Studienarbeit

Manuel Stübler

Technische Informatik

Fakultät Informationstechnik

Hochschule Esslingen

Wintersemester 2010/2011

Betreuer	**Prof. Dr. Jörg Friedrich**
Aufgabe	**Entwicklung einer Analyse-Software für das Remote-Monitoring eines Quadrocopter.**
Stand	**18.01.2011**

Table of Contents

List of Figures

List of Tables

1 Project

This chapter describes and introduces the internal structure of the “Valiquad” project. It defines the standards and roles that will be used within the project. Beginning with a brief description of the project and the task it fulfills, this chapter will advance with explaining the infra-structure and architecture of the project management. It also lists the artifacts that will result from the development process and all responsibilities between the participating instances. Those modular artifacts will form the final system and are put together in the end.

To guarantee a high quality of the system, the project management will keep to approved methods and tools. This process will be explained in detail within the following sections.

1.1 Description

Valiquad is an analyzing tool for tracking and remote monitoring the Quadrocopter with all of its stored and measured sensor data. The Quadrocopter is a helicopter with four, oppositely located propellers. It was originally developed by the master's course at the “University of Applied Sciences Esslingen”. The analyzing software, namely Valiquad, is written in Java and uses radio-frequency to communicate with the helicopter. It can observe the measured data from the sensors that are mounted on the system. This analyzing tool is also able to set and retrieve system parameters, for example from the controllers and other configurable parts within the aircraft. Those two essential tasks of the software enable the analyzing of the flight behavior with different controller and system settings and is therefore kind of a debugging tool on the systems level.

Valiquad will primarily provide the possibility to read out measured values over time and for example show them within a diagram of an according graphical interface to the user. Therefore it is a tool for debugging and analyzing the helicopter and its system parameters. This is necessary as it is not possible to use a stepwise debugger while the system is in the air, as this means that the micro-controller would be interrupted and therefore prevented from controlling the propellers, which would cause the helicopter to crash. This fact makes it necessary to find other possibilities of observing the system and analyzing its behavior. Exactly at this point the responsibility is given to the Valiquad project.

For the transmission of the data from the Quadrocopter to the analyzing software and vice versa, a special application layer protocol will be introduced, which matches the requirements of the system. Those system requirements will be explained and stated in the next chapter and form the basis of the whole software.

The protocol is the heart of the system and will base upon a radio-frequency technology called ZigBee, which is one of the constraints to the system. As the lower layers of this communication system are already given, the protocol uses them and adds its functionality at the higher levels as explained in detail in the following chapters.

The software itself will result in a library that can be used in further application specific software, which for example could add a graphical user interface (GUI). It is also possible to use this library as a middle-ware technology connecting other programs directly to the analyzing and debugging process of the helicopter. Such a program might be Matlab or any other kind of an analyzing or automation software that shall control or observe the helicopter and its states remotely. There are many possible applications this library can be used for and therefore a quite generic approach will be used to enable as many new services as possible.

1.2 Process Overview

This section gives a brief description of the artifacts and products generated within the process of the project management, architectural design and requirements engineering. Those artifacts will also be part of the final documentation and are the basis for any further steps. The technical product will be generated in another step and is part of chapter 5 Implementation.

Activity	Product
PM	Project manual and project plan
RE	Requeriments document for the system
SA	Documentation of the software architecture and implementation
SE	Protocol design and documentation
SE	API documentation of the java-related software
SE	Source-code documentation for the helicopter software
IT	Testing and integration documentation

Tabelle 3-1: Process Overview

1.3 Organization

This section describes the structure and organization used for the project. Main goal of the organization is the separating of the system into several tasks that can be done sequentially or in parallel. A task is linked to a corresponding role. Those roles are explained in the next chapter.

The project plan in chapter 1.5 shows which tasks can be done in parallel and which have to be done sequentially because of dependencies to earlier tasks.

1.3.1 Tasks and Responsibilities

Project Manager (PM)

- Developing the project manual
- Tracing the development process
- Organizing team meetings
- Planning code reviews

Requirements Engineer (RE)

- Developing requirements and the according documentation
- Informing the customer about the project status
- Developing the solution concept and abstracting the development process

Software Architect (SA)

- Developing the overall software architecture and keeping track of the whole development process
- Creating UML-diagrams to illustrate the interaction between different software components
- Defining the API (application programming interface) between the upper and lower layer of the software

Software Engineer Hardware (SE)

- Developing and integrating the hardware specific software
- Developing the radio-frequency communication protocol and data synchronization between the application and the Quadrocopter
- Developing the model according to the MVC-pattern
- Implementing the communication protocol in Java for the application on the host and in C for the micro-controller on the remote system (the Quadrocopter)

Integration and Test Engineer (IT)

- Integrating and testing the final system
- Defining suitable test cases for the system
- Tracking the whole testing process of the system

1.4 Standards and Regulations

This section handles the defined standards and regulations for the overall development process. Such standards are the tools that have to be used during the process of development and style guidelines for the source-code itself

1.4.1 Tools

The following table defines the tools that were used within the project. Those tools shall also be used if the system is developed any further to reduce the complexity of introducing a completely new project infra-structure.

Activity	**Tool**
Documentation	Open Office 3
Project Plan	GanttProject
Target IDE	CodeWarrior
Host IDE	Eclipse (+ several plug-ins)
Communication	X-CTU (analyzing the traffic and configuring the XBee module)

Tabelle 5-2: Activity and Tools

1.4.2 Style Guideline

The source-code that is written in Java must adhere to the Java conventions. This can be accomplished by activating the Eclipse formatter using the pre-defined and built-in “Java Conventions” with a tab-size of 4. The Java source-code has to be documented by using JavaDoc comments. This is used to automatically generate an API documentation out of the source files.

The code that is written in C for the micro-controller on the Quadrocopter has to keep to the established style that is already used within the existing source-code.

1.5 Project Plan

The following figure shows the project plan as a gantt diagram.

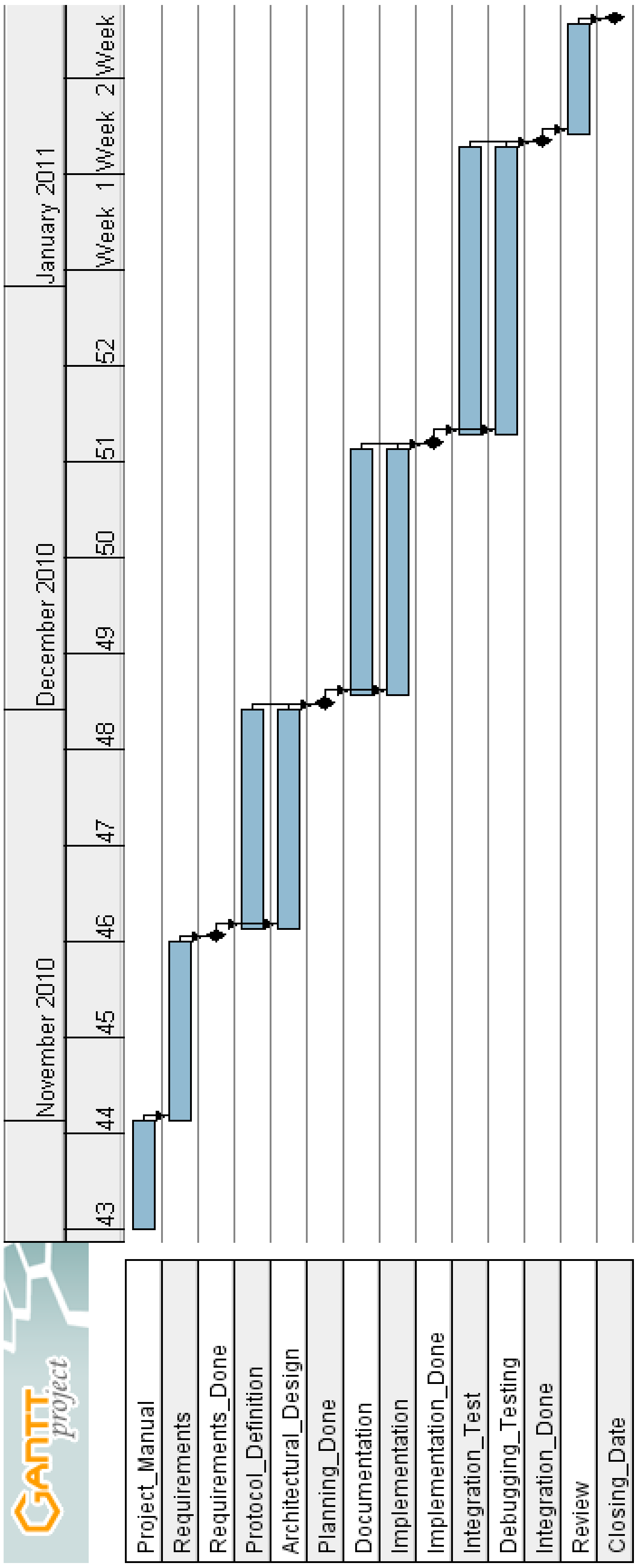

Figure 1: Project Plan

2 Requirements

This chapter describes and lists the requirements for the lower layers of the Valiquad project. Those layers include the communication protocol that is used between the analyzing software (Valiquad) and the helicopter (Quadrocopter), the data synchronization and storage within the data model of the analyzing software and also the software that runs on the micro-controller of the helicopter to control the sensors and the parameters for the controllers. The requirements were defined in advance before the protocol and the concrete implementation were started. This is an established way in software engineering to ensure an adequate quality and is also the fist step within the V-Model as described in the following section.

2.1 V-Model

The V-Model is one of the major concepts to handle the raising complexity of software projects. It describes the process of developing highly complex software step by step in a circular manner. The development starts with the requirements analysis to define the system constraints and functions. The next step is the architectural design of the complete system. If the design of the system is finished, the modules that form the system are described and defined. The deepest step is the implementation of the modules as defined in the previous steps. This implementation can be split over several teams and may itself form a V-Model recursively. If all relevant modules are implemented, they can be tested and integrated. Putting all the tested modules together will form the complete system which itself has to be tested in the last step. In this step the interfaces of the modules are the weak point of the system. If they were not exactly defined, the whole system won't work.

