage scenario and thus cannot be generalized so easily.

In the Scintilla usage scenario, we are mostly selecting whole working cycles in order to analyze the usage profile of power tools.

The following SQL statement selects all data of the working cycle with ID 1 of the device with ID 3869. The selected data is ordered by their timestamp in descending order.

```
SELECT * FROM data_0200 WHERE device_id=3869 AND workingcycle_id=1 ORDER BY  
timestamp_microseconds DESC
```

What is common to all scenarios is that there must be at least one data column that allows the ordering of the data sets and useful association of data.

When data is collected constantly over time, a good approach is a timestamp column, because it allows sorting the data sets in the order they were inserted into the database. It also allows the extraction of data that was collected within a certain timespan.

There are different concepts of evaluating the data. A web-based application can be used, which is feasible when the data should be accessed from a single point and be accessible from anywhere. A desktop application might also be useful in case the data has to be visualized using complex graphics, a process which needs a lot of computational power.

Efficient caching of the selected data is important, because when there are millions of data sets in the database, selecting specific data sets can take some time, as the database tables have to be searched upon each access. Creating indexes on certain columns or caching data locally after reading from the database can speed up the process and save resources.

Part VI

Data Logger

The data logger device is the heart of the data acquisition system. This part describes the embedded hardware the data logger is built of, and how the embedded software is designed.

18 Data Logger Hardware

18.1 Core System

The core system of the data logger is an embedded computer system that contains a CPU core, memory, persistent data storage and basic I/O functionality to interact with the other system components.

Several embedded system candidates have been evaluated for their functionality provided and their price.

In the end, the Eddy-CPU module from SystemBase has become the system of choice, because a single module costs only 60 €, whereas other competitors were often full developer boards not available as small modules or more expensive. The Eddy-CPU module can however be used for development and productive use, as it can be stacked on top of an other PCB.

The Eddy-CPU module shown in Figure 28 is a small ARM-based CPU module with the following features:

- Atmel AT91SAM9260 ARM9 CPU at 210 MHz
- 32 MB SDRAM
- 8 MB of flash memory
- 4 serial ports
- USB 2.0 full-speed bus
- Fast Ethernet interface onboard
- dimension of 25 mm x 48.5 mm

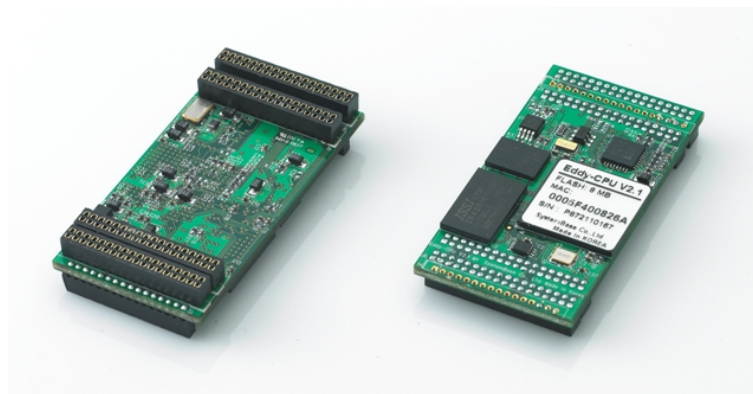


Figure 28: The Eddy-CPU module

By default, the Eddy-CPU module comes with a pre-installed embedded Linux based on Kernel 2.6.21.

The CPU module alone is not very useful for development, because all I/O is only available on PCB pin headers. This is why SystemBase offers the Eddy-DK developer board, which is shown in Figure 29. The Eddy-DK developer board provides all I/O that are supported by the Eddy-CPU module, for example RS232, RS485 and USB ports.

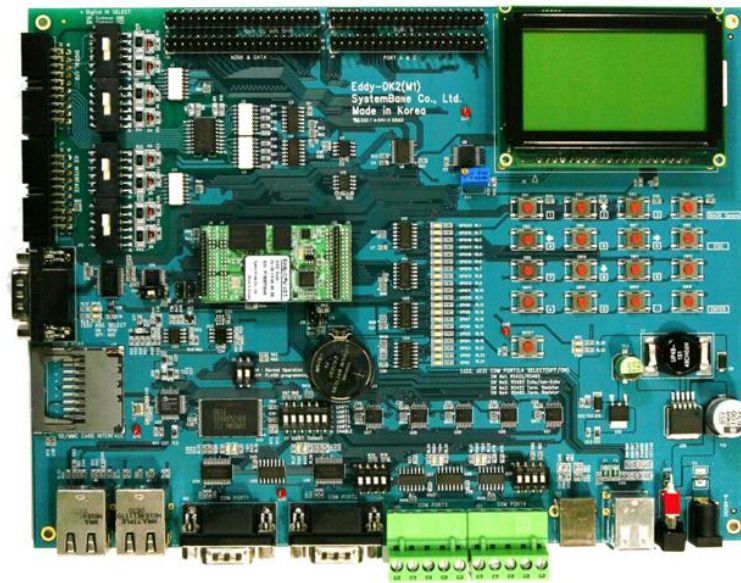


Figure 29: The Eddy-DK developer board

The Eddy-DK developer board can be connected to development PC through a LAN cable, which is also used to transfer compiled programs and for remote debugging. A debug serial port can be connected to a terminal emulator in order to watch program output during debugging.

The Eddy-DK comes with its own integrated development environment named *LemonIDE*. This IDE is a modified version of Eclipse 3.2 and provides some extra plug-ins specifically for the Eddy-CPU module. The IDE can be used for C and C++ development and is available for Windows and Linux. The GNU toolchain for ARM processors is used for building the binaries from source.

18.2 Connectivity

The Eddy-CPU module has a total of four serial ports that can be used to communicate with external devices.

On the Eddy-DK developer board, the first two serial ports are provided as RS232 D-Sub connectors, while the other two serial ports are provided as RS485 screw-type terminals.

We use two serial ports for GSM and GPS access, one serial port to access sensor interfaces (RS485), and one serial port for future remote configuration.

18.3 Local Data Storage

The Eddy-CPU module (or the Eddy-DK developer board, respectively) provides two kinds of storage media to be attached: SD or MMC cards and USB sticks. It makes sense to use a removable medium inside the data logger device, because it can easily be replaced if a bigger capacity is needed or in case the flash memory is damaged. If an integrated flash memory chip is used, it cannot be replaced in case of damage.

The SD or MMC card is a bit difficult to use with the Eddy-CPU module. The SD/MMC slot available on the Eddy-DK board is connected by an USB to SD/MMC bridge chip, because the SD/MMC interface

provided by the ARM CPU on the Eddy-CPU module cannot be used, as it is already used internally by the program flash memory.

The USB interface, however, is available and can be used directly, requiring only a 5 V power supply. This why the data logger uses USB memory to store data locally. Using a USB stick is just as easy as using an SD card, and there are USB sticks on the market which are hardly bigger than the USB port itself. The price is also comparable.

18.4 Communication

The Eddy-CPU module has a built-in Ethernet interface, but since we are designing a mobile data logger, it is not used. Instead, we use the GM862-GPS GSM module as a communication interface, which provides an internet connection over GPRS and processes SMS for remote configuration or status messages. The GM862-GPS also provides the GPS receiver that is used to locate the data logger device.

One of the serial ports of the Eddy-CPU module is used to access the GM862-GPS modem, and one is used to retrieve GPS data. During the development process, we connect the GM862-GPS base board, as described in Section 11.3.1, to the Eddy-DK developer board.

The GM862-GPS modem port is connected to the first serial port of the Eddy-CPU module, and the GPS port is connected to the second serial port.

19 Data Logger Embedded Software

Most of the data logger functionality is realized in software, as the data logger device itself is mainly a small embedded computer system that only provides processing power and hardware interfaces to interact with external components.

19.1 Fundamentals

19.1.1 Overview of the Eddy-CPU Software Features

The Eddy-CPU module is shipped with the Lemonix operating system, which is an Eddy-CPU-specific Linux distribution. The current version is based on Kernel 2.6.21, the sources of the whole operating system can be downloaded for development purposes.

Additionally, a C library featuring some Eddy-CPU-specific real-time functions is provided. These functions include serial port access, GPIO and system configuration. This library is proprietary, no sources are provided.

19.1.2 C++ Development

By default, the Eddy-CPU module only supports the development in C, which is a common practice in the development of embedded software. It is often claimed that higher level programming languages such as C++ and Java are not efficient enough to be used on embedded hardware.

