Lucerne University of
Applied Sciences and Arts

HOCHSCHULE
LUZERN

Technik & Architektur
FH Zentralschweiz

MSE | MASTER OF SCIENCE
IN ENGINEERING

UAV Serial Switch Documentation

Stefanie Schmidiger

MASTER OF SCIENCE IN ENGINEERING
Vertiefungsmodul I

Advisor: Prof. Erich Styger

Experte: Dr. Christian Vetterli

Horw, 2018

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig angefertigt und keine anderen als die angegebenen Hilfsmittel verwendet habe. Sämtliche verwendeten Textausschnitte, Zitate oder Inhalte anderer Verfasser wurden ausdrücklich als solche gekennzeichnet.

Horw, 10.01.2018 Stefanie Schmidiger

Versionen

Version 0	Initial Document	10.01.18	Stefanie Schmidiger

Abstract

With unmanned vehicles, there are always on-board and off-board components. Data transmission between those components is of vital importance. Depending on the distance between vehicle and ground station, different data transmission technologies are ideal. So far, each device was connected to a single modem and the data transmission technology used could not be switched during operation.
In a previous project, a serial switch had been designed with four RS-232 interfaces that act as data input and output for devices and four RS-232 interfaces for transmitters and modems. This hardware is very flexible: data routing and transmission behavior is configurable by the user. The application running on the serial switch collects data from connected devices, puts it into a data package and sends it out via the configured transmitter. The corresponding second serial switch receives this package, extracts and verifies the payload, sends it out to the corresponding device and optionally sends an acknowledge back to the package sender.
A Teensy 3.2 development board has been used as a micro controller unit. The software was written in the Arduino IDE with the provided Arduino libraries. As the project requirements became more complex, the limit of only a serial interface available as a debugging tool became more challenging. In the end, the software ran with more than ten tasks and an overhaul of the complex structure was necessary.
This document describes the refactoring process of the previous project. In the scope of this work, an adapter board has been designed so the previous hardware could be used with the more powerful Teensy 3.5 development board and a hardware debugging interface. A new software concept for the Teensy 3.5 was developed and implemented.
The Teensy 3.5 is configured to run with FreeRTOS. The developed software uses the task scheduler and queues of the operating system to provide the same functionalities as the previous software for Teensy 3.2.
The new software concept for the Teensy 3.5 is easy to understand, maintainable and expandable. Even though the functionality of the finished project remains the same as in the first version with Teensy 3.2 and Arduino, a refactoring has been necessary. Now further improvements and extra functionalities can be implemented more easily as suggestions are given and issues are reported within this document.

this is a copy-paste of va1!! rewrite!!

Horw, January 2017 Stefanie Schmidiger

Summary

With unmanned vehicles, there are always on-board and off-board components. Data transmission between those components is of vital importance. Depending on the distance between vehicle and ground station, different data transmission technologies have to be used.
In a previous project, the hardware for a Serial Switch has been designed that features four RS-232 interfaces to connect data processing and generating devices and four RS-232 interfaces to connect modems for data transmission. The application running on the designed base board assembled data packages with the received data from its devices and sent those data packages out to the modems for transmission. The corresponding second Serial Switch received those data packages, checked them for validity and extracted the payload to send it out to its devices.
A Teensy 3.2 development board acted as the main micro controller. Teensies are small, inexpensive and powerful USB development boards for Arduino applications. The software developed was flexible and in its header files the user could configure individual baud rates for each RS-232 interface, data routing and the use of acknowledges for data packages for each modem side. The application was running with many tasks, was complex and not easy to debug because of no hardware debugging interface.
Then this follow up project was initiated with the aim of an application with better maintainability and expandability. The main requirement for this follow up project were the use of a more powerful micro controller with Free FROS as an operating system, the use of an SD card for a configuration file and data logging and a hardware debugging interface.
In the scope of this project, the Teensy 3.2 was replaced with a Teensy 3.5 development board, which featured an on-board SD card slot. The Teensy 3.5 was prepared for hardware debugging and an adapter board to use the new Teensy 3.5 with headers meant for the Teensy 3.2 was designed. This adapter board allowed the use of the same base board as was designed in the previous project.
For the Teensy 3.5 application, the concept with data packages is applied as well and the same configuration parameters are used. The configuration is read from an .ini file saved on the SD card.
The functionality of the application remains the same as in the Teensy 3.2 software but with better maintainability and an easier software concept with less tasks. Hardware debugging is now possible which is of vital importance for this application to be further expandable.
The Teensy 3.2 application was neither well documented nor running stable. While the Teensy 3.5 application provides the same functionalities as the previous software, all its issues are documented and possible workarounds are suggested. Data handling and data loss in case of unreliable data transmission channel is handled better and the application is running stable.