If an error occurred on the right side of the V-Model, one has to go back horizontally to the according design step. This is an iterative process and will go on and on until the system is integrated and tested successfully. This shows that every design step has its integration and testing step and it also shows that errors in the early steps of the requirements analysis and the systems design will cause a higher deferral of the system than those that occurred deeper in the model, because more steps actually have to be redone. According to this fact it is necessary to make sure that the analysis and requirements definition is completely correct and putting more time in those first steps will result in a higher quality of the whole system.

The V-Model is illustrated in the next figure.

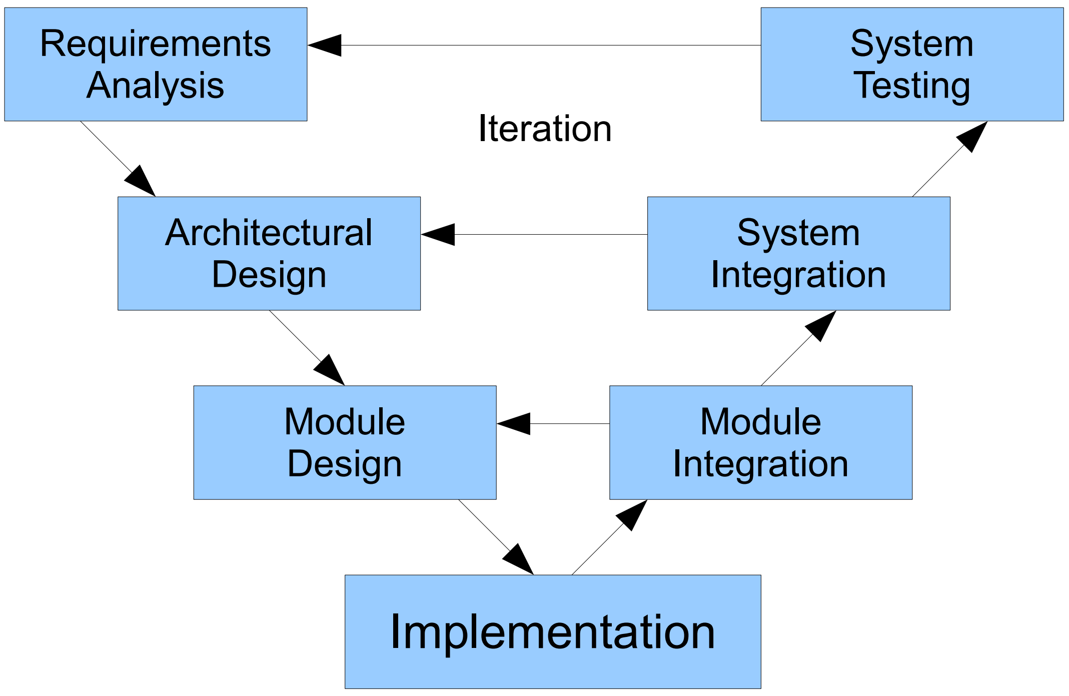

Figure 2: V-Model

2.2 Brief Description

The requirements are described and stated within a unified table, an example and reference table is shown below.

ID	Unique identification for the requirement
Title	Short title
Author (Date)	Name of the author (date of creation)
Description	The description of the requirement in detail
Test	A short explanation on how this requirement can be tested
Solution	An abstract idea on how the requirement can be implemented
Links	References to other requirements

Table 3: Reference Requirement

2.3 Requirements

ID	R1
Title	Communication Protocol
Author (Date)	Manuel Stübler (23.10.2010)
Description	The analyzing software shall communicate with the Quadrocopter using a standard radio-frequency technology. This technology is given by the currently mounted communication interface on the helicopter, which is an XBee module that uses the ZigBee communication standard. Another XBee module is attached to the host through a serial RS-232 communication port.
Test	While sending a packet from one instance to another and returning the same packet to the sender, the system shall check the sent and received packet for equality.
Solution	ZigBee allows to send a stream of bytes from one instance to another via a standardized radio-frequency protocol. By creating an appropriate application layer protocol that uses a small header and a following body, this stream of bytes is put together within a packet. Such a packet is transmitted over the communication interface and interpreted by the receiver at once. For the communication with the serial port, Java provides the 'Java Communications API' (javax.comm), which will be used by the analyzing software to communicate with the attached XBee module.
Links	None

Table 4: R1 Communication Protocol

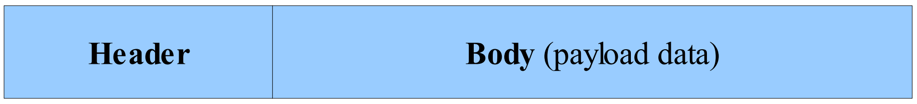

Figure 3: Structure of a physical packet (application layer)

ID	R1.1
Title	Data Integrity
Author (Date)	Manuel Stübler (23.10.2010)
Description	As the packets that are sent over the air can be accidentally modified by disturbance such as interference and other physical treatments, those packets have to be secured by a check sum. This check sum must guarantee a hamming-distance of at least 2, better 4.
Test	Send a packet with a wrong check sum and check if the receiver will detect the error.
Solution	A hamming-distance of 2 can be achieved by a simple parity check. For a hamming-distance of 4, calculating a CRC value might be a good choice. If there is a possibility to calculate CRC values for given packets with some kind of hardware-acceleration, this method shall be used instead of calculating the check-sum in a software-based fashion.
Links	R1

Table 5: R1.1 Data Integrity

ID	R1.2
Title	Data Synchronization
Author (Date)	Manuel Stübler (23.10.2010)
Description	By using the communication protocol described in R1, the analyzing software shall synchronize its database (data model) to the measured data on the helicopter. This synchronization shall implement a mechanism that allows soft real-time constraints. The data that is transmitted has to arrive at the analyzing software within a given time slot to ensure the usefulness of the sensor data. If the data arrives at a later time it still has some, but only little usage and is therefore also stored in the database.
Test	Sending loads of packets and calculating the average latency of the arriving packets. This latency has to be under a defined threshold.
Solution	The sensor values shall be send periodically one after another. The transmission rate must adhere to the constraints of the communication medium and the highest possible sample rate for the sensors.
Links	R1

Table 6: R1.2 Data Synchronization

ID	R1.3
Title	Packet Loss
Author (Date)	Manuel Stübler (23.10.2010)
Description	To fulfill the requirement R1.2 a mechanism for the handling of erroneous packets must be introduces. If the check sum reveals that an error occurred while transmitting a packet with sensor data from the helicopter to the analyzing software, it has to be discarded. This is necessary to not slow down the whole process of data transmission. Especially there will be no resending of an invalid packet, as this would cause the complete system to stumble. There will also be no acknowledge if a packet was successfully received by the host. Packets that are send the other way round by the analyzing software to the helicopter will be acknowledged by the receiver. If the acknowledge is not received by the sender within a defined time-slot, the analyzing software will resend the message. If the message was not acknowledged after the second retry, this means after sending it for the third time altogether, the analyzing software will assume the helicopter to be currently unavailable.
Test	Send erroneous packets from the helicopter to the analyzing software and check whether those packets are discarded without initiating any further steps. Shut down the helicopter and try to send a message from the analyzing software to the helicopter, check if the software will assume the helicopter to be unavailable after sending a packet for the third time.
Solution	Delete erroneous packets that are received from the helicopter within the analyzing software and do not initiate any kind of a resend. Define a special acknowledge packet that will be sent from the helicopter to the analyzing software if a packet was received successfully. If the acknowledgment is invalid, the host discards this packet and acts as if he didn't receive a packet at all.
Links	R1.2

Table 7: R1.3 Packet Loss

ID	R1.4
Title	Data Throughput
Author (Date)	Manuel Stübler (23.10.2010)
Description	The communication protocol must guarantee, that the measured data of all available sensors can be transmitted in (soft) real-time with a sample rate of at least one value per 100ms per sensor. The sensor values shall be put together in a so-called bundle, this bundle holds the values of all sensors for the same time-stamp. If necessary, several successive bundles can be sent within one packet, to ensure an adequate throughput. However the maximum latency, not regarding the time for transmitting a packet, must not exceed 1 second. This means the value of a sensor must be send within 1 second after it has been captured.
Test	Analyze the traffic of the system and check if the given constraints can be fulfilled. The oldest measured value of a sensor within a packet must not be older than 1 second when the packet is sent by the helicopter.
Solution	Regarding the highest possible sample rate of one value per 10ms per sensor, a maximum number of 100 successive bundles can be put together in one packet. A good choice might be 10 bundles per packet, which means packets will be sent every 100ms and hold up to 10 values per sensor.
Links	R1.2

Table 8: R1.4 Data Throuhput

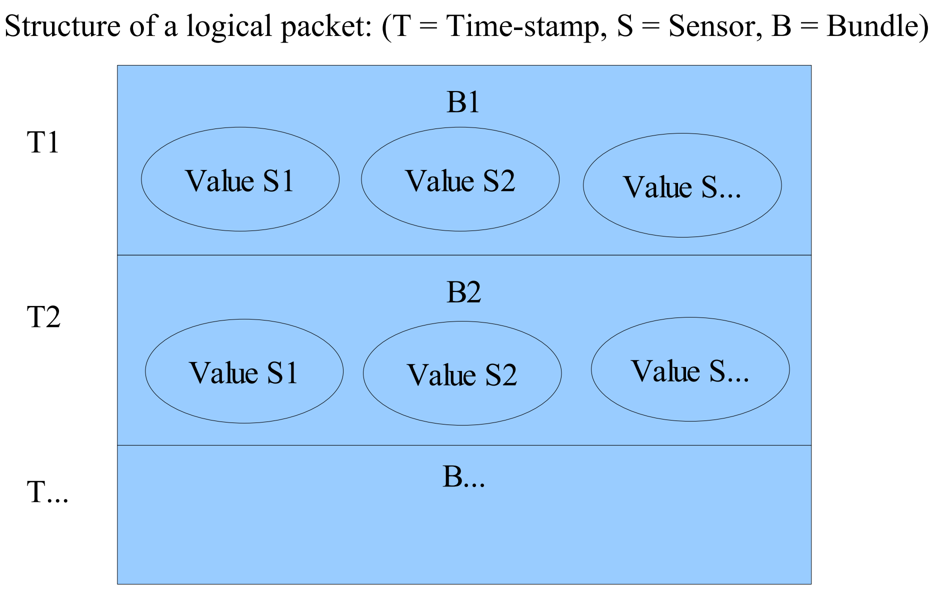
Structure of a logical packet: (T = Time-stamp, S = Sensor, B = Bundle)
T1
B1
Value S1
Value S2
Value S...
T2
B2
Value S1
Value S2
Value S...
T...
B...