Since we develop the data logger application for an embedded Linux operating system, we are quite flexible when choosing which libraries to use. Instead of C, we want to use C++ to develop the application. Despite being slightly more complex, C++ has a few advantages over C. It allows us to create an object-oriented design instead of a bunch of functions interacting with each other. By using class inheritance, it is also easier to extend existing functionality.

The Eddy-CPU development tools already support C++, at least theoretically. The GNU C++ compiler and libraries are included, but the makefiles used to compile the applications have to be modified first.

A short version of the original makefile looks like this:

```
CROSS          = /opt/lemonix/cdt/bin/arm-linux-
LDLFLAGS       += -L/opt/lemonix/cdt/lib -L/opt/lemonix/cdt/bin
IFLAGS         += -I/opt/lemonix/cdt/include -I./include
CFLAGS         =          -O2 -Wall -Wno-nonnull
DEST           = ../../ramdisk/root/sbin
DEST_ETC       = ../../ramdisk/root/etc

CC             =          $(CROSS)gcc
STRIP          =          $(CROSS)strip
AR             =          $(CROSS)ar

TARGET         =          datalogger

LIBS           =          -lrt ./SB_APIs/SB_APIs.a

all : $(TARGET)
```

```

datalogger : datalogger.o
    rm -f $@
    $(CC) $(CFLAGS) $(LDFLAGS) $(IFLAGS) -o $@ $(LIBS)
    $(STRIP) $@

clean:
    rm -f *.bak *.o

release:
    cp -f $(TARGET) $(DEST)
    cp -f sb_default_config $(DEST_ETC)

```

In order to support C++, we add the C++ compiler command and options. Additionally, compiler targets for C and C++ are necessary to distinguish between C and C++ source files.

```

CROSS          = /opt/lemonix/cdt/bin/arm-linux-
LDFLAGS        += -L/opt/lemonix/cdt/lib -L/opt/lemonix/cdt/bin
IFLAGS         += -I/opt/lemonix/cdt/include -I./include

CFLAGS         =          -O2      -Wall -Wno-nonnull
CXXFLAGS       =          -O2      -Wall

DEST           = ../../ramdisk/root/sbin
DEST_ETC       = ../../ramdisk/root/etc

CC             =          $(CROSS)gcc
CXX           =          $(CROSS)g++
STRIP         =          $(CROSS)strip
AR            =          $(CROSS)ar

TARGET        =          datalogger

LIBS          =          -lrt ../../Eddy_APPS/SB_APIs/SB_APIs.a

all : $(TARGET)

datalogger : datalogger.o
    rm -f $@
    $(CXX) $(CXXFLAGS) $(LDFLAGS) $(IFLAGS) -o $@ $^ $(LIBS)
    $(STRIP) $@

%.o : %.cpp
    $(CXX) $(CXXFLAGS) $(IFLAGS) -o $@ -c $<

%.o : %.c
    $(CC) $(CFLAGS) $(IFLAGS) -o $@ -c $<

clean:
    rm -f *.bak *.o

```

```
release:
    cp -f $(TARGET) $(DEST)
    cp -f sb_default_config $(DEST_ETC)
```

With these modifications, C and C++ applications can be compiled. The introduction of the $\$^{\wedge}$ modifier allows the inclusion of multiple object files, which is essential when the C++ classes are implemented in separate files. The $\%.o$ targets are used to compile each source file into an object file and distinguish between C and C++ sources.

We are now able to compile C++ applications, but they cannot be executed on the Eddy-CPU module. The reason is that the Eddy-CPU does not ship with the standard C++ library *libstdc++* in the default firmware. Therefore, we must create a custom firmware that includes the C++ library.

This is achieved by copying the file `libstdc++.so.6` from the ARM toolchain's library directory to the `ramdisk/root/lib` directory. The RAM disk file system is provided by the Eddy-CPU development tools. It also includes a makefile to create a firmware image.

After the new firmware is uploaded to the Eddy-CPU module and the module is rebooted, C++ applications can be executed.

19.2 Eddy-CPU Software API Wrapper

The Eddy-CPU-specific API provides real-time functions used to access the serial ports. The problem is that this API is designed to be used in C applications that only consist of a single source file, its header files are badly structured and also contain source code. When the header files are used in C++ classes or included in multiple source files, the compiler throws an error because it finds multiple declarations of the same functions.

This is why a wrapper is introduced which encapsulates the required Eddy-CPU functions, this wrapper can be included in multiple source files.

Currently, the wrapper `EddyAPIWrapper.h` encapsulates the serial port functions:

- `int Eddy_OpenSerial(int portNum)`
- `void Eddy_SendSerial(int handle, const char* src, int len)`
- `int Eddy_ReadSerial(int handle, char* buf, int maxLen, int waitms)`
- `void Eddy_InitSerial(int handle, char speed, char dps, char flow)`
- `void Eddy_SetRts(int handle, int dir)`

The interface of these functions is identical to the original Eddy-CPU functions, despite the prefix being changed from `SB_` to `Eddy_`.

19.3 Serial Port Implementation

The serial ports are used by different components of the data logger, such as the sensor access library, the GPS receiver and the GSM module. An RS485 and an RS232 implementation are required, as there are small differences in the timing and connection handling.

To allow more than one implementation, the interface *SerialPort* is introduced. The RS232 and RS485 implementations are derived from this interface, which makes them interchangeable.

- `bool Open()`
- `void Close()`
- `bool IsOpen()`
- `bool Send(const char* buf, int numBytes)`
- `int Receive(char* buf, int maxBytes)`
- `bool Configure(int baudrate, char dataBits, char stopBits,
char parity, char handshake)`
- `void SetDirection(int dir)`

The interface provides the standard methods for serial port access. The `SetDirection` method allows to set an RS485 half-duplex port to transmit or receive. It is irrelevant for the RS232 implementation, as RS232 is always bi-directional.

Two implementations are provided: `EddyRS232Port` and `EddyRS485Port` use the Eddy-CPU API functions, while `PosixRS232Port` and `PosixRS485Port` use the standard C functions provided by the Linux system API. In the data logger application, the Eddy-CPU implementation is used, because it provides better timing due to the real-time support.

19.4 Interfacing with the Sensors

The sensor interface is accessed through an RS485 interface, which is provided on the Eddy-DK developer board on serial port 3. The Versatile Sensor Protocol C++ implementation described in Section 9.7 is used by the datalogger software to control the RS485 interface.

For the Scintilla usage scenario, sensor commands for the TC77, MMA7455L, ADE7753 and the RPM counter are derived from `VSReadSensorValueCommand` and `VSWriteSensorConfigCommand`.

19.5 Interaction with the Server

The data logger software interacts with the data collection server through a TCP/IP connection, which is provided by a socket implementation on the Telit GM862-GPS GPRS module.

The data logger client implementation is a C++ class `DataLoggerClient` that implements the data collection transmission protocol described in Section 15. It uses the GM862-GPS via its `ISocket` interface, thus being independent of the socket implementation.

DataLoggerPacket Data Structure

This data structure contains the data of a communication packet, according to the packet specification in Section 15.3.1.

- `ID: PacketID`
- `DataHeaderSize: unsigned short`
- `PayloadSize: unsigned short`
- `DataHeader: char*`
- `Payload: char*`

DataLoggerClient Class

The `DataLoggerClient` class provides methods to manage a TCP/IP connection to a data collection server.

- `DataLoggerClient (ISocket& socket)`
Instantiates a data logger client using the given socket.
- `~DataLoggerClient ()`
Destroys the data logger client object and terminates the connection if still open.
- `bool ConnectToServer (std::string address, int port, unsigned char protocolID, unsigned char protocolRevision, long long int deviceID, std::string password)`
Establishes a connection to the server at the given IP address and port. A device ID and password must be specified for authentication. The method returns true if the client is authenticated correctly, false otherwise.
- `void DisconnectFromServer ()`
Terminates the current connection.
- `bool IsConnected ()`
Checks if the client is connected to a server.
- `bool SendData (const char* data, int len)`
Sends a C.TX_DATA packet in its raw form to the server.

19.6 Caching Acquired Sensor Data

During a working cycle, the acquired sensor data is cached on the data logger device before it is transferred to the data collection server. This method ensures that no acquired data is lost, even if there is no GPRS connection and the data logger is turned off.

The data is stored in a FIFO or queue data structure. For reasons of simplicity, the data is stored in a format that can be sent to the data collection server without further processing. This is why the data is stored as a binary stream of C.TX_DATA packets, which are described in Section 15.3.2. These binary packets correspond to the data packets that are meant to be transferred to the server, so they do not have to be converted first.