this is a copy-paste of VA1!! rewrite!

Table of Contents

1 Introduction

this is a copy paste of VA1!!! rewrite!

This work is being done for Aeroscout GmbH, a company that specialized in development of drones. With unmanned vehicles, there are always on-board and off-board components. Data transmission between those components is of vital importance. Depending on the distance between on-board and off-board components, different data transmission technologies have to be used.
So far, each device that generates or processes data was directly connected to a modem. This is fine while the distances between on-board and off-board components does not vary significantly. But as soon as reliable data transmission is required both in near field and far field, the opportunity of switching between different transmission technologies is vital. When data transmission with one modem becomes unreliable, an other transmission technology should be used to uphold exchange of essential information such as exact location of the drone.
The goal of this project is to provide a flexible hardware that acts as a switch between devices and modems. Data routing between all connected devices and modems should be configurable and data priority should be taken into account when transmission becomes unreliable.
It should be possible to transmit the same data over multiple modems to reduce the chance of data loss for vital information. At receiving side, this case should be handled so the original information can be reassembled correctly with the duplicated data received. In case of data loss or corrupted data, a resend attempt should be started.
The configuration should be read from a file on an SD card. This SD card should also be used to store logging data. The system should run with Free RTOS and have a command/shell interface. When no devices are connected, the Free RTOS should go into low power mode.
Data loss should be handled and encryption and interleaving should be implemented for data transmission.
A hardware should be designed that is ready for field, with a good choice of connectors, small and light weight.

It was not necessary to start from scratch for this project. Andreas Albisser has already developed a hardware with four RS-232 interfaces to connect different data generating and processing devices and four RS-232 interfaces to connect modems. As a micro controller he used the Teensy 3.2, a small, inexpensive and yet powerful USB development board that can be used with the Arduino IDE.
Andreas Albisser also developed a software for the designed UAV Serial Switch base board. The software concept implemented became more complex as the requirements were expanded during development. The finished product did not fulfill all requirements of Aeroscout GmbH. Therefore this follow up project was initiated with new requirements and the hope of a better and easier expandable software as an outcome.
Not all requirements can possibly be implemented within one semester, but good ground work should be provided for further modifications and expansions.
Because encryption requires a more powerful micro controller than has been used by Andreas Albisser, some hardware modifications are required in the scope of this project. The most profund change is the micro controller and usage of Free RTOS. This will therefore be the main focus inside the project. The aim is to have a stable application with at least as many features and working configuration parameters as the old software had.
Some requirements demand hardware changes on the base board so an evaluation needs to be done inside this project to decide how to proceed and where to invest time.
A detailed task description can be found in chapter two. An overview and critical analysis of the

hardware and software provided by Andreas Albisser is in chapter three. In chapter four, all hardware changes that have been done in the scope of this project are described, followed by chapter four with a description of the software developed. Chapter six is for the conclusion and lessons learned.