Figure 4: Structure of a logical packet

ID	R1.5
Title	Flight Control
Author (Date)	Manuel Stübler (05.11.2010)
Description	The calculation and transmission of packets must not block the flight control longer than allowed. It has to be guaranteed that the time used for the protocol handling does not exceed the time-slice that is given to the task.
Test	Calculate the time used for the protocol handling and check if it is smaller than the time-slice that is assigned to the task.
Solution	Keep the calculation time as short as possible, e.g. by calculating the check-sum for a packet in hardware, if possible.
Links	R1

Table 9: R1.5 Flight Control

ID	R1.6
Title	Dynamic approach
Author (Date)	Manuel Stübler (04.12.2010)
Description	The protocol shall be made as dynamic as possible. The parameters that can be set and observed will change over time, but the protocol and the application working with shall be able to handle any future parameters that are added to the system.
Test	Add yet unknown parameters to the system and check if the protocol can handle them without knowing any further things about them.
Solution	To allow this dynamic mechanism to work properly, for example so-called parameter IDs can be introduced. Those IDs must describe such a parameter uniquely and hold all necessary information about it, for example the data type and if it is a read-only parameter or not.
Links	R1

Table 10: R1.5 Dynamic apporach

ID	R2.1
Title	Buffering Target
Author (Date)	Manuel Stübler (29.10.2010)
Description	To temporarily store the sensor values, before they are going to be transmitted, a buffer on the target (the helicopter) might be necessary. This buffer must be big enough to store all collected values from the sensors, before they can be transmitted. The lowest possible rate of data transmission has to be calculated from the available buffer size within the memory on the helicopter. The buffer will always keep new values and discard old ones if the end was reached. This form is also known as a FIFO-buffer, meaning first in first out.
Test	Set the highest sample rate with the lowest transmission rate and check, whether none of the sensor values will be lost within the depth of bits and bytes.
Solution	Check for the available memory that can be used for such a buffer and calculate backwards the possible sample and transmission rates.
Links	R2.2

Table 11: R2.1 Buffering Target

ID	R2.2
Title	Buffering Host
Author (Date)	Manuel Stübler (01.11.2010)
Description	The data received from the helicopter has to be stored in a database within the analyzing software (the host). Therefore an appropriate data structure is required, to efficiently store all sensor values for a given time-stamp. This database has to be limited to not cause a memory leak. This limit must ensure, that enough values can be stored in the database to correctly analyze the flight dynamics.
Test	Check if the defined data structure stores the values efficiently without having redundancy. Also check for a useful interface to retrieve the data from the data base that matches the requirements of the view and controller objects.
Solution	A possible solution could be a hash-map that uses time-stamps as key values and stores all sensor values with the same time-stamps within one entry in this hash-map. As an indicator for the limit of the database, the time-stamp can be used. For example values that are older than 60 seconds will be discarded.
Links	R2.1, R5

Table 12: R2.2 Buffering Host

ID	R3
Title	State Machine
Author (Date)	Manuel Stübler (29.10.2010)
Description	The software for the helicopter shall be modeled and implemented throughout a state machine, which will be written entirely in C. There is already an existing state machine, which therefore has to be adapted for the new purposes of the protocol and the system.
Test	Run the software and check for the correct states within pre-defined scenarios.
Solution	The state machine will be implemented in C using the given structure of the current software status.
Links	None

Table 13: R3 State Machine

ID	R4
Title	Dependencies
Author (Date)	Manuel Stübler (12.01.2011)
Description	The software shall be written from scratch using less dependencies as possible. This requires a higher overhead but ensures having the complete control over all parts of the system. It also ensures having no copyright or licensing problems. The software developed and delivered within this project will be completely released under the (L)GPL and is therefore free and open source software.
Test	Ensure that there are less dependencies as possible. If too many dependencies were found, this requirement was not fulfilled.
Solution	Write the software from scratch without using external libraries.
Links	None

Table 14: R4 Dependencies

ID	R5
Title	Model-View-Controller
Author (Date)	Manuel Stübler (29.10.2010)
Description	The software for the analyzing tool itself will be written in Java. The central architecture of the system is the Model-View-Controller (MVC). This pattern is also used to define the interface between the hardware-related part of the tool, that is concerned with the data transmission (data model) and the software-related part, which ought to create an adequate user interface (view and controller).
Test	Putting both parts together and checking for the correct behavior of the system.
Solution	As Java is an object oriented programming language, it makes it easy to encapsulate functionality and create usable interfaces to abstract the principles behind an implementation. This mechanism will be used to define the model and therefore the hardware-related part, and on the other hand the view and controller will be defined for the software-related part.
Links	None

Table 15: R5 Model-View-Controller

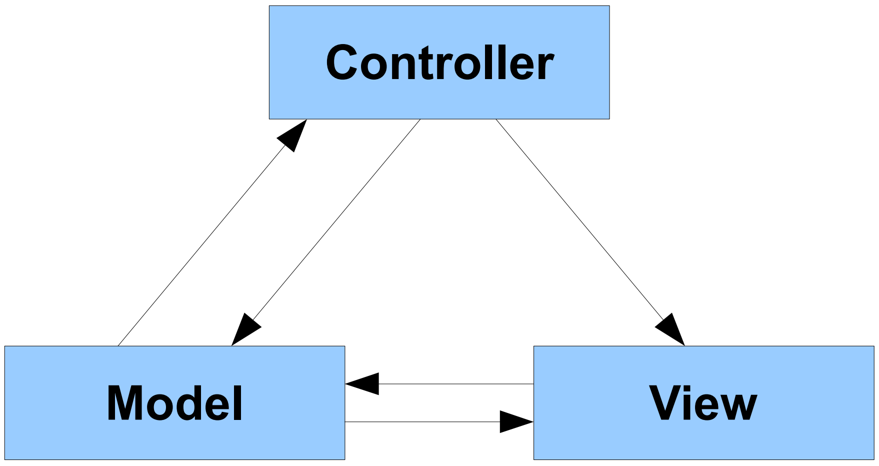

Figure 5: Model-View-Controller

3 Protocol

This chapter describes the protocol used for the communication between the helicopter (Quadrocopter) and the analyzing software (Valiquad). It is part of the *Architectural Design* as stated in the V-Model in the previous chapter. Implementation details are part of the hardware implementation as described in the chapter *Implementation* and are not specified any further here.

This protocol is independent from the implementation and the sample-rates the system is using. This approach is useful to separate the protocol from the system constraints. Of course there will be a matching of the protocol to the given system, but separating both leads to a higher portability of the protocol. This idea is also part of the ISO/OSI-Model.

The protocol described in this document acts on the layers 6 and 7 (Application and Presentation Layer) of the ISO/OSI-Model and is independent from the sub-layers it is working on. One possibility for those sub-layers might be the ZigBee protocol working upon an XBee module, which is currently used for the system.

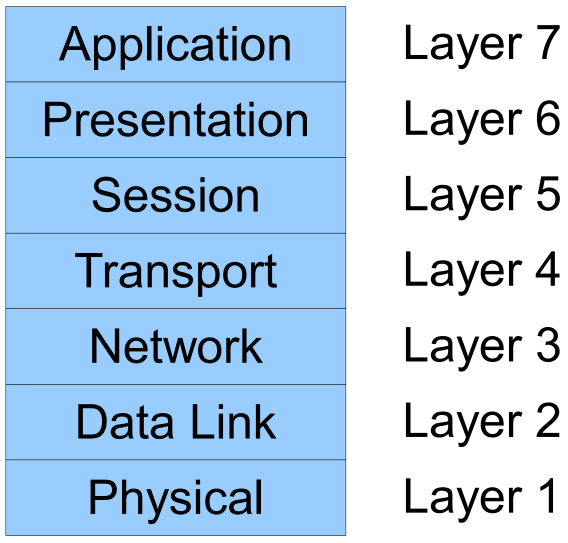

Figure 6: ISO/OSI-Model

The layers are linked together through so-called service-points. This means a sub-layer offers a service to the layer above it. On the wire this can be seen as additional packet-overhead in form of headers in front of the real payload data holding only meta-data and no payload itself. Those meta-data might be composed of source and destination addresses, payload length counters and check-sums.

3.1 Protocol in Detail

The protocol itself is the heart of the Valiquad project, as it is responsible for the correct handling of the data communication. All services the software will offer are based on this communication and there everything is based upon the protocol and its definition.

Due to this fact it is very important to verify its behavior and its correctness exhaustively. If there are any errors during the process of developing the protocol, they will not occur until the final integration test. If this test fails due to a malicious protocol definition the complete process according to the V-Model has to be done all over again. This would cause the project to stumble and fall. There will be no chance to complete the Valiquad software within the given time-constraints.

3.1.1 Packet structure

The protocol used for the data transmission is based upon packets that are sent from one instance to another. Such a packet consists of a message identification (MsgID) that describes the structure of a packet, the size of the payload data (length or counter), the payload itself and a check-sum that ensures the integrity of the data within a packet.

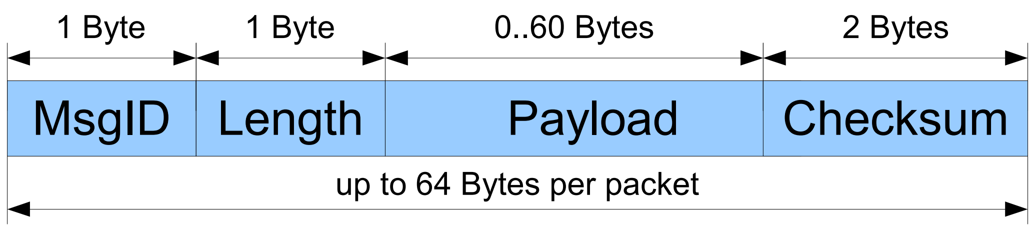

Figure 7: Packet structure

Currently such a packet has a maximum length of 64 Bytes, this is used to ensure that the protocol works with hardware layers that can only transmit packets with a maximum size of 64 Bytes. The XBee modules for example allow the transmission of packets with a size of 70 to 100 Bytes, dependent on the version of the module. With the constraint of having only 64 bytes of data for a packet the usage of all currently available XBee modules is ensured.