If a custom data structure or database was used to cache the data, it would have to be converted to the custom format after acquisition, and again when it is transferred to the server.

The current proof-of-concept implementation uses a Linux FIFO handle, which only caches the data in memory. This means that if the data logger is turned off, all data that has not been transferred to the server is gone.

The `LinuxFIFO` class provides the following methods:

- `LinuxFIFO (const std::string& fileName)`
Instantiates a Linux FIFO with the specified file name.
- `~LinuxFIFO ()`
Destroys the Linux FIFO object. If the file handle is open, it is closed automatically.
- `bool Open ()`
Opens the FIFO handle. If the file system entry does not exist, it is created using Linux I/O functions. If the FIFO is opened successfully, true is returned. Otherwise false is returned.

- `void Close()`
Closes the FIFO handle.
- `bool Enqueue(char b)`
Puts one byte into the FIFO. Returns true if successfully inserted, false otherwise.
- `bool Enqueue(const char* src, int size)`
Puts the specified number of bytes into the FIFO. Returns true if successfully inserted, false otherwise.
- `bool Dequeue(char* dst, int size)`
Gets the requested number of bytes from the FIFO and puts them into a byte buffer. If there is enough data in the FIFO, true is returned. Otherwise, no data is removed from the FIFO and false is returned.
- `int GetSize()`
Gets the number of bytes currently in the FIFO.

A future implementation will replace the Linux FIFO by a persistent FIFO implementation which automatically synchronizes its data with a file residing on a local USB storage medium.

19.7 The Data Logger Application

The data logger application is a single C++ application running on the Eddy-CPU module, which merges all the software components into one application forming the data logger.

19.7.1 Tasks

The data logger application uses multiple worker threads, because it has to perform different tasks in a parallel way, or in reality, in a quasi-parallel way, as the Eddy-CPU module has only one CPU core.

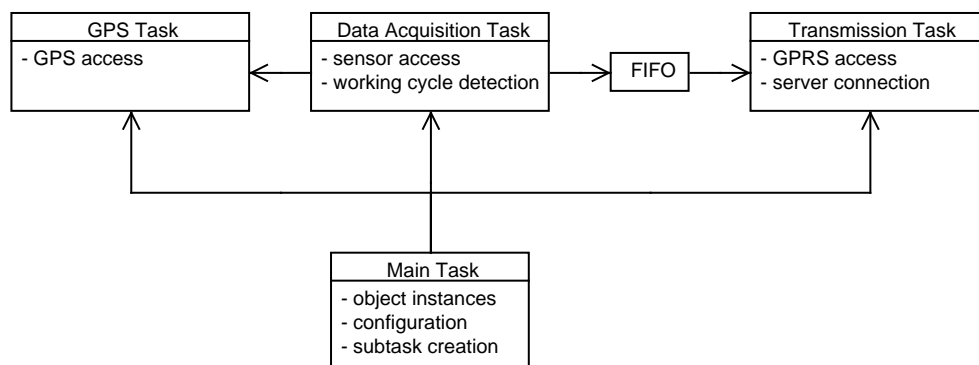


Figure 30: The data logger software tasks

The data logger application consists of four main tasks, as seen in Figure 30. The main task instantiates all objects, loads the data logger configuration and starts the three sub-tasks. These sub-tasks run indefinitely, as the data logger application is not supposed to terminate.

The data acquisition task periodically reads out the sensors and decides when it has to record data. This data is then put into a FIFO for local caching.

The GPS task is a separate task that constantly retrieves the current GPS position, which is saved in memory. The data acquisition task accesses this GPS data when it has to store the current location.

The transmission task manages the GPRS connection and establishes a connection to a data collection server, when there is sensor data waiting in the FIFO.

19.7.2 Thread Implementation

The data logger tasks are implemented as separate threads. The POSIX thread library *pthread* provides C functions that implement threads. We would like to implement them more user-friendly, using an object-oriented approach. This is why a C++ wrapper is developed.

The `Thread` class is an abstract base class that encapsulates *pthread* functions. The interface is defined as follows:

- `bool Start()`
Creates a new thread that runs the thread method.
- `void Join()`
Waits for the thread to finish.
- `virtual void Run() = 0`
The actual thread method which is to be implemented by concrete thread objects.

Actual thread implementations must override the `Run()` method, this is the method that is executed when the thread is started.

In order to work with *pthread* functions, a static protected method `void* redirectPthreadRun(void*)` is defined which provides a function pointer to the `pthread_create` function and calls the `Run()` method internally, since the thread object itself is passed as a parameter to the `redirectPthreadRun` method.

A class derived from the basic `Thread` class is the `TimerThread` class. It provides a periodic timer task based on a thread and implements the additional methods:

- `TimerThread(long us)`
Instantiates the timer task with a given microsecond interval. The actual granularity of the interval depends on the hardware and operating system beneath.
- `void Stop()`
Stops the timer task after the next tick.
- `virtual void Tick() = 0`
This method is called every time the internal timer expires. It is to be implemented by concrete timer tasks.

19.7.3 Data Logger Software Structure

The data logger software is structured as described in the UML class diagram in Figure 31.

Most basic objects, such as the serial ports and mutexes, are instantiated in the main application and then passed to the task objects as they are instantiated. The reason for this is that some objects, for example the mutexes and the data logger configuration, are shared between the task objects.

An additional feature of the GPS task is system clock synchronization. The Eddy-CPU module bears an RTC chip, but manual date and time setting is tedious. An advantage of GPS is that the current date and time are sent with each record, so after booting the data logger device, the first GPS record received is used to synchronize the system clock with the GPS date and time.

This ensures that the data logger is always set to the correct date and time, which is important because all sensor data collected is tagged with a timestamp.

19.7.6 Transmission Task

The transmission task is implemented in the `TransmissionThread` class.

Its duty is to manage a GPRS connection using the GM862-GPS module, and when sensor data is available in the FIFO, the task reads this data and opens a connection to a data collection server in order to transfer the sensor data.

After starting up, the GM862-GPS module is initialized first. When it is initialized, the thread periodically checks if there is data in the FIFO. If at least one complete `C_TX_DATA` packet is in the FIFO, the transmission thread tries to open a socket to the data collection server using the data logger client implementation and, if successful, transfers the data from the FIFO.

In case there is no GPRS connection at the moment, the thread sleeps for a few seconds and tries again.

19.7.7 Configuration

When the data logger application starts, the configuration values must be loaded. In the current proof-of-concept implementation, these values are stored in a `DataLoggerConfig` object and hard-coded into the main application.

In a future implementation, the configuration is loaded from EEPROM or flash memory instead, so it can be changed at runtime.

The `DataLoggerConfig` provides the following configuration values:

| | |
|-----------------------------------|--|
| <code>SensorPort</code> | the serial port number of the RS485 sensor interface |
| <code>GPSPort</code> | the serial port number of the RS232 GPS interface |
| <code>GSMPort</code> | the serial port number of the GM862-GPS module |
| <code>SIM_PIN</code> | the SIM PIN code as string |
| <code>GPRS_APN</code> | the GPRS access point name |
| <code>GPRS_IP</code> | the GPRS IP address |
| <code>GPRS_User</code> | the GPRS username |
| <code>GPRS_Password</code> | the GPRS password |
| <code>HighThresholdCurrent</code> | the minimal current to detect a new working cycle |
| <code>LowThresholdCurrent</code> | the lower current limit of a working cycle |
| <code>UMax</code> | the maximal voltage to measure |
| <code>IMax</code> | the maximal current to measure |
| <code>IdleSensorInterval</code> | the sensor read-out interval in idle mode (microseconds) |
| <code>ActiveSensorInterval</code> | the sensor read-out interval in active mode (microseconds) |
| <code>GPSInterval</code> | the GPS read-out interval in microseconds |
| <code>ServerIP</code> | the data collection server IP address as string |
| <code>ServerPort</code> | the data collection server TCP port number |
| <code>DeviceID</code> | the device ID of the data logger |
| <code>DevicePassword</code> | the device password of the data logger |
| <code>WorkingCycleFile</code> | the file to save working cycle IDs persistently |

20 Data Logger Miniaturization

This section describes a concept of creating a data logger prototype that is as small as possible.

20.1 Two-Layer Concept

When trying to miniaturize the data logger, we must first specify which components are required on the PCB.