2 Requirements

This project has been done for the company Aeroscout GmbH. Aeroscout specialized in the development of drones for various needs.
With unmanned aerial vehicles, the communication between on-board and off-board devices is essential and a reliable connection for data transmission is necessary. While the drone is within sight of the control device, data can be transmitted over a wireless connection. With increasing distances, other means of transmission have to be selected such as GPRS or even satellite.
So far, the switching between different transmission technologies could not be handled automatically. The data stream was directly connected to a modem and transmitted to the corresponding receiver with no way to switch to an other transmission technology in case of data transmission failure. A visualization of this set up can be seen in Figure 2.1. In the previous project, a flexible platform was developed that acted as a Serial Switch with multiple input and output interfaces for connecting devices and transmitters. See a sample setup in Figure 2.2. The hardware has four UART interfaces for devices such as sensors and actors and four UART interfaces for modems. The software developed in the previous project provided basic functionalities such as routing data between devices and modems and retransmission in case of data loss due to interference or an unstable connection. The application was still in its first stage but mostly running stable and provided the basic functionalities correctly.
In the scope of this project, the software should first be refactored and all pending requirements from the previous project that are necessary to proceed with the requirements for this project should be implemented. The refactoring and implementation of pending features includes:

- Order and assemble at least two Teensy Adapter Boards with silk screen
- Overview of task priorities and interrupt priorities

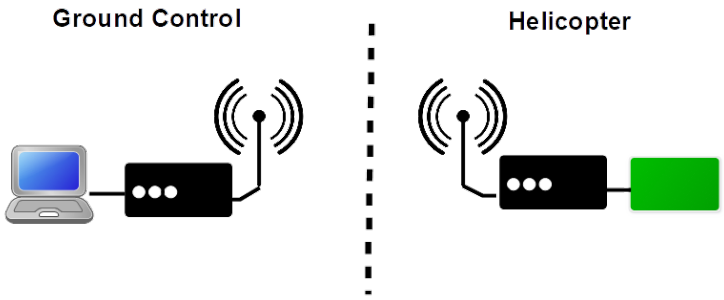

Figure 2.1: Previous system setup for data transmission

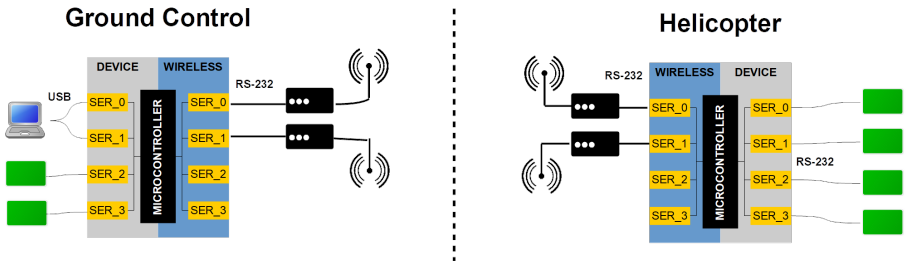

Figure 2.2: New system setup for data transmission

- Logging of exchanged data
- Improved debug output
- Analysis of runtime behavior of application
- Analysis of memory usage of application

Only when the above mentioned tasks are completed can the next and most important phase of the project be started.
The main goal of this project is to implement two more key features: Reliable data exchange and data security. Both items are elaborated in more detail below. They should not only be part of the software but the hardware used should also be analyzed for possible useful settings that enhance the data security and reliability behaviour.

2.1 Reliable Data Exchange

Because unmanned aerial vehicles constantly change their position, transmission is not always reliable. About 10% - 20% of the transmitted data are lost and the received data might be corrupted due to interference. The application developed in the scope of this project should take this into account and ensure a reliable data stream.
Reliability can be improved in various ways:

- Recovery of lost bytes by adding redundancy
- Retransmission of lost packages
- Improving the algorithm that selects the wireless connection to be used

Improvements on all of the above mentioned concepts should be made within this project.