The central point of this protocol is the reading and setting of parameters and the cyclic sampling and transmitting of sensor values, called observables. Each parameter is described by a unique parameter identification number (ParID). Using this generic approach it is possible to make the software and the protocol as dynamic as possible. If a new parameter is added to the system, the analyzing tool does not need adaptation for the new purposes. By defining a new parameter with its according ID and its data type, the system is able to fulfill all its purposes immediately without knowing any further things about this parameter, than its parameter ID, which contains all relevant data like the data type, the group membership and a sequence number that is unique within group.

This dynamic and generic approach is an essential part of the protocol and a huge improvement to the current situation. Until now, the data transmitted over the communication medium is fixed and hard-coded. If any settings will change, the complete protocol of the old Valiquad software had to be adapted for its new tasks. This is not only a loss of time and flexibility, it is also a point that is very error-prone. One example is, that the system is currently partly converted from fix-point arithmetic to floating-point arithmetic. This adaption is such a big change to the system constraints that the current protocol would have to redefined at all. But this is only one issue that makes it necessary to develop a dynamic and generic protocol, there are many more and some of them are discussed later in this document.

3.1.2 Sensor data

As the helicopter contains several sensors that shall be read out periodically, it is more efficient to introduce a mechanism that tells the helicopter to send special parameters at a given sample rate without polling for every single value over and over again, which would cause a dramatic raise of payload to be transmitted. The software can tell the helicopter which parameters shall be sent within a given period of time, those parameters are also called observables.

This means the user can specify which parameters shall be send at which sample-rate. This is also an enormous enhancement compared to the former system that transmitted a hard-coded combination of values at a fixed sample-rate.

3.1.3 Frames

To reduce the complexity of the protocol and still guarantee the possibility to configure the system individually, four different frames are created: A, B, C and D. Every frame is linked to a periodically sent packet that can be configured by the application. Those frames are sent one after another periodically at a given sample rate. Every frame has a payload length of up to 60 Bytes.

For every frame a list of parameters can be defined, that will be sent within this frame from the helicopter to the analyzing tool. Those frames can be configured individually by adding / deleting parameters to / from the frame list. Such a list is available for every frame.

To sample a value at the highest possible sample rate it has to be put into all frames (A, B, C and D). If a value is sent within the frames A and C or B and D, its sampling rate is half of the highest possible one. By putting a value only in one of the four frames, it is sampled at a forth of the original rate. This is a dynamic approach with only little constraints.

3.2 Frame transmission

The following figure illustrates the frame-transmission over time:

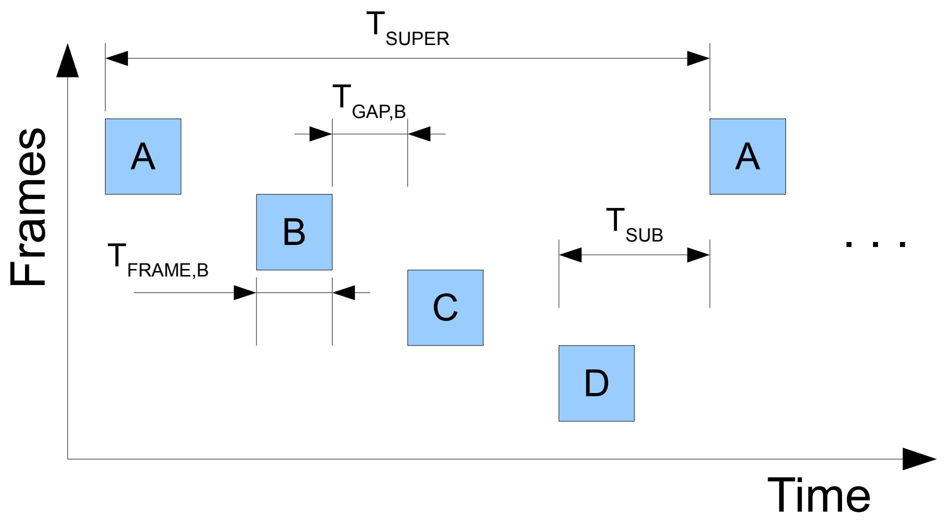

Figure 8: Frames over time.

T_{SUPER} is the super-period and T_{SUB} is the sub-period of the system, they are the same for every frame. This is an essential concept of the protocol, the super-period is the time the frames will repeated. The sub-period is the time between two consecutive frames, for example between the beginning of the transmission of frame A and the beginning of the transmission of frame B.

$T_{FRAME,B}$ is the time for transmitting frame B, in other words this time indicates how long the communication medium is blocked due to the transmission of frame B. $T_{GAP,B}$ is the gap between the end of the transmission of frame B and the beginning of the next sub-period, so it is the free space where nothing is transmitted over the communication medium. The relations between those values are:

- $T_{SUPER} = 4 \cdot T_{SUB}$
- $T_{SUB} = T_{GAP,i} + T_{FRAME,i}$ *with* $i \in \{\mathrm{A, B, C, D}\}$

To ensure that the processor has still enough time for doing other things than just transmitting frames, the following relation must be fulfilled for every single frame:

- $T_{GAP,i} \approx T_{FRAME,i}$ *with* $i \in \{\mathrm{A, B, C, D}\}$

3.2.1 Sample-rate categories

By using four frames it is possible to create 3 different sample rates for the observables. Those are one, two or four times the sub-period T_{SUB}. Having only two frames would restrict the configuration of the sample-rate too much, having more than 4 frames would cause an enormous raise of complexity. Using three frames is not a good choice, as this uneven number would result in the impossibility to sample a value at twice the sub-period, because the placing of the values within the frames is fixed and cannot be changed from one super-period to another.

The implementation of the protocol can have various interfaces for the user to configure the frames. One possibility is to configure every frame separately, this a very generic way that offers a lot of individual settings for the system. The more user friendly way is to tell the system which parameter shall be sampled at which rate and the system itself handles the matching of those parameters to the frames. Which interfaces are offered is part of the responsibility of the implementation and further described in the according chapter 5.

If the observables are automatically placed to their corresponding frames, the system must guarantee an adequate balancing mechanism. The mapping of the observables must be done equally, this means that a new observable that shall be sampled at a given rate shall be put into the smallest frames. The smallest frames are those, that have the highest amount of free bytes. This helps to utilize the frames in a good manner and can prevent the system from running into a one-way-street. If the system came into such a one-way-street street, no more new parameters can be added to the frames, even if there is theoretically still enough space for the placement of those observable parameters.

For example if frame A is full with observables that are sampled within the lowest possible sample-rate, which means that they were only put into this frame A, no new observable with the highest possible sample-rate can be added to the system, because nothing can be placed into frame A as this frame is already full. If those observables would have been mapped equally to the frames A to D, none of those frames would be completely full, but all frames would have about a quarter of their maximum possible payload filled with those observable parameters. Now the same observables are sent at the same sample-rate, but this time there is still enough space for placing new observables to the system, even when using the highest possible sample-rate, which would need the placement of this observable into every single frame, namely the frames A, B, C and D.

3.2.2 Parameter IDs (ParIDs)

Parameter IDs (ParIDs) consist of 2 Bytes, and identify a parameter within the system uniquely. There are two different types of parameters, the observableparameters and the normal parameters. Observables (short for observable parameter) are values a sensor delivers periodically within the cyclic frames (as defined in the previous chapter). Normal parameters cannot be sent within those frames, they are set and read out by polling the system.

The composition of those 2 Bytes is described in the following figure:

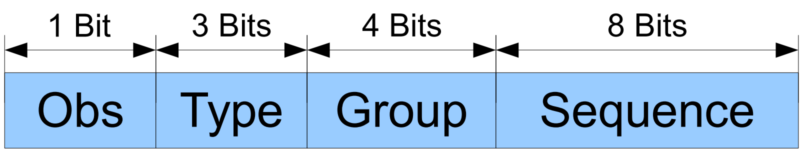

Figure 9: ParID Composition

The particular elements of a Parameter ID (ParID) are described in the next table:

Name	Range	Description
Obs	0x0..0x1	Parameter is an observable (Obs = 1) or not (Obs = 0).
Type	0x0..0x7	Defines the data type of the parameter.
Group	0x0..0xF	Assigns a group ID to the parameter.
Sequence	0x00..0xFF	The sequence number must be unique within a group.

Table 16: Parameter ID (ParID) Composition

3.2.3 Group IDs and conventions

The group ID and the sequence number form a unique ID for every parameter, therefore it has to be enforced that no sequence number is used several times for parameters in the same group. Those groups are used to arrange and structure parameters. Because only the group ID and the sequence number are used to describe a unique parameter, the type and the observable state can be changed easily without affecting the group ID and sequence number. This is important for example to change the type of a parameter from fix-point to floating-point arithmetic without affecting any other system parameters.

But changing the group of a parameter may result in the need of also changing the sequence number of it, as there might be another parameter in the new group with the same sequence number.

Those group IDs were introduced to group several parameters logically together. Those groups can be used to arrange parameters together, for example within the graphical user interface. They can also be used to form application specific conventions. An example for such a convention is that setting a parameters from group 0 needs all motors to be switched off. The parameters in group 0 also require calling the `copterSaveActualConfig()` function to act correctly, which is another part of this already implemented convention.

3.2.4 Type IDs

The following table assigns the existing data types to the available type IDs. Such a type ID is encoded in the Parameter ID (ParID) to calculate the size of a configured frame and check for the integrity of this configuration (for example the configured frame does not exceed in size (max. 60 Bytes payload)). It is also necessary to interpret a parameter correctly on the host.

Type ID	Type Description
0x0	float32 (4 Byte)
0x1	float64 (8 Byte)
0x2	uint8 (1 Byte)
0x3	int8 (1 Byte)
0x4	uint16 (2 Byte)
0x5	int16 (2 Byte)
0x6	uint32 (4 Byte)
0x7	int32 (4 Byte)

Table 17: Type IDs

Boolean values will be represented as an 8-bit unsigned integer (uint8) holding the values 0 for *false* and 1 for *true*. Any other type has to be encoded within those given types at the application layer. For example an integer value consisting of 64 bits, a so-called long integer, might be split into an upper 32-bit integer (int32) and a lower 32-bit unsigned integer (uint32) and put together on the other side accordingly by multiplying (or shifting) the upper integer with 2^{32} and adding the lower unsigned integer to it.