- a small power supply
- a 3.3 V and a 3.7 V voltage regulator
- an Eddy-CPU module as the core system
- an USB port to connect a memory stick
- an RS485 transceiver for the sensor interface
- a GM862-GPS module
- GSM and GPS antenna for the GM862-GPS module
- an RS232 for PC connectivity
- an RS232 D-Sub connector
- a screw-type terminal or RJ-11 connector for RS485

Suggestions for the actual components are:

- Traco Power TMLM04105 embedded switched power supply (5 V, 0.8 A)
- LM1117 voltage regulators for 3.3 V and 3.7 V
- a MAX3232 RS232 transceiver
- a MAX3485 RS485 transceiver
- antennas printed directly on the PCB, or small external PCB antennas with MMCX connector

The design of the PCB depends on the size of the components being used. The GM862-GPS module measures 44 mm x 44 mm, which is rather large for a single module. Also, the Eddy-CPU module cannot be split up. Finally, the power supply is the most critical part in terms of size, even a very small switched power supply delivering 5 W measures approximately 35 mm x 25 mm.

If all components were placed on a single PCB, its surface would be rather large and not resulting in a compact design. This is why we go for a two-layer concept as seen in Figure 32, which consists of a base board and a communication board, which is mounted on top of the base board.

The base board contains the power supply, core system, an RS232 and an RS485 port, whereas the communication board only contains the GM862-GPS module. The advantage of this design is the smaller surface area, while the total height is increased, resulting in a very compact box-like design.

The dimensions of the design concept are just rough estimates, they might be further reduced by optimizing the PCB layout.

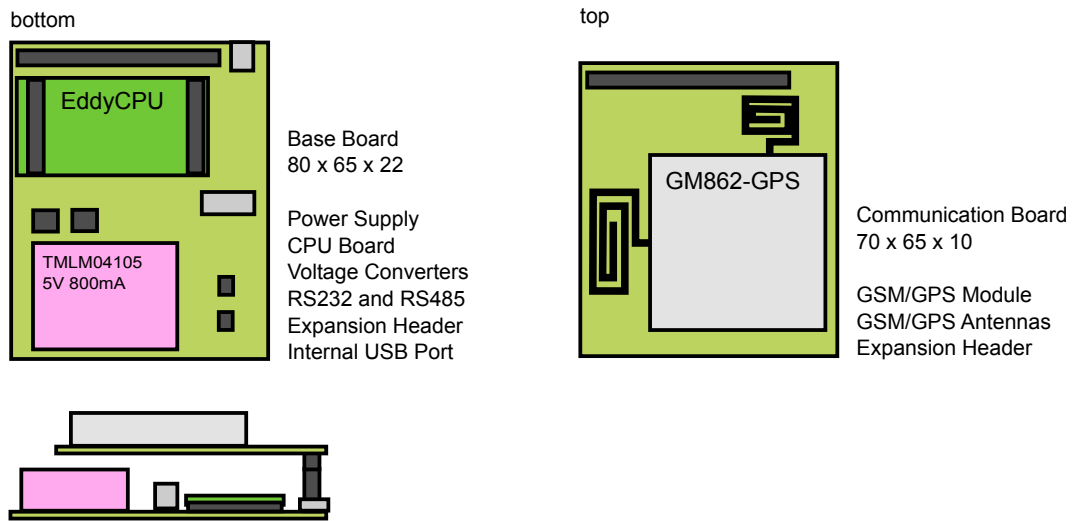


Figure 32: A two-layer concept of a miniaturized data logger

20.2 Further Miniaturization

The miniaturization concept provides a compact design, but it can still be optimized even further. Using two PCBs instead of one is a disadvantage, because production costs for two PCBs are generally higher than for a single PCB. A second cost factor are the modules: The Eddy-CPU module costs approximately 60 € and the GM862-GPS 70 €. As soon as the data logger is produced on a larger scale, it might be less expensive to buy the components of the Eddy-CPU module (Atmel AT91SAM9260 CPU, 32 MB RAM, 8 MB flash memory) separately and integrate them into the PCB design. Furthermore, the Eddy-CPU module contains additional components that are superfluous in our design, such as an Ethernet controller.

An overview of the components of a fully miniaturized data logger and their relationships are given in Figure 33. These are only the active components of the system, supporting parts like resistors and capacitors are omitted here.

Based on this concept, the PCB can be made even more compact. If a smaller GSM module like the Telit GM863-GPS can be integrated, it might be possible to create a single-PCB design which is not bigger than the original miniaturized design.

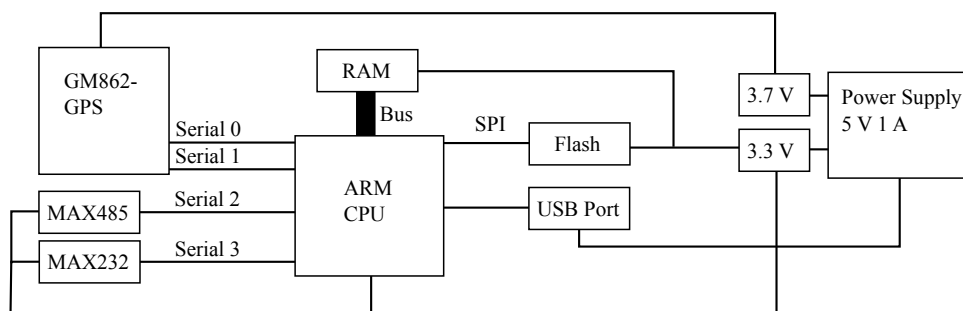


Figure 33: A simplified schematic of a fully miniaturized data logger

Part VII

Experiments and Results

During the project, several experiments and tests have been conducted in order to evaluate concepts and to test implementations. This part describes the tests conducted on the various data acquisition system components being developed, and the conclusions that can be drawn based on the results.

21 Testing Hardware

The power tool all tests are conducted with is the Bosch GST 135 BCE jigsaw, which is provided by Scintilla AG.



Figure 34: The Bosch GST 135 BCE. Image taken from Bosch website.

We use this tool for the following reasons:

- it is powered by 230 V AC line voltage
- at a peak power of 650 W, it is well suitable for measuring both high voltages and currents
- it has the most space inside for potential sensor interface integration
- the speed can be regulated, there are 6 speed levels
- an automatic speed control keeps the motor speed steady, even under load
- thanks to the speed control, speed measurement is possible without extra hardware

22 Sensor Interface Experiments

22.1 Experiments on the original Sensor Interface Design

Most of the tests conducted on the original sensor interface have been done in an early stage of development.

22.2 Experiments on the new Sensor Interface Design

After designing and implementing the new sensor interface, the newly introduced concepts are tested to find out if they bring the desired improvements over the original design.

22.2.1 RS485 Latency Tests

Using a half-duplex RS485 connection to access the sensors brings forth some concerns about the performance. Since the sensors have to be read out by sending a command first, we have a polling mechanism that introduces a delay to each command. More precisely, a sensor command must return its response or time out before the next command can be executed. Therefore, it is important to know how many commands can be executed in a given timespan, so it can be estimated how often the sensors can possibly be accessed.

The *latency* of a command is the total time it takes from sending the command request until the response is received and processed.

In this test, we measure the latency of the echo command (see Section 8.1.3). It is a command whose request and response are both 7 bytes long (1 byte of payload is used).

The test is conducted as follows: 1000 echo commands are sent from the Eddy-DK developer board to the sensor interface. No other tasks are running on the Eddy-CPU module at this time. The total time to process all commands is recorded and divided by 1000 to get the average latency of each echo command.

At a baud rate of 38400 bps, and depending whether debug output is disabled or enabled, the average latency of each command is between 5 and 10 milliseconds. This result is within the expected range, as the (theoretical) minimal latency to send and receive 7 bytes sequentially over a 38400 bps connection is 2.9 ms. Considering additional time needed for command processing, this is a good result.

As far as the echo command is concerned, between 100 and 200 commands can be processed per second. Considering that the sensor access commands, especially the responses which contain sensor data, are a bit more complex, so a maximum of 100 commands per second is a good estimate.

22.2.2 Voltage Measurement Issues

The Problem

During testing of the new sensor interface design, ADE7753 energy meter measures a wrong voltage. When accessing the sensor, the output of the RMS voltage register is approximately equal to 10 V, while the GST 135 BCE, which is the power tool used for testing, is running at the highest setting and thus should run at the full 230 V.

The old sensor interface based on the original design does not have this issue when the same sensor interface firmware is loaded, so it is not a software-related problem.

Measuring the analog signal at the input of the ADE7753 using an oscilloscope, the output looks like the signal shown in Figure 35.