2.2 Data Security

Data communication between the two Serial Switches should not be interceptable. This results in the following requirements for security during communication:

- Ensure that the CPU is working at maximum 15% capacity during periodic data exchange between on-board and off-board components so that there are enough resources left for encryption
- Chose a suitable encryption algorithm that requires little computational power and results in little additional data traffic
- Find a solution for encryption key generation and encryption key storage

Additionally, data transfer between the two serial switches should be logged, similar to the black box concept known from aviation. This results in the following requirements for logging security:

- A solution has to be found how tempering of the logged data can either be prevented or detected.
- Assure that logging does not use more than 10% CPU capacity

3 Approach

This project builds on the outcome of the project ÜAV Serial Switch". In this last project, the hardware was designed for the Serial Switch and a software was developed that provides the basic functionalities such as data routing between devices and modems, package transmission on wireless side and the concept of acknowledges and retransmission.
The requirements for this follow up project can be found in Kapitel 2.
Before improving the software developed in the previous project, the hardware components needed for further software development were ordered and assembled. In the scope of the UAV Serial Switch project, a Teensy adapter board was designed and ordered. But because of manufacturing errors, only one adapter board could be assembled successfully. In order to fully test the application in a real-life setup, two Teensy adapter boards are required. Therefore, more adapter boards were ordered, assembled and tested as a first step within this project.
Afterwards, the software developed in the scope of the UAV Serial Switch project was tested with the new hardware. Improvements were made where necessary and new features such as logging were added. Before starting with the implementation of data security and realiability, the runtime behavior of the application was analyzed and assessed. This took up more time than expected because of faulty measurements. It seemed that the CPU was already working at full capacity with no room for the extra traffic that the implementation of security and reliability would cause. Only when finding out that the measurements were faulty due to the extra traffic caused by the analyzation tool could the implementation of the project requirements proceed.
Reliability of data exchange was implemented first, starting with an error correcting code. Aeroscout then claimed that their focus lays on retransmission in case of package loss which lead to the testing phase of the error correcting code being cut short.
Improvements of the retransmission behavior were done next, followed by the implementation of encryption. Due to a lack of time, both retransmission behavior and encryption could not be implemented fully.

add project plan

4 Software Refactoring

In the scope of a previous project, the basic functionalites of the Serial Switch were implemented. The end product was working but never tested throughly. Not all features were implemented that are needed for this next project phase. So in order to continue, the application first needed some refactoring and clean up before starting with the implementation of data security and reliability of data exchange. This chapter focuses on the improvements made on the UAV Serial Switch software that was developed in the previous project. As mentioned in the requirements in Kapitel 2, an order for Teensy adapter boards needs to be placed and an overview of the task and interrupt priorities given. The refactoring shall at least include an added logging task and improved debug output.
The status of the software and hardware at the start of the project are described in more detail below, followed by the improvements made in the scope of this project.

4.1 Outcome of Previous Project

In previous projects, the basic functionalities of the Serial Switch were implemented. A hardware was designed and software was written for it. More details about the components provided can be taken from the sections below.

4.1.1 Hardware

The hardware consists of three main components: a baseboard, the main microcontroller board and an adapter board to fit the microcontroller used onto the baseboard.

Baseboard

The baseboard has eight serial UART interfaces, four of which are to connect devices such as sensors and actors and four of which are to connect modems for transmission. See Figure 2.2 for more details.