3.2.5 Message IDs (MsgIDs)

In order to fulfill the task of getting / setting / sampling parameters, a few methods have to be defined. Those methods are uniquely described through a so-called Message ID (MsgID):

Message ID (MsgID)	Explanation
0x01	Set sub-period. (T_{SUB}) If this is set to 0, no frames are sent and the frame transmission is stopped. This message is only acknowledged if the transmission is stopped, otherwise the acknowledgment is indirectly done with the first frame packet received. Setting the sub-period to another value requires to stop it first and restart the transmission with the new value.
0x02	Set the value(s) for a (list of) parameter(s).
0x03	Get the value(s) for a (list of) parameter(s).

Table 18: Message IDs (MsgID) Set / Get Parameter

3.2.6 Splitting configuration frames

As the parameter ID (ParID) of a value has 2 Bytes, the smallest types (uint8 / int8) are encoded in just 1 Byte and the size of the packets is fixed, every frame is divided into another two sub-frames (A1, A2 … D1, D2). Every sub-frame (first and second half of a normal frame) is configured separately, but the values that are sent from the helicopter to the analyzing software are put together into one single frame. If just values are used that are encoded in 2 or more Bytes ((u)int16 and (u)int32), configuration of a frame only requires the first configuration packet (A1 / B1 / C1 / D1). This mechanism is just some kind of a fragmentation of packets, because the configuration packets can be twice as long as the frames themselves and both packet types have the same maximum transmission size.

To be even more precise, the first configuration packet configures up to 30 parameters for a frame, if there is still space for more parameters in this frame, they are configured with the second configuration packet, which itself can also hold up to 30 parameters.

The packet configuration and calculation is done at the host, because there is more processing power available, than on the helicopter. Configuring the frames A to D therefore means splitting them into two sub-frames that are configured separately but sampled at once. To configure those frames, the frame transmission has to be stopped (setting the sub-period (T_{SUB}) to 0). The transmission has to be stopped first to avoid any conflict by accessing the communication medium. Another advantage of this constraint is the reduced calculation time on the helicopter. Analyzing the configuration frames and calculating the new configurations needs a lot of CPU power, which means that there is less power available for the sampling and this could cause the whole process to stumble or even stop.

Those messages are sent from the analyzing software to the helicopter to configure the sampling mechanism:

Message ID (MsgID)	Explanation
0x10	Configure parameter(s) for frame A, part 1. (A1)
0x11	Configure parameter(s) for frame A, part 2. (A2)
0x12	Configure parameter(s) for frame B, part 1. (B1)
0x13	Configure parameter(s) for frame B, part 2. (B2)
0x14	Configure parameter(s) for frame C, part 1. (C1)
0x15	Configure parameter(s) for frame C, part 2. (C2)
0x16	Configure parameter(s) for frame D, part 1. (D1)
0x17	Configure parameter(s) for frame D, part 2. (D2)

Table 19: Message IDs (MsgID) Frame Configuration Messages

Response messages to a request have the same ID. Because the participants of the protocol are asymmetric, response messages can be matched to an according request without using a separate Message ID for those responses. If no error occurred, an empty response packet is sent, if an error occurred, an error code is sent back within the response packet.

Those are the frame messages that are send periodically from the helicopter to the analyzing software at the defined sample-rate. Only observables can be sent within those frames, retrieving the values of normal parameters is done with the *getParameter* method as defined previously. Only the values of the configured observables will be transmitted as a stream of byte. This means there is no information about how to handle a special value, which type it has or how many bytes belong to a value, so there are only the raw sampling values without the parameter IDs.

All this meta information, which is encoded in the parameter ID, has to be stored on the client side (within the analyzing software) when configuring the frames. This reduces the amount of data to be transmitted and enables a higher trough-put. The system must guarantee a consistent state, so only if setting a frame configuration was successful as indicated by the acknowledge frame, the analyzing software can rely on the correctness of the previously sent frame configuration.

By starting the sampling mode by setting the sub-period to a value higher than zero, the configured frames are sent from the helicopter to the analyzing software. The following table shows the message IDs for those sampled frames.

Message ID (MsgID)	Explanation
0x30	Frame A with the according sampling values.
0x31	Frame B with the according sampling values.
0x32	Frame C with the according sampling values.
0x33	Frame D with the according sampling values.

Table 20: Message IDs (MsgID) Frame-Transmission

3.2.7 Well-known message IDs

Message IDs below 0x80 are the well-known IDs and are used for this protocol. Message IDs above 0x80 are reserved for user-defined purposes and are ignored by this protocol. They are used to add new functions to the helicopter without affecting the Valiquad protocol. This is important as there is only one medium available for all radio-frequency based functionality.

3.2.8 Response messages

A message that was sent from the analyzing software to the helicopter always requires an according response message. As the roles within the process of communication are defined in advance and there are only two participants (server and client) that can be uniquely identified, the same message IDs for the response messages are used. This means if requesting a parameter with the message ID 0x03 a response message is received with the same message ID and the payload data following.

There is another important thing to be mentioned. The first byte of the payload data in the response message always holds an error code. Additional data that has to be sent within the response message is transmitted in the second part of the payload data. Due to this fact, the payload of the response message will be at least one byte long, as there is always an error code returned to the analyzing software.

The figure below illustrates how the payload data of the response message is split into two parts, the error code and the response data requested by the analyzing software:

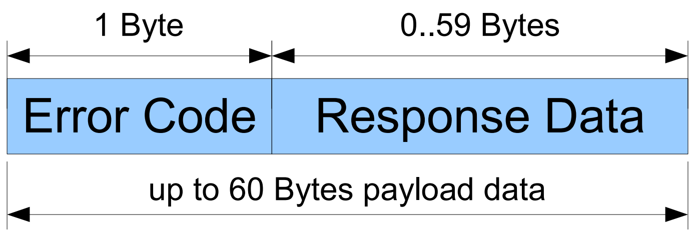

Figure 10: Response Message

The error codes and their meaning are following in the table shown below:

Error Code	Explanation
0x00	No error occurred. If some kind of data was requested, it can be found in the following bytes within the payload data of the response.
0x01	Format error: The format of the message that was received by the server (helicopter) was not correct.
0x02	Overflow error: Something exceeded its maximum value, e.g. there were too many observables configured within a frame.
0x03	Parameter error: The requested parameter does not exist (unknown paramter ID). This can also mean that there is a type mismatch between the requested and the available parameter.
0x04	Subperiod error: The requested sub-period cannot be set.
0x05	Motor error: The motors are still running and therefore a configuration or a parameter could not be set. This only applies to parameters belonging to group 0 and is a so-called convention as defined in chapter 3.2.3.

Table 21: Error codes of the response messages

Those error codes only refer to logical errors within the request messages. If there was an error on a lower layer of the communication, the packet will be discarded. Such an error might be a detected CRC failure or a loss of the communication link. To detect such low level errors, a timeout failure is introduced. When the analyzing software sends out a message and does not receive an answer within a given time, a loss of the packet is assumed. It does not matter where this error exactly occurred, as it will always result in a resending of the message. If there were three sequential timeouts after another, the analyzing software will assume the helicopter to be offline or at least unreachable. This is necessary to prevent the system from entering an endless loop while polling for an unavailable helicopter over and over again.

But sampled frames (A to D sent from the helicopter to the analyzing software) do NOT have such an error code. They are not a direct response to any request and therefore need no error code. This also means they can use the complete payload data consisting of 60 bytes.

3.2.9 Frame merging

If transmitting a frame at a given sample-rate is not possible, because this would cause the calculation of the control-loop parameters to stumble, several frames might be put together in one super-frame and sent at a lower sample-rate. For the sampled values it is no problem, if they arrive a bit later.

If frames are merged and sent at a lower rate than the sampling rate of the containing values, the sensor data has to be buffered within the helicopter. The possible merging of packets is part of the user specific functions and is not described any further here. If desired, this mechanism has to be realized at the layer above the Valiquad project.

4 Design

This chapter describes the software design using common UML charts. One of the basic design patterns used as an interface between the Valiquad library and any external software using this library, is the *Model-View-Controller* pattern.

This pattern is an approved method to separate a graphical user-interface from the data collection. The Valiquad library is responsible for the communication and therefore holds all data collected at the lower layers. This is the model of the MVC-pattern, it is responsible for providing the data will be showed to the user for example within the view. The controller enables the possibility to change the view on the data or the data itself and is also separated from the view and of course from the model.

If any software is linked against the API of the Valiquad library, it has to keep to this idea more or less. But it is strongly recommended to use this pattern for any software using the library, as it guarantees an interface that can be tested separately from the other modules. For example a GUI using the Valiquad library can be developed without the physical connection to the helicopter by using a list of pre-sampled or dummy values and integrating the complete system not until this module works with is simulated data.

The relationship between those three parts of the MVC pattern is showed in the following figure.

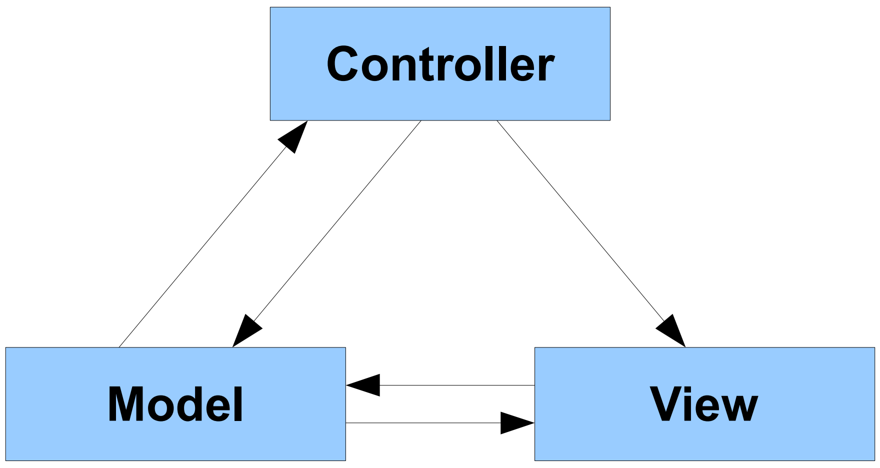

Figure 11: Model-View-Controller

The communication layer is responsible for the data base, which is part of the model. The view and the controller are parts of the user interface. The view is responsible for showing the current data in an adequate way, whereas the controller handles user input and acts according to this input, this could be a possibility to zoom in and out the diagrams showing the sampled values.