Looking at this signal, the cause of the measurement issue is revealed. The voltage measurement transformer is very sensitive to high frequencies, as it is normally supposed to measure AC voltages, which have a sine waveform. In contrast, the GST 135 BCE jigsaw uses phase angle control to regulate the motor speed, which cuts off the sine wave and therefore creates transitions of high frequencies. The phase angle control signal is shown in Figure 36, in this case it is measured directly at the motor.

The voltage measurement transformer differentiates the input signal, that is why there are only short spikes in the output signal where the high frequency edges of the input signal have been.

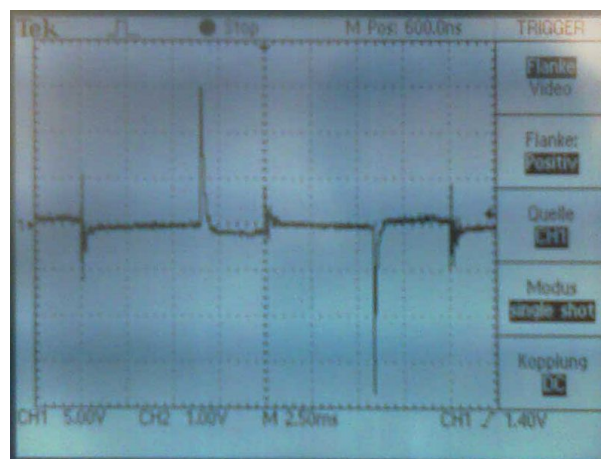


Figure 35: The crippled phase angle signal measured with a voltage measurement transformer

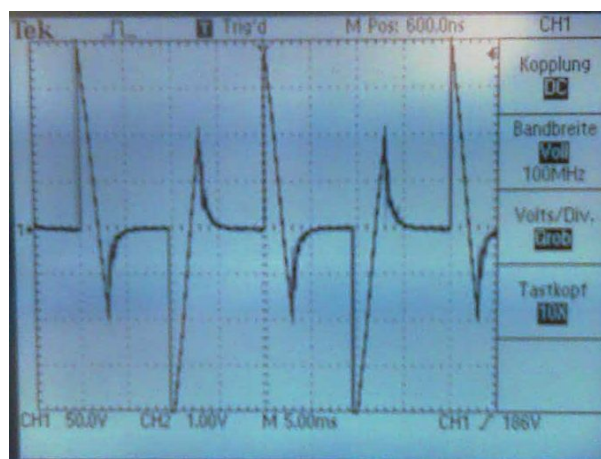


Figure 36: The phase angle signal measured directly at the motor of the Bosch GST 135 BCE

Fixing Attempts

Finding a quick fix for the problem without a re-design of the sensor interface is difficult. The signal output by the voltage measurement transformer is already missing most of the information.

The only possibility is adding a capacitor parallel to the output that might smoothen the spikes.

Different capacities between 33 nF and 100 nF are tested but have little effect on the output signal. The spikes are slightly smoother, but this does not result in the desired output signal.

Consequently, using the voltage measurement transformer in the sensor interface is a failure, it must either be replaced or omitted completely.

23 GPRS Experiments

23.1 GPRS Upstream Bandwidth Test

A performance test has been conducted to measure the GPRS upstream bandwidth. This is also a performance test for the GM862-GPS C++ library. The upstream bandwidth is more relevant than the downstream in this case, because we are mainly transferring acquired sensor data.

The test is set up as follows:

- a test server providing a TCP socket, implemented in Java, available from the internet
- the GM862-GPS equipped with a Swisscom Mobile Unlimited SIM card
- a C++ test application running on the Eddy-DK developer board
- the GM862-GPS configured with an MTU size of 64
- the GM862-GPS library uses AT command mode to transfer data, the data format is hexadecimal

The following test is performed:

1. set packet size to 32, 64, 128, 256 and 512 bytes²
2. connect GPRS
3. transmit 10 data packets in a row
4. disconnect GPRS

The test is repeated 10 times for each packet size. For each iteration, the time needed to transmit the 10 data packets is measured.

Test Results:

| Packet Size | 32 | 64 | 128 | 256 | 512 |
|----------------|---------|---------|---------|---------|---------|
| Upstream (bps) | 1734.93 | 2686.95 | 4054.43 | 5482.33 | 6752.77 |
| | 1633.38 | 2867.49 | 4136.27 | 5151.41 | 6054.15 |
| | 1689.18 | 2867.25 | 3770.76 | 5572 | 6558.19 |
| | 1565.21 | 3149.62 | 4187.27 | 5497.09 | 6686.59 |
| | 1635.2 | 2916.29 | 4119.58 | 5618.19 | 6558.22 |
| | 1808.43 | 2851.09 | 3841.54 | 5419.21 | 6889.59 |
| | 1917.37 | 2658.84 | 4366.02 | 5602.63 | 5583.72 |
| | 1808.43 | 2983.38 | 3524.37 | 4881.25 | 5948.64 |
| | 1584.59 | 2851.57 | 4119.68 | 4952.04 | 6117.34 |
| | 1808.39 | 2672.67 | 4054.48 | 5453.22 | 6589.81 |
| Average (bps) | 1718.51 | 2850.52 | 4017.44 | 5362.94 | 6373.9 |

Table 16: The GPRS upstream bandwidth test results

The results of the test are rather disappointing. Using GPRS, an upstream bandwidth of 10 to 20 kbps would be a realistic estimate. However, even at the largest packet size, the throughput is only approximately 6.5 kbps.

²A test with a packet size of 1024 was also performed, but was aborted because it takes too long to fully transfer the buffer, resulting in too many errors.

The reason for the low throughput at small packet sizes is most likely the inefficient AT command mode. The execution of each write command takes up to 100 ms. The hexadecimal data format should not be a problem, despite requiring to send twice the amount of data (each byte to transfer is represented by two hexadecimal digits), because the serial port uses a baud rate of 115200 bps. This is much faster than the expected GPRS bandwidth.

After conducting a second test with a single write command and a packet of 1024 bytes, the bandwidth is about 9.5 kbps. 1024 bytes is the maximal amount of data that can be sent in a single command. In this case, the AT command mode should not have a great effect on the result. It is therefore likely that the Swisscom GPRS service is limited to an upstream bandwidth of 9600 bps.

Considering the results, the amount of data that can be transmitted using GPRS is limited. At the current rate, only a bit more than 1 KB can be transmitted each second. However, this is enough for the Scintilla usage scenario, as we are not constantly acquiring data, and when acquiring, the amount of data recorded per second is less than 1 KB.

24 Data Logger Experiments

24.1 Using SQLite for Local Data Storage

While searching for a suitable technology to cache sensor data on a local memory card, the idea to use an SQLite database has come up. This concept is based on the idea to use a similar database structure for the local data storage as for the server database. This could make it easier to analyze data from the local storage if necessary.

SQLite is a very simple and small implementation of a relational SQL database and can therefore be used on systems with little processing performance, so it should also be usable on an embedded system.

Nevertheless, the question whether the performance is good enough when putting the database directly on a flash memory card remains.

As a test case, 10000 entries are inserted into a fresh database which resides on an SD card attached to the Eddy-DK developer board. Each entry consists of a timestamp, a floating point and an integer value, since these are the data types most likely used when acquiring sensor data.

Besides inserting single entries one after one, SQLite also supports transactions. A transaction combines multiple statements in a single command, which can also be rolled back later, in case one of the statements fail. Using transactions requires more memory but potentially speeds up the insertion.

To get a comparison, we test single insertions as well as transactions combining 10 and 20 insertions respectively. The test is repeated a few times to get a better average. No other tasks are running on the Eddy-CPU module at the time of testing. The test results are shown in Table 17.

| Transaction Size | Seconds | | | | | |
|------------------|---------|------|------|------|------|------|
| 1 | 1285 | 1331 | 1312 | 1294 | 1293 | 1319 |
| 10 | 145 | 150 | 164 | 152 | 149 | 156 |
| 20 | 88 | 85 | 83 | 91 | 92 | 97 |

Table 17: SQLite Performance Test. Time needed to write 10000 entries at different transaction sizes.

When trying single insertions, it takes approximately 1300 seconds to insert all 10000 entries. This equals to roughly 8 insertions per seconds, which is only sufficient for applications that acquire data infrequently.

Using transactions, the performance is significantly better. Combining 10 insertions in one transaction, it takes only approximately 150 seconds to insert all 10000 entries, which equals to 67 entries per seconds. This is clearly an improvement over the single insertion, as the throughput is 8 times better.