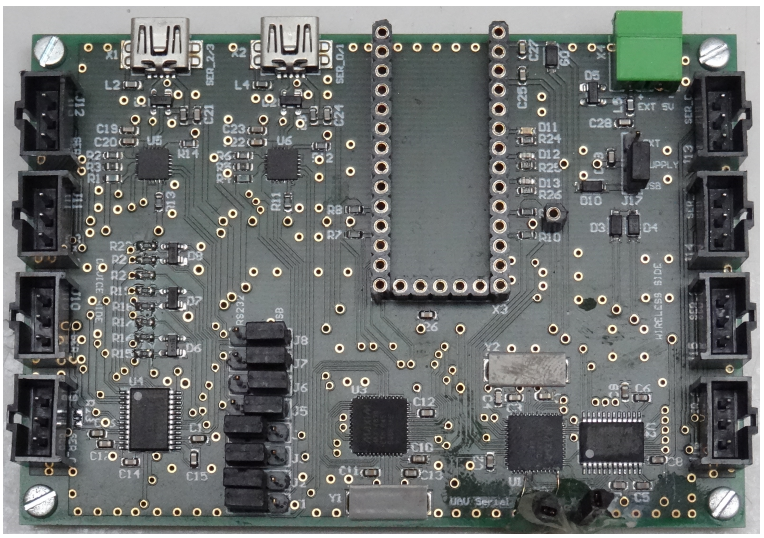

Figure 4.1: Baseboard

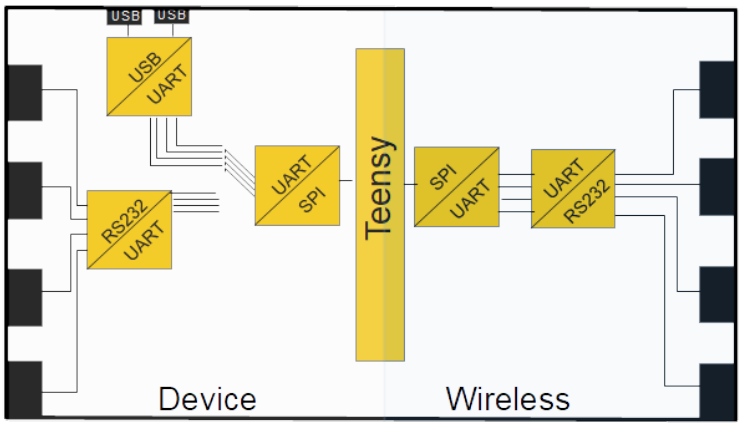

Figure 4.2: Block diagram of the components used on the baseboard

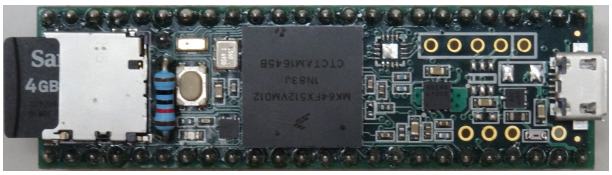

Figure 4.3: Teensy 3.5

For future references, the side where sensors and actors are connected will be referred to as the device side and the four interfaces for modems will be referred to as the wireless side. The bare baseboard can be seen in Figure 4.1. A block diagram about the components used on the baseboard can be seen in Figure 4.2.
The eight user interfaces available are all UART serial interfaces with configurable baud rates. They run on RS232 level, which is +-12V. On device side, there are jumpers available so the user can chose between RS232 input/output and USB input/output for each interface. When the USB is chosen, a serial COM port will appear per USB to UART converter and act as a device input/output.
The SPI to UART converter is needed as an interface between the serial interfaces accessible to the user and the microcontroller. It also acts as a hardware buffer that can store up to 128 Bytes of data. There are two hardware buffers on the baseboard, one for the four UART interfaces on device side and one for the four UART interfaces on wireless side. For a detailed description of all hardware components, please read the documentation of the UAV Serial Switch project.

Microcontroller

In a first version of the Serial Switch, the Teensy 3.2 development board was used as a main microcontroller unit but was soon replaced by the Teensy 3.5 development board which is still used today. The Teensy 3.5 can be seen in Figure 4.3. The Teensy 3.5 features a more powerful microcontroller, more memory and a hardware encryption unit which the Teensy 3.2 does not have. It is therefore more suitable for data encryption implemented in the scope of this project.

Adapter Board

Because the baseboard has originally been designed for the less powerful and slightly smaller Teensy 3.2, an adapter board was developed to map the pins of the Teensy 3.5 to the footprint of the Teensy 3.2. The adapter board can be seen in Figure 4.4.
In the scope of the UAV Serial Switch project, the adapter board has been ordered at the interal production at HSLU several times but all boards came back with multiple production errors (due to