4.1 Analyzing software

The communication interface for the analyzing software is written in Java. The software is object oriented and was designed using UML 2.0 class diagrams. UML is a set of different forms of diagrams that are used to model software, especially object oriented systems like the Valiquad project. Modeling the system with UML diagrams is part of the *Architectural Design*.

The following figures in this section use the class diagrams for designing and modeling the object oriented relationships between different classes working together to achieve one common goal.

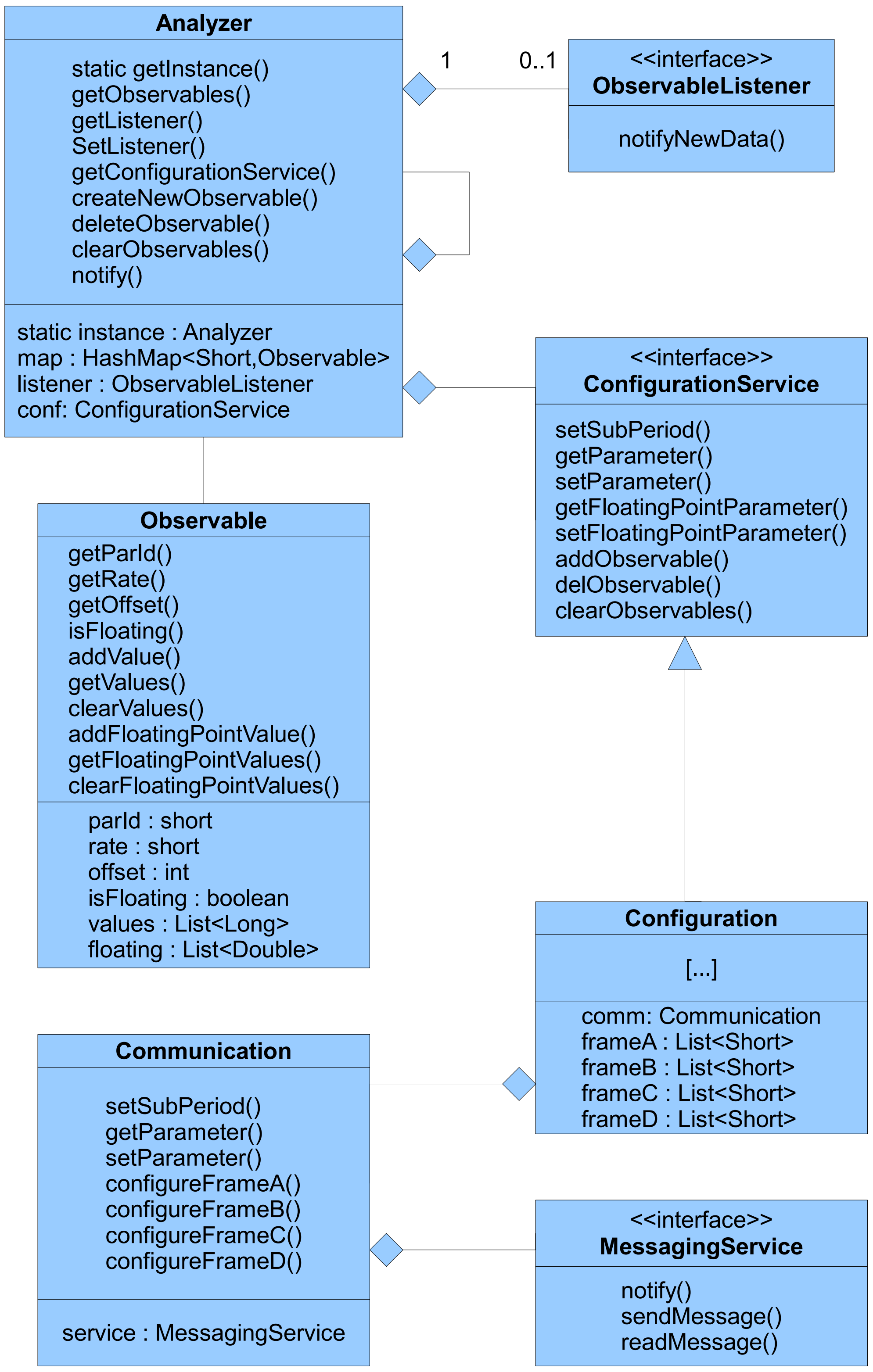

Figure 12: Analyzer Design

The following section describes the interaction of the classes used within the analyzing software. All names refer to those defined within the class diagram designed in Figure 12: Analyzer Design above.

The central instance is the *Analyzer* which is a singleton class and unique within the whole system. This central point is notified about incoming frames and analyzes their content. The values of the observables that were analyzed and read out are stored in a central data base in form of a hash map. If a listener (*ObservableListener*) was set before, it is informed about the arrival of new data. This is the heart of the model according to the MVC pattern, as it is responsible for holding the consistent data. The listener might be the view that is informed about new data and will update the presentation widget for the user.

The data base holds an *Observable* object for every sampled observable and its according parameter ID. This object stores the buffered values for its corresponding observable and it also has any additional information stored, that might be useful for displaying the values within the right context. If stopping the sampling mode the buffer will store the values that were received until this point in time. If restarting the sampling mode the buffers for all observables will be erased to ensure a consistent data model. If this would be omitted, expired data is compared to new data, confusing the user.

To set system parameters within the helicopter, the *ConfigurationService* must be invoked. The *Analyzer* holds an instance of this interface that can be retrieved and used for setting up the system. With the *ConfigurationService* the sub-period and parameters (integer and floating point values are handled separately) can be set. There are also methods within this service to configure the observables and place them within the frame A to D. How those methods are exactly invoked can be read in the according API documentation as it is part of this software.

The *Configuration* class is derived from the *ConfigurationService* and is the current standard implementation of the *ConfigurationService*. It holds itself an instance of a *Communication* object. The *Communication* class handles the low level part of the message and packet transmission and therefore uses functions provided by its underlying *MessagingService*. This service interface composes the packets and form the communication basis of the system, modeled in detail within the next figure.

The *MessagingService* and its underlying classes are working separately in an own module. This module's design can be found in the next figure.

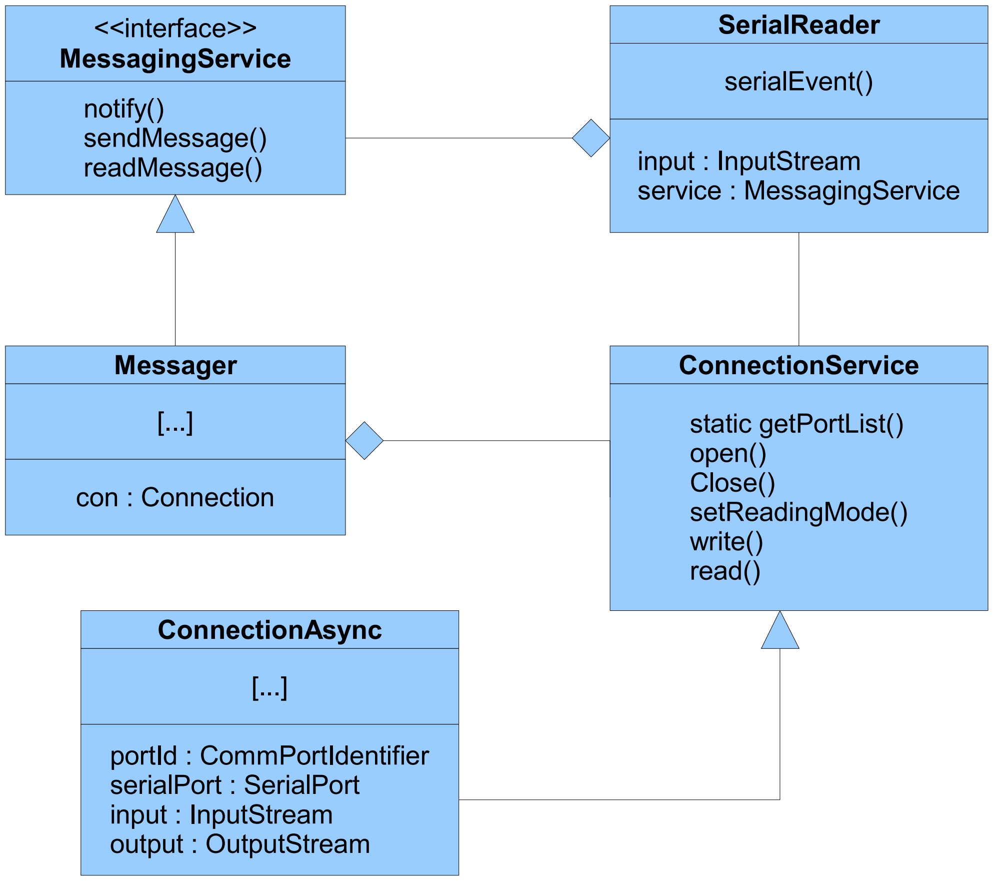

Figure 13: Messaging Design

The *MessagingService* interface is the link between the low level communication module and the high level configuration module. The *Messager* is derived from this interface and holds an object that is derived from the *ConnectionService* interface. This *ConnectionService* is responsible for handling the low level communication. The current implementation of this interface is the *ConnectionAsync* class and is based upon the asynchronous serial connection provided by the javax.comm API.

The main task of this lower level module is the calculation of the check-sum, the sending of single bytes and the access to the medium.

The documentation of the source code can be found as an HTML-attachment which was generated using *javadoc* annotated comments.

4.2 Helicopter software

The communication interface for the helicopter is written in C. It is programmed in a procedural way as it is common for embedded systems, because of their resource constraints. The software itself is embedded in the existing helicopter software which itself is running on a HCS12 16-bit micro-controller.

The former module called *Basestation* was adapted for the new purposes and a new module named *Valiquad* was added to the helicopter software.