A further improvement is achievable when combining 20 insertions in one transaction. In this case, the test only takes about 90 seconds. This is equal to 111 insertions per second and about 65% better than a transaction with 10 insertions.

Despite promising test results, the concept of using SQLite for local data storage is ditched in favor of using a simple FIFO in a later development phase. This is because the use of a database requires the sensor data to be converted multiple times instead of just once.

24.2 Multi-Threading Issues

During the implementation of the data logger application's different tasks, a strange problem has occurred.

The data logger application is based on four threads: the main thread that initializes the application, the acquisition thread which retrieves sensor data, the GPS thread which reads GPS data continuously from the GM862-GPS GPS port, and the GPRS transmission thread which tries to transmit sensor data whenever it is available. Since the Eddy-CPU module has only one CPU, there is no real parallel tasking but only quasi-parallel tasking.

The problem is that whenever more than one thread is run in addition to the main thread, the data logger application sooner or later crashes with a segmentation fault. This only seems to happen when more than one thread accesses one of the serial ports using the Eddy-CPU API functions. However, each thread accesses a different serial port, so there cannot be a conflict due to concurrent access of the same serial port.

24.3 Final Test of the Complete System

After sorting out most of the software-related problems, a final test is conducted to see if the system can work as a whole.

For this test, the sensor interface is attached to the GST 135 BCE jigsaw and connected to the Eddy-DK developer board, which represents the data logger device, using RS485. The GM862-GPS evaluation board is also connected to the Eddy-DK developer board, with both serial ports connected. The GM862-GPS is enabled in advance, because there is no GPIO pin that can activate it remotely yet.

The TCP server is running on a dedicated server accessible from the internet, with a configuration according to Appendix B.

After connecting all components, the data logger application is run from the developer IDE using the remote target agent.

To test if the working cycles are detected and data is acquired, the GST 135 BCE is run at full power a few times. After that, the server database is checked remotely using a PHPPgAdmin web interface.

A part of the test result data is shown in Table 18.

The test result data shows that working cycles are detected when the measured current exceeds its turn-on threshold value, and they are ended when the current drops below the turn-off threshold value. The data recorded during the working cycles is also transferred correctly.

However, there are some hardware-related issues with some sensors that are yet to be resolved. Therefore, the orientation, voltage and RPM columns are omitted in the table as they show incorrect values. The good news is that the data is still transferred correctly to the server.

| Device | Working Cycle | Time (us) | Latitude | Longitude | Temperature | Current |
|--------|---------------|-----------|-----------|-----------|-------------|------------|
| 1337 | 1 | 815059407 | 47.480907 | 8.2108936 | NULL | 1.387278 |
| 1337 | 1 | 815611759 | 47.480907 | 8.2108936 | NULL | 2.8247941 |
| 1337 | 1 | 816115483 | 47.480907 | 8.2108936 | 28.6875 | 2.8247941 |
| 1337 | 1 | 816644572 | 47.480907 | 8.2108936 | 28.6875 | 0.95357764 |
| 1337 | 1 | 817176040 | 47.480907 | 8.2108936 | 28.6875 | 0.11172178 |
| 1337 | 2 | 868351605 | 47.481026 | 8.210535 | 28.6875 | 5.2753506 |
| 1337 | 2 | 868894475 | 47.481026 | 8.210535 | 28.6875 | 2.6606088 |
| 1337 | 2 | 869395241 | 47.481026 | 8.210535 | 28.6875 | 0.17824055 |
| 1337 | 3 | 889581532 | 47.481071 | 8.2105064 | 28.9375 | 7.4513283 |
| 1337 | 3 | 890084128 | 47.481071 | 8.2105064 | 28.9375 | 2.6769404 |
| 1337 | 3 | 890712761 | 47.481071 | 8.2105064 | 28.9375 | 1.1106955 |
| 1337 | 3 | 891228039 | 47.481071 | 8.2105064 | 28.9375 | 0.11323177 |
| 1337 | 4 | 945242133 | 47.481033 | 8.2105951 | 28.75 | 9.236578 |
| 1337 | 4 | 945704484 | 47.481033 | 8.2105951 | 28.75 | 2.8245499 |
| 1337 | 4 | 946342569 | 47.481033 | 8.2105951 | 28.75 | 0.62300467 |
| 1337 | 4 | 946974007 | 47.481033 | 8.2105951 | 28.625 | 2.6193795 |
| 1337 | 4 | 947490352 | 47.481033 | 8.2105951 | 28.625 | 0.19792144 |

Table 18: Test results from the final system test. Orientation, voltage and RPM columns have been omitted here due to false measurements caused by hardware problems.

25 Conclusion

Looking back at the development process of the data acquisition system, a lot of knowledge and insight has been gained.

At first glance, developing a data logger sounds easy, as it does nothing but record data. However, considering the effort required to create a whole data acquisition system that consists of several hardware and software components, there is much more complexity than one might think.

The development of hardware can be much more difficult than software development, as it needs a lot of planning, purchase of parts, and experimenting. Mistakes cannot be fixed as easily as it can be done with software, every mistake detected can lead to time-consuming tasks of finding a work-around so that the hardware does not have to be re-designed from scratch. When working with hardware, theory and practice sometimes diverge. A concept of an electronic circuit may look good in theory, all the calculations may be great. In reality, one cannot be sure a concept works until it is built with real hardware and tested.

Until a hardware is fully developed, several functional models and prototypes may be required until all issues are resolved.

Looking at the results, we have a good approach to a flexible data acquisition system that could be used in different situations. We have an extensible implementation of sensor hardware and software, and we have a centralized data collection system which is already tested with different scenarios.

Still, there is much to do until the data acquisition system reaches its desired functionality and reliability. Implementing new sensor takes a considerable amount of work, as every sensor works differently.

The data logger is still just a functional model, which demonstrates its functionality based on the Scintilla usage scenario. It however demonstrates that the system as a whole can function, and the outlook of reaching production-quality looks promising, when more time is invested in the development.

Part VIII

Future Work

Looking back on the whole development process, the amount of work and time required to design and implement a whole data acquisition system has been underestimated. Some aspects of the system have been thought about but could not yet be implemented due to lack of time. This part describes concepts that will be implemented in the future to improve the data acquisition system.

26 Sensor Interface Improvements and Fixes

The sensor interface design described in Section 9.5 has two fundamental flaws:

1. The transformers that isolate the sensor interface from the AC line phase potential require too much space for the PCB to be built into a power tool.
2. The voltage measurement does not work, because the voltage measurement transformer cannot handle a phase angle control signal.

An improved concept eliminates the measurement issues and allows the sensor interface to be smaller.

26.1 Elimination of Measurement Issues

The current sense and voltage measurement transformers are used to prevent the operator and the circuit from being electrocuted in the first place, in case something goes wrong. On the other hand, the sensor interface is supposed to be inside the power tool and not accessible from the outside, so the danger for the operator is minimal. However, an RS485 connector should be accessible so the sensor interface can be connected to the data logger.

This leads to the conclusion that, if the RS485 connector is isolated, the current and voltage measurements can be performed without using transformers. Instead, it can be done according to the reference design of the ADE7753 evaluation board [EVAL7]. A schematic of the concerned part is given in Figure 37

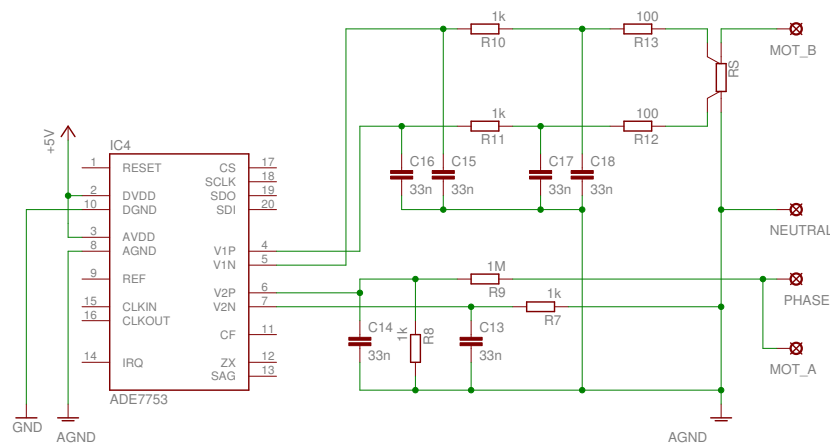


Figure 37: Non-isolated current and voltage measurement according to the ADE7753 reference design

In this design, a shunt resistor R_S is used to measure the current. At least a 3 W model is necessary to handle the power dissipation of the resistor due to the high currents of up to 10 A running through the motor wires. The voltage is measured directly from the motor contacts, a voltage divider is used to convert the AC line voltage to a level that can be measured by the ADE7753.

