Project Name:	Valiquad
Document Title:	Implementation
Autor(s):	Manuel Stübler
Created on:	15.11.10
Last modified:	13.12.10
Version:	v1.3
Status:	| | in progress | | presented |x| closed
Located at:	/docs/se/impl-hw.odt

Revision History:

#	Date	Version	Section	Description	Author	Status
1	15.11.10	v1.0	all	First creation	Manuel Stübler	closed
2	03.12.10	v1.1	2	Added data type explanation	Manuel Stübler	closed
3	09.12.10	v1.2	2	Data type representation	Manuel Stübler	closed
4	09.12.10	v1.3	3	Helicopter software	Manuel Stübler	closed

Table of Contents

List of Figures

List of Tables

1 Introduction

This document describes the software implementation for the helicopter (Quadrocopter) in C and the communication interface of the analyzing software (Valiquad) in Java.

2 Protocol

2.1 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 $\Delta e = (13.0208 - 13) / 13 = 1.6 * 10^{-3}$.

The relative error for the previous settings with a baud-rate of 9600 was about the same: $\Delta e = (156.25 - 156) / 156 = 1.6 * 10^{-3}$. 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).

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.

2.2 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.

2.3 Protocol 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 need 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.

2.4 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 1: 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 documentation protocol.odt) the data type of a parameter is encoded. Using this information it is possible to read and write the parameters correctly.

To be more memory efficient, the software will store those values in their smallest possible representation within Java, this means values will use the Java type as defined above and will not use a long value for all types. This is done with a generic observable class that stores the sampled values in its internal buffer structure using the smallest possible memory representation of those values.

3 Helicopter software

3.1 Observables

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.

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.