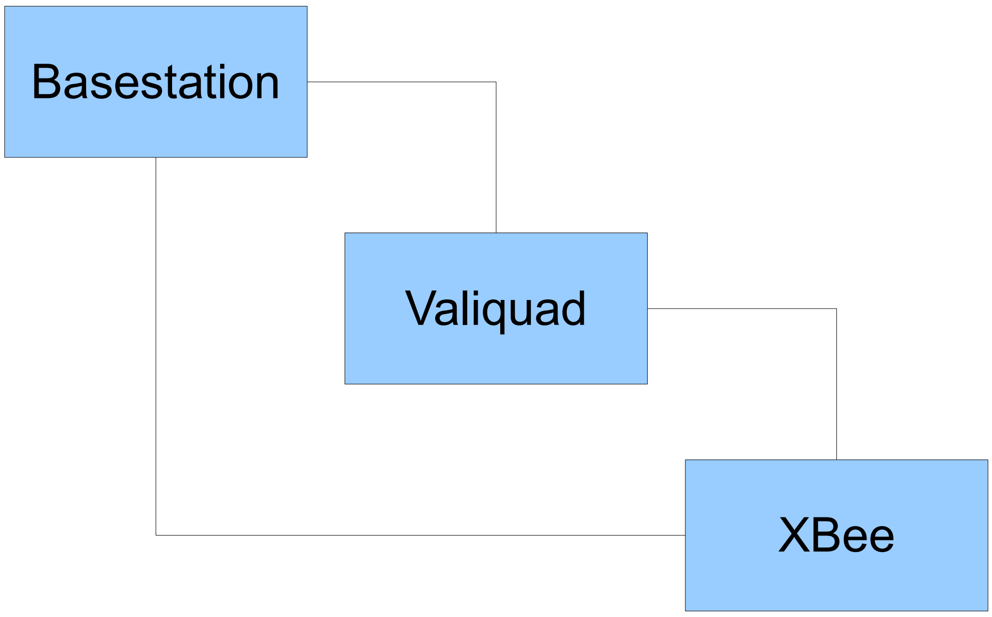

Figure 14: C Module

Basestation is the main module that is executed periodically as a task. This task is currently hard-coded but can also run on a real-time operating system in the future. The *Valiquad* module is invoked by the *Basestation* which itself is using the *XBee* module for handling the communication.

The *XBee* module is the lowest level of the communication handling and is responsible for sending and receiving well-formatted ZigBee-messages. There are two different categories of messages. Category 1 is used for the Valiquad analyzing software, those messages are identified by a message ID that is less than 0x80. Category 2 are messages for application or debug specific purposes and have a message ID greater then 0x80. This separates the Valiquad protocol from any additional message traffic on the same communication medium.

If the message is from category 1 it is directly delivered to the *Valiquad* module, which then handles all necessary steps to fulfill the protocol needs correctly. Messages from category 2 bypass the Valiquad analyzing software and therefore they need direct access to the *XBee* module without invoking the *Valiquad* module first. Those messages can be used to add specific debugging information without the need of all the protocol overhead that comes with the dynamic approach of the Valiquad analyzing software.

The documentation of the source code can be found as an HTML-attachment which was generated using *doxygen* annotated comments.

5 Implementation

This chapter describes the software implementation for the helicopter (Quadrocopter) in C and the communication interface of the analyzing software (Valiquad) in Java. It is therefore part of the *Implementation* as stated in the V-Model in chapter 1.

One of the main affords within this hardware-related part of the java software was the minimizing of all dependencies. Therefore nearly every function was programmed directly within this library without using any other external software except for the javax.comm API which is responsible for the serial port communication. It does not matter which implementation of this standard API is used, as a default it is recommended to use the one from the GNU project (rxtx). To say it again, everything except the javax.comm API was programmed from scratch. This also means that the complete software can be released under the GPL, if using the GNU implementation of the javax.comm API.

The helicopter software is based upon the software that was developed at the University of Applied Sciences Esslingen. All parts of this software are also released under the GPL including the new “Valiquad” module that was added to the helicopter software package and created from scratch to fulfill the requirements and the protocol as defined earlier. Another major change was done within the “Basestation” module, that was nearly completely reprogrammed and adapted for the new purposes.

5.1 Protocol

The protocol is part of the upper layers of the ISO/OSI model. Therefore only the software for those layers had to be written on the helicopter. The “XBee” module was already programmed and could be taken as is without any further changes but some minor ones affecting the baud-rate. This was able due to the approach of separating the protocol into several independent layers that could also be tested separately.

5.1.1 Throughput and baud-rates

For the desired communication protocol the following throughput is required:

One packet consists of 1 Byte ID + 1 Byte PLC + 60 bytes payload + 2 bytes check-sum = 64 Bytes. This data has to be transmitted at a given sampling rate of currently 100ms. In the future a packet with the same size might be sent every 10 ms. The throughput for this future system configuration has to be at least 64 * 8 bits * 100 sec^{-1} = 51.200 bits/sec.

The next higher standard baud-rate within the XBee Pro module is 57.600 baud/sec. Using this baud-rate would mean, that there is nearly no time to spent for different things between the transmission of 2 packets. The next higher standard baud-rate is 115.200 baud/sec and is the highest possible baud-rate that can be used with the XBee module.

This baud-rate would guarantee enough time for the calculation of other things between the sending of 2 successive packets. Currently a baud-rate of 9600 baud/sec is used with a payload of 64 * 8 bits * 10 sec^{-1} = 5.120 bits/sec. The time-factor for this current setting is (5.120 bit/sec) / (9.600 baud/sec) = 0.533. The time-factor of the future setting will be (51.200 bit/sec) / (115.200 baud/sec) = 0.444 and needs less time for the data transmission than the current setting, while having a ten times higher throughput. This time-factor has no unit, because for the serial data transmission one baud is equal to one bit.

The Dragon12 development board with its HCS12 processor has 2 serial ports. The port 0 is used for the debugger, but the port 1 can be used for any application. This serial port (SCI1) is connected to the XBee module on the helicopter. There are several control registers for each serial port that have to be set up for the correct data transmission. One of the most important things is the baud-rate. This baud-rate is calculated through a bus-clock divisor. The control register for the baud-rate of the serial communication interface 1 is called SCI1BD and is calculated as follows:

- $SCI1BD = \frac{f_{BUSCLK}}{(16 \cdot f_{BIT})}$ with $f_{BIT} = f_{BAUD}$ and $f_{BUSCLK} = 24\,MHz$

With a baud-rate of 115200 baud/sec this register has to be set to SCI1BD = 13.

The exact value for this setting would be SCI1BD = 13.0208 which results in a relative error Δe = (13.0208 – 13) / 13 = 1.6 * 10^{-3}.

The relative error for the previous settings with a baud-rate of 9600 was about the same: Δe = (156.25 – 156) / 156 = 1.6 * 10^{-3}. This error causes the stream of bits to drift slightly away from the expected baud-rate and can cause an error while transmitting the data. This error normally does not affect the data transmission as long as the packets do not exceed in length, as they are re-synchronized at the beginning of the transmission of a new data-block.

Setting the baud-rate on the host is done by invoking the `setSerialPortParams()` method on a `SerialPort` object that comes with the java communication interface (javax.comm). The baud-rates also have to set within the XBee modules using the X-CTU software that configures them accordingly. On the helicopter the *XBee* module has to be adapted if using another baud-rate is desired. Within this module there is a variable that can be easily set to a new value and that's it.

If using such a high data transmission rate, we have to worry about the time-consumption of calculating the check-sums for every single packet. Currently this is done in software. Calculating this check-sum in hardware would be more time-efficient and would enable the raising of the baud-rate to its maximum.

5.1.2 Optimization of the sampling rates

One of the rules established in the UNIX world, namely the rule of optimization, advices a programmer to first prototype and then polish a software. In other words, first make it work and then make it fast. Keeping to this idea, the protocol for the analyzing software will be first implemented based on the old sample- and baud-rates of the system. If the protocol is working correctly with those settings, the optimization and tuning of the system can be done. Raising the sample- and baud-rates should not be a big thing if the generic approach of the protocol is implemented correctly, therefore it is not restricted to a given configuration of the lower layers of the protocol stack it is working on.

5.1.3 Frame implementation

The central point of the protocol are the frames that handle the sampling of observable parameters. The interface for the frame configuration offers several possibilities to fulfil the task of creating a parameter sampling constellation. The most generic way is adding each parameter separately to any of the four frames, but this is not quite user-friendly. A better way is to allow the configuration of parameters at a given sampling rate, while the mapping to the frames is automatically done by the system.

For those two different ways of configuration, different methods are available in the hardware library. The interface details can be found in the API documentation of the system.

As described in the protocol documentation, the system needs an adequate balancing mechanism for the placement of parameters to the frames. This placement is done in a special way, that is described in the following section:

If an observable shall be added to the system using the lowest possible sample-rate, it has to be put to one of the four frames. The observable is then put into the frame with the highest amount of free bytes.

If an observable shall be added with the medium sample-rate, which means it has to be put into two of the four frames, while the only possibility is to put the observable parameter to frame A and C or frame B and D, which brings the complexity down. The system calculates the length of A and C together and compares this to the length of B and D together. If A and C together have a higher amount of free bytes, the system tries to place the observable to those two frames. If this placement is not possible, it tries to place the observable to the frames B and D. Now it can happen, that frame B and D have a higher amount of free available bytes, but it is not possible to place the observable to them. If this occurred, the system has to try to place the observable to the frame A and C, because it might be possible that this works, even if those two frames have a lower amount of free bytes than the other combination.

For sampling an observable at the highest possible sample-rate no calculation has to be done, because the parameter has to placed in every single frame, there is no choice where to put them in.

5.1.4 Parameter implementation

Setting and getting of parameters is the second main task of the analyzing software, beside the sampling of measured values. Every parameter is identified throughout a unique parameter ID as described earlier. There are special message IDs that are used to set and get parameters. To reduce the traffic that has to be transmitted, several parameters can be read and set at once.

The following figure illustrates how the payload data of a parameter request message is formatted.

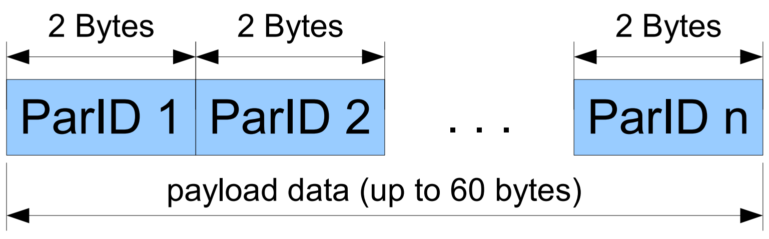

Figure 15: getParameters request

If the values of several parameters shall be retrieved at once, a getParameters-message has to be sent to the helicopter with a list of the parameter IDs that shall be retrieved. This list is limited by the maximum amount of bytes that can be transmitted within the payload data. Within the protocol this maximum amount is 60 bytes. This means it is not possible to retrieve more than 30 parameters at once, as a parameter ID is 2 bytes long.