As a consequence, the analog ground plate $AGND$, which is also connected in to the digital ground GND in one spot, could be at neutral or phase potential, since the AC plug can be plugged in both ways.

The isolation is done at the RS485 interface in the improved design. Figure 38 shows what the schematic of an isolated RS485 interface might look like. The data lines that connect the MAX485 RS485 transceiver to the microcontroller are isolated by optocouplers. Since these optocouplers need a power supply on either side, an optically isolated DC-DC converter provides an isolated power supply to the transceiver and the optocoupler contacts on the transceiver side. The DC-DC converter gets its input voltage from the sensor interface power supply, which also powers the optocoupler contacts on the microcontroller side.

This design ensures that, even if the sensor interface's ground plate is at phase potential, the RS485 transceiver is not, because the DC-DC converter and the optocouplers provide a floating potential on the transceiver side.

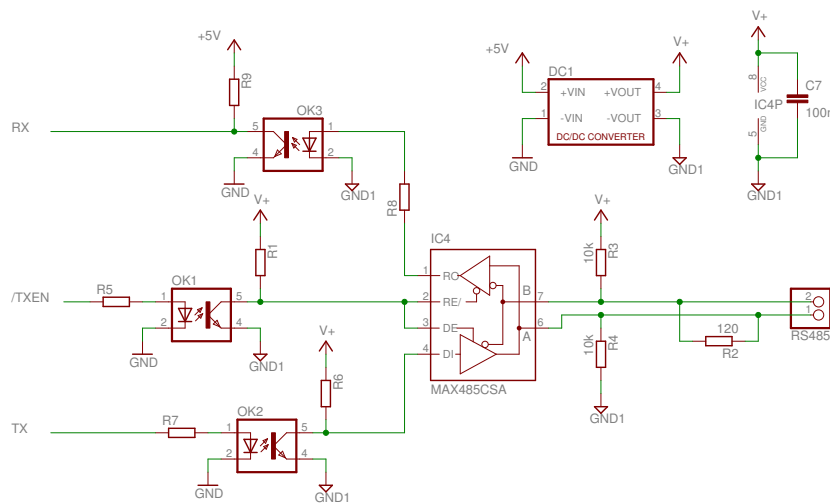


Figure 38: Schematic of an isolated RS485 interface

26.2 Built-In Sensor Interface

The sensor interface design according to Section 9.5 describes a sensor interface consisting of a single PCB that contains all components including sensors. The PCB measures 83 mm x 63 mm x 24 mm, which is still big when it is supposed to be built into the back of the GST 135 BCE. A rather large extension has to be made to the back cover of the GST 135 BCE for the sensor interface to fit in. A mock-up of this extension is shown in Figure 39, the GST 135 BCE will still be usable, but the suction pipe that connects to a vacuum cleaner cannot be used anymore.

With the improved sensor interface, the size of the PCB can be drastically reduced, because the two measurement transformers are no longer necessary. With a few optimizations of the PCB layout, an estimate is a 50% reduction in size and height.



Figure 39: The Bosch GST 135 BCE modified for the built-in sensor interface

27 Implementation of Miniaturization

Currently, the data logger is just a functional model implemented on a developer board. The miniaturization described in 20 is only a concept, it has not been implemented yet.

27.1 Implementation of the Miniaturization Concepts

The implementation of the two-layer concept described in 20 would be the first step to a production-ready data logger device. It still uses the Eddy-CPU module as a core system, which makes the PCB easier to develop, as the CPU module bears the smallest components and most complex wiring.

For maximal miniaturization, the second concept must be implemented, which eliminates the Eddy-CPU module and features its own ARM CPU instead. The PCB gets more complex that way, but when produced in higher quantities, the production costs are reduced, because the components of the Eddy-CPU module alone would be cheaper than buying a complete module for each device.

27.2 Battery-Powered Data Logger

The current data logger hardware concept features an integrated power supply, so that the data logger can be run off a 230 V line voltage. This makes sense, because in the Scintilla usage scenario, power tools are used which are corded 230 V models.

However, since this data logger device is supposed to be mobile, there should also be a battery-powered model.

Using a battery instead of a power supply would not require too much modification, except for different voltage regulators. Also, the battery must be chosen carefully, as the data logger needs 3.3 V for logic and 5 V for the USB port. The GM862-GPS module is not a problem, since it is already designed to run off a 3.7 V LiPo cell.

As a consequence, in order to get a 5 V supply for the USB port, either a step-up converter must be used, or a LiPo cell with a higher voltage, for example 7.4 V.

28 Data Evaluation Software

In 2008, a sensor data evaluation software for the Scintilla usage scenario has been developed at the Institute of Mobile and Distributed Systems.

This evaluation software is a Java-based desktop application which accesses an SQL database containing device information and recorded data. Working cycles of each device can be analyzed graphically for all sensors.

The problem with this desktop application is that it is based on earlier specifications, which have changed during the development process of the data logger. It is therefore not compatible with the database layout of the data collection server.

In the future, the desktop application will be modified to match the current database layout.

29 Remote Configuration of the Data Logger

In the current implementation, the data logger configuration is hard-coded for testing purposes, it cannot be changed remotely. For such an embedded mobile device, it is however important that the configuration can be changed. Due to the lack of an input device and display, the data logger must be configured remotely.

Two scenarios are possible: configuring the data logger by connecting it to a computer, using a communication interface, for example RS232, and remote configuration over the internet.

29.1 Configuration through a Computer

It is important that the data logger can be connected to a computer for configuration purposes. When the data logger is used for the first time, some configuration parameters have to be set up, or it will not work.

Most important parameters are the SIM card PIN code, the device ID and password and the address of the data collection server. Without PC connectivity, these parameters must be hard-coded and the PIN code must be set up accordingly before the SIM is inserted into the data logger. Remote configuration via SMS or GPRS is not possible at this time, because the SIM card has to be operational in order to make SMS and GPRS work.

29.2 Configuration over the Internet

If the data logger can be configured remotely over the internet, people can make modifications or check its status without the need of physical interaction.

There are two concepts that could be used to realize remote access: using SMS or GPRS.

Using SMS is relatively simple. The data logger periodically checks for incoming SMS, and if there is an SMS containing valid command data the data logger can handle, the configuration is updated or the command executed. If necessary, the data logger can send back a status response SMS to the sender.

The advantage of this method is that a GSM device can always be reached by SMS from anywhere a GSM network is available. Since anyone could send SMS to the data logger in this case, the incoming SMS must be validated so nobody can access the data logger without permission.

Using GPRS is a bit more complicated. GPRS connections are usually protected from the outside by using NAT. This means that the GPRS client gets a private IP address, which is not accessible from the outside. For example, this is also true for the Swisscom GPRS service used for testing. To circumvent this problem, an intermediate server could be used as a relay. The command requests are placed on this server, and the data logger acting as a GPRS client periodically connects to this server to check if there are commands to be executed.

29.3 Embedded Software Modification

To allow remote configuration, an additional thread has to be created within the data logger software.

Currently, there are threads for data acquisition, GPS and transmission of acquired data. The new remote configuration thread periodically polls the RS232 port to check if a PC is connected. The PC commands have to be interpreted and executed. During configuration update, the data acquisition and transmission must be interrupted.

To allow remote configuration via SMS, the GM862-GPS module has to be polled periodically for incoming SMS, which are then interpreted. When SMS are accessed, data transmission must be interrupted, as the GM862-GPS can only execute one command at a time.

30 Persistent Local Data Storage

In the current implementation, the local data storage is merely a memory-based FIFO, which does not persist the data in case the data logger device is turned off. As a consequence, sensor data which is acquired but not yet transferred to the server is lost.

This issue can be eliminated when a persistent FIFO is used. A persistent FIFO is backed by a file residing on a USB flash drive attached to the data logger. The FIFO is always synchronized to its file, whenever data is put into the FIFO or removed, thus allowing the data logger device to be turned off without losing data.

In addition, the data put into the FIFO can be kept instead of being removed when read. In this case, the FIFO file can grow up to a pre-defined size or until the USB flash drive is full. The position to read from must be kept track of, so data that was already transferred is not read again. This concept is visualized in Figure 40.