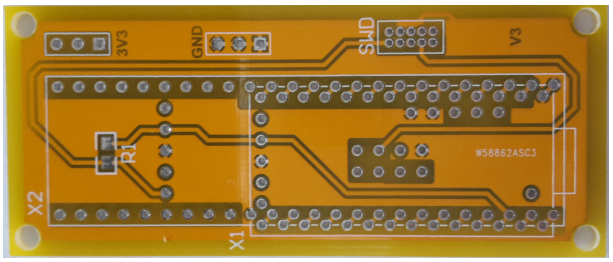

Figure 4.4: Adapter board from Teensy 3.2 to Teensy 3.5 footprint

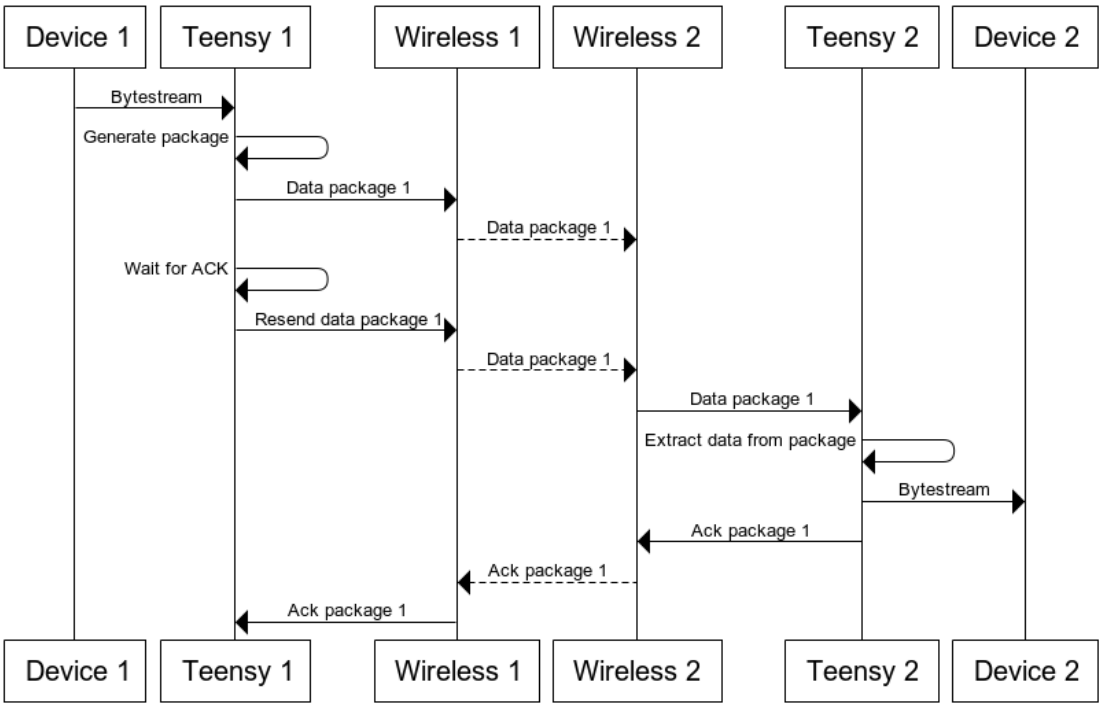

Figure 4.5: Package retransmission in case of no acknowledge received

no silk overlay). The previous project was finished with only one functional adapter board and more were ordered externally in the scope of this project. The new Teensy adapter boards now have a yellow silk overlay and were assembled and tested successfully.

4.1.2 Software

The Teensy 3.5 development board acts as the main microcontroller and runs with the operating system FreeRtos V9.0.1.
The main functionality of the software is data transmission on wireless side. For this, bytes read on device side are collected and put into packages for transmission. Received packages are checked for validity and their payload extracted and pushed out on device side.
Acknowledges can be configured to make the data transmission more reliable. If acknowledges are enabled, a package is resent in case no acknowledge is received within the specified timeout. A visualization of this can be seen in Figure 4.5.
The software can be configured by modifying the configuration ini file saved on the SD card located on the Teensy 3.5 development board.
The main functionality of the software is provided by three tasks, a simplified software concept can