But there is another constraint. The response-message to the request above is shown below. This response can only have 59 bytes of payload data, because the first byte of the response-message is reserved for the error code. This means the parameters that can be requested are limited by the total amount of bytes those requested values need altogether.

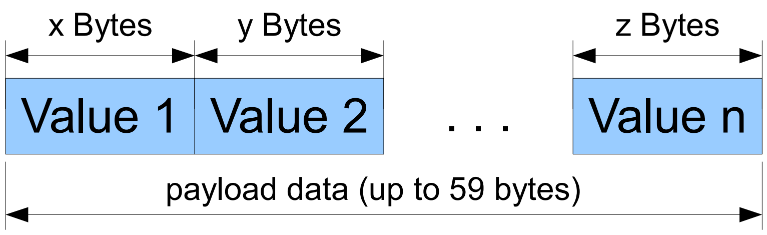

Figure 16: getParameters response

Setting the parameters to special values is done quite similar. The following figure illustrates the mechanism in detail.

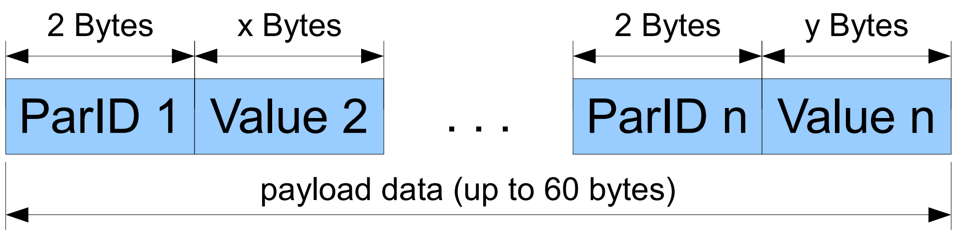

Figure 17: setParameters request

Parameter IDs and their corresponding values are alternating within the payload data of a setParameters-message. A parameter ID is always followed directly by its corresponding value. The whole message can only have 60 bytes of payload data, which limits the total amount of parameters can be set at once. This mechanism ensures, that the recipient of the message can always calculate from the parameter ID the length of the following value and can determine its value.

This is an iterative process, first the parameter ID is read, then the length of its according value is calculated, with this length the value itself can be retrieved and the software then knows where the next parameter ID starts and does this iterative process again until the payload data reached its end.

5.1.5 Data types

One big problem in Java is the absence of unsigned data types. For the helicopter we need unsigned data types and therefore we have to introduce a way to handle them correctly in Java. All data types in Java are signed, this means storing an unsigned value needs the next higher signed variable.

The following table describes the helicopter data types and their counterpart in the Java-based analyzing tool:

Data type of the helicopter	Data type used in Java
bool	byte
int8	byte
uint8	short
int16	short
uint16	int
int32	int
uint32	long

Table 22: Data type matching

As the biggest data type that has to be used is long, this is also the one used for generic methods that have to handle any data type.

The protocol is byte-oriented and therefore those data types have to be mapped to their according byte representation before sending them over the wire. On the other hand the data types have to be restored from a received stream of bytes.

Within the parameter IDs (as described in the protocol chapter) the data type of a parameter is encoded. Using this information it is possible to read and write the parameters correctly.

As the representation of a value is part of the lower layers and only done to reduce the memory usage, all values will be presented to the upper layers in the same way, because there are no such strict memory constraints anymore.

This means there are only two different number formats that the higher layers have to handle. On the one hand there are all the integer representations that are hold within a *long* variable. On the other hand there are the floating point numbers stored within *double* variables.

As those data types are handled in a completely different manner, they will be strictly separated. This means that methods and functions have different interfaces to process integer and floating point values. For example the getting and setting of parameters is done at the lower level of the communication protocol in the same way, but for the user of the hardware library there are two different methods to be invoked. The methods used for floating point operations convert the double values to their corresponding bit representation using a long integer variable as a container for holding the double value. Those integer values are then transmitted over the communication medium and put together on the other side accordingly.

5.2 Helicopter software

The helicopter software runs on a HCS12 micro-controller and is programmed in a procedural way using the standard embedded programming language C.

5.2.1 Sampling mechanism

There are two major constraints in the implementation of the protocol on the helicopter. The first thing is the execution time of the task. This time is limited by the maximum execution time that is reserved for this task, which itself is dependent on the period and the other tasks running on the system. The other problem is the limited size of memory that can be used for the buffering.

As described earlier, observable parameters can only be read in a cyclic manner. To keep the execution time as small as possible, the pointers to the sampling values of those observable parameters are stored in a buffer with their corresponding parameter ID and parameter length. This means that sending a frame only requires to copy the current sampling values at their given locations in the memory into a message payload buffer and then transmitting the message without any further steps. This also requires the frames to be configured correctly in advance, which itself might take some extra time. This higher configuration overhead is one of the reasons, that the sampling mode must be stopped before a new parameter configuration can be loaded. This mode is called the configuration mode. Another reason for introducing those two modes is to ensure a consistent process image, this can only be achieved by separating the configuration from the sampling mode.

The frame buffers must be big enough to hold the sapling values of all frames. For each of those observable parameters, their location in the memory, their length and their parameter ID has to be stored within this buffer. Every frame can have up to 60 observables, this means we need enough memory to store 4 times 60 observables with the corresponding configuration data as described above. The configuration data for one observable is about 6 bytes long, which results in a total buffer size of about 4*60*6 bytes = 1440 bytes for storing the current observable configuration.

5.2.2 Parameters and observables

The parameters must be created within the helicopter. They are constant and will not change ones they were created. For every parameter and also every observable that shall be available through the protocol, a new entry has to be created within the helicopter data base. This data base is encoded in the code block of the ROM and is not part of any kind of dynamic memory.

If a new parameter or observable shall be added, first its group has to be defined. For every group there is a separate callback function that is invoked if a parameter or observable is requested from the system. Within this group callback function a new parameter or observable is created within the given switch-case-block by invoking the macro CREATE_PARAMETER and CREATE_OBSERVABLE respectively. Those macros need the sequence number, which must be unique within a group, the type and also the (memory location of the) parameter or observable itself. Introducing those macros eases the handling of creating new parameters and observables while keeping the data within the code-block in the ROM, so none of the valuable RAM memory bytes are used.

6 Integration and Testing

The integration and testing of the software was done in two steps. The first step was the deployment of the software that was developed for the micro-controller mounted on the helicopter. The second step was the testing of the communication by running only the lower layers of the analyzing software on a remote computer and adding the higher layers step by step.

6.1 Separation

The approach of separating the software into several independent parts was possible because of the precise definition of the application programming interfaces (API) between the software pieces. The precise definition of the communication protocol between the helicopter and the analyzing software was another important thing to separate the development on the host and target side. Testing one instance could be done by simulating the other instance of the communication protocol and vice versa.

6.2 Sampling rates

The testing of the time-consumption showed that a minimum sub-period of 10ms is possible for the over-all sample process. Setting the sub-period to only 5ms made the system stumble, which causes a loss of about half of the packets. This means every 10ms a frame can be send with a resulting super-period of 40ms.

So every 40ms the frame-cycle will start again. This possibility of setting the sub-period to 10ms enables a ten times higher sampling rate than the one used before in the old version of the analyzing software. This even works with the check-sum being calculated in software. If using hardware-acceleration for calculating this check-sum the sample-rate can even be set to 5ms or less, but one has to keep in mind that this cannot be guaranteed unless the exact calculation time of all software parts are known.

7 Release information

The hardware-related part will be released in two packages. The java software is kind of a library which will be packed as a jar file. The classes that are only used for testing purposes will not be shipped and all debug information will be suppressed by setting the debug level within the software to zero. The other part is the current software release of the Quadrocopter with the newly developed additional communication protocol, which is also shipped as a separate c module that can be integrated in any other version of the Quadrocopter software.

7.1 Library

This library is responsible for the complete handling of the communication protocol and will offer the interface as described in the appropriate API documentation. This API documentation is shipped with the library as *html* pages as generated by the *javadoc* program. This API documentation describes how the library can be exactly used. It is independent from the upper layers using it. Not only a graphical user interface can be created to interact with the system, but also kind of an automation software could be written. The library can so be used as a middle-ware to connect other software to the system, for example to directly use the measured values in *Matlab* or any other simulation software.

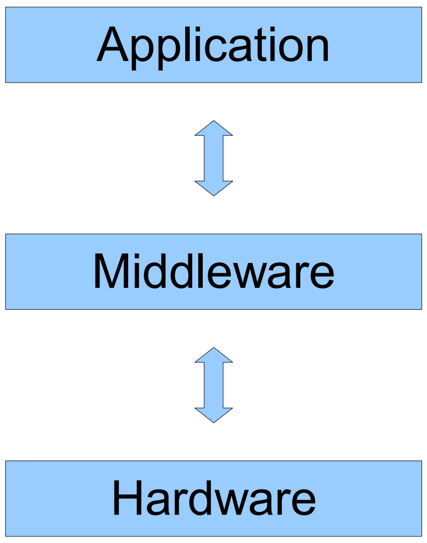

Figure 18: Middleware

7.2 Appendix explanation

A compact-disc (CD) is added in the appendix containing the following data:

- The Valiquad Java library as a jar file.
- The javax.comm API implementation of the GNU project (rxtx) as the only external dependency of the Valiquad project.
- This documentation as a pdf file.
- The original Quadrocopter documentation as pdf files.
- The Java source-code documentation for the API (index.html)
- The Java source-code that can be imported as an Eclipse project.
- The C source-code documentation for the Valiquad module (index.html)
- The C source-code and the according CodeWarrior project metadata.

8 References

[1] Quadrocopter Documentation (Documentation_Final.pdf)

[2] Quadrocopter Hardware Interface (Hardware_Interface.pdf)

[3] Wikipedia: V-Model (http://en.wikipedia.org/wiki/V-Model)

[4] Wikipedia: OSI-Model (http://en.wikipedia.org/wiki/OSI-model)

[5] Wikipedia: MVC Pattern (http://en.wikipedia.org/wiki/MVC_Pattern)

[6] Wikipedia: UML Diagram (http://en.wikipedia.org/wiki/UML_Diagram)

[7] Wikipedia: Middleware (http://en.wikipedia.org/wiki/Middleware)