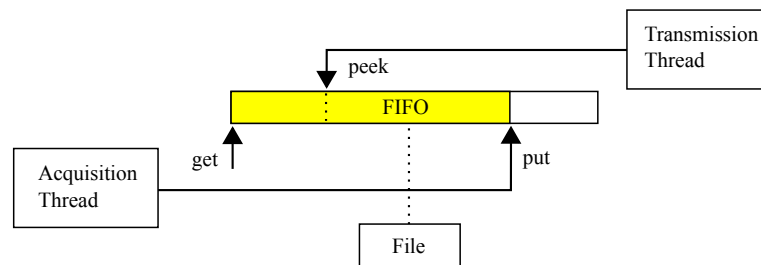


Figure 40: A file-based persistent FIFO. Data can be inserted (*put*), removed (*get*) and read without removal (*peek*).

Part IX

Appendix

A Versatile Sensor Protocol Identifiers

A.1 Valid Device Addresses

| | |
|----------|---------------------------------------|
| 00h | reserved for master device |
| 01h..FEh | free choice of slave device addresses |
| FFh | reserved for broadcast |

A.2 Valid Command IDs

| | |
|-----|----------------------------|
| 00h | Echo |
| 01h | Read sensor value |
| 02h | Write sensor configuration |
| FFh | Command error |

A.3 Error Codes

| | |
|-----|--|
| 00h | No Error |
| 01h | Invalid frame size |
| 02h | Checksum error |
| 10h | Unknown command |
| 11h | Unknown sensor type |
| 12h | Invalid data size |
| 20h | Command timed out |
| 21h | Sending command request failed |
| 22h | Device is busy (command being processed) |
| 23h | Invalid source address |
| FEh | Error in receiver state machine |
| FFh | No Reply |

A.4 Sensor IDs

These sensors are supported in the current implementation:

| | |
|-----|-------------------------------------|
| 01h | RPM counter |
| 02h | Microchip TC77 temperature sensor |
| 03h | Analog Devices ADE7753 energy meter |
| 04h | Freescale MMA7455L accelerometer |

B Example XML Protocol Specification

This is an example XML specification used in the proof-of-concept implementation.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<protocol>
  <title>Protocol for Scintilla Data Logger V1.0</title>
  <protocol_properties>
    <id>0x02</id>
    <revision>0x00</revision>
    <description>Scintilla Tool Data Acquisition</description>
  </protocol_properties>
  <data_channels>
    <channel>
      <id>1</id>
      <name>WorkingCycle</name>
      <preserve>true</preserve>
      <subchannel>
        <name>ID</name>
        <bytes>8</bytes>
        <format>int 64</format>
      </subchannel>
    </channel>
    <channel>
      <id>2</id>
      <name>Timestamp</name>
      <preserve>true</preserve>
      <subchannel>
        <name>Microseconds</name>
        <bytes>8</bytes>
        <format>int 64</format>
      </subchannel>
    </channel>
    <channel>
      <id>3</id>
      <name>Position</name>
      <preserve>true</preserve>
      <subchannel>
        <name>Latitude</name>
        <bytes>4</bytes>
        <format>float</format>
      </subchannel>
      <subchannel>
        <name>Longitude</name>
        <bytes>4</bytes>
        <format>float</format>
      </subchannel>
    </channel>
    <channel>
      <id>4</id>
      <name>Temperature</name>
      <preserve>true</preserve>
```

```

    <subchannel>
      <name>T1</name>
      <bytes>4</bytes>
      <format>float</format>
    </subchannel>
  </channel>
<channel>
  <id>5</id>
  <name>Orientation</name>
  <preserve>>true</preserve>
  <subchannel>
    <name>Pitch</name>
    <bytes>4</bytes>
    <format>float</format>
  </subchannel>
  <subchannel>
    <name>Roll</name>
    <bytes>4</bytes>
    <format>float</format>
  </subchannel>
</channel>
<channel>
  <id>6</id>
  <name>Electricity</name>
  <preserve>>true</preserve>
  <subchannel>
    <name>Current</name>
    <bytes>4</bytes>
    <format>float</format>
  </subchannel>
  <subchannel>
    <name>Voltage</name>
    <bytes>4</bytes>
    <format>float</format>
  </subchannel>
  <subchannel>
    <name>Energy</name>
    <bytes>4</bytes>
    <format>float</format>
  </subchannel>
</channel>
<channel>
  <id>7</id>
  <name>Speed</name>
  <preserve>>true</preserve>
  <subchannel>
    <name>RPM</name>
    <bytes>2</bytes>
    <format>u_int16</format>
  </subchannel>
</channel>
</data_channels>

```

</protocol>

C Glossary

| | |
|-----------------|--|
| ADC | Analog-digital converter |
| BGA | Ball Grid Array |
| CSD | circuit switched data, cellular data connection |
| Data logger | a device that receives data from input sources and stores the acquired data |
| GGSN | Gateway GPRS Support Node |
| GPIO | General Purpose I/O, usually stands for an output or input pin for custom use |
| GPRS | General Packet Radio Service, cellular networking technology |
| GSM | Global System for Mobile Communications, cellular communication technology |
| LGA | Land Grid Array |
| MCU | Microcontroller Unit |
| Microcontroller | an embedded processor which integrates CPU, RAM, flash and I/O in one chip |
| NMEA | National Marine Electronics Association, NMEA 0183 is a standard for GPS data |
| PCB | printed circuit board |
| PPP | Point-to-point protocol |
| RS232 | an asynchronous serial interface |
| RS485 | a serial bus interface with differential signaling |
| SGSN | Serving GPRS Support Node |
| SMD | surface-mounted device, an electronic part that is soldered directly on top of a PCB |
| SMS | short message service, cellular text messaging |
| SPI | serial peripheral interface |
| UART | Universal Asynchronous Receiver Transmitter (normally used in RS232 interfaces) |

References

- [CGS] Smart Sensors and Network Sensor Systems - Two Approaches to Providing Smart Sensor Capabilities: Incorporating Identification Capabilities per IEEE 1451.4 (Smart Isotron), or Complete Data Acquisition and Networking Capabilities (Network Sensors). Anthony Chu, Fernando Gen-Kuong, Bruce Swanson, ENDEVCO Corporation
- [EVAL7] Evaluation Board Documentation - ADE7753 Energy Metering IC, Analog Devices
- [FS06] Feasibility study on application of GSM-SMS technology to field data acquisition. Chwan-Lu Tseng, Joe-Air Jiang, Ren-Guey Lee, Fu-Ming Lu, Cheng-Shiou Ouyang, Yih-Shaing Chen, Chih-Hsiang Chang, National Taipei University of Technology Taiwan, National Taiwan University, 2006
- [L485] A Study on Intelligent Greenhouse Temperature Measurement System Based on RS485BUS. LI Dongming, LIU Yongfu, Mechanical and Electrical College of Hebei Agriculture University / Information Scientific and Technologic College of Hebei Agriculture University, Baoding, 071001
- [LE] A "SMART SENSOR" BUS FOR DATA ACQUISITION. Lee H. Eccles, Boeing Commercial Airplane Company
- [M485] MAX485 Datasheet. Maxim Corporation, 2003. http://www.datasheetcatalog.com/datasheets_pdf/M/A/X/4/MAX485.shtml
- [MB02] MODBUS over Serial Line - Specification and Implementation guide V1.0. Modbus.org, 2002
- [MH07] RS-485 Serial Data Communication – Physical Layer for Factory Automation and Process Control Networks. Mark E. Hazen, High Performance Analog, Intersil Corporation
- [MI03] Using Gravity to Estimate Accelerometer Orientation. David Mizell, Cray Inc., 2003
- [PB98] Normative Parts of PROFIBUS -FMS, -DP, -PA according to the European Standard EN 50 170 Volume 2. 1998
- [TU07] Tilt Sensing Using Linear Accelerometers. Kimberly Tuck, Accelerometer Systems and Applications Engineering, Tempe, AZ, 2007 (Freescale Application Note 3461)
- [WL09] The Design of Remote Medical Monitoring System Based on Sensors and GPRS. Wang Xiaohong, Information and Engineering Department, Shandong Jiaotong University, Jinan, 250023, China, Liu Li, Waterborne Transportation Institute the Ministry of Communications, Beijing 100088, China
- [WX09] Application and Research of Data Acquisition Technology Based on GPRS. Wenbin Fan, XiangLi, Peng Chen, School of Computer, China University of Geosciences Wuhan, 2009
- [YAL10] Yaler – a simple, open and scalable relay infrastructure for the Web of Things. Marc Frei, Thomas Amberg, Oberon microsystems AG, Zurich, Switzerland