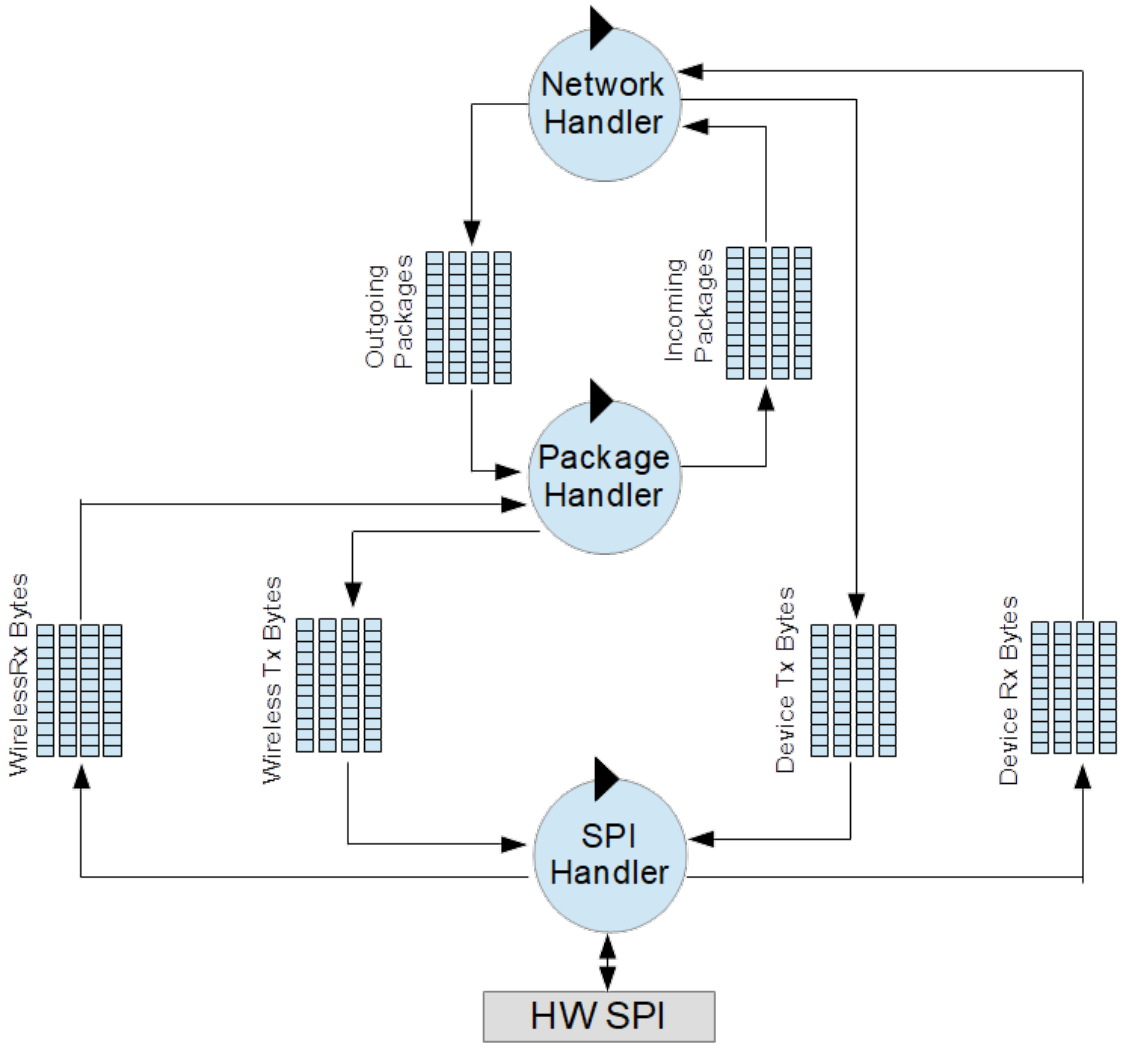

Figure 4.6: Simplified software concept at the beginning of this project, showing only the three main tasks

7	Application Layer
6	Presentation Layer
5	Session Layer
4	Transport Layer
3	Network Layer
2	Data Link Layer
1	Physical Layer

Figure 4.7: ISO OSI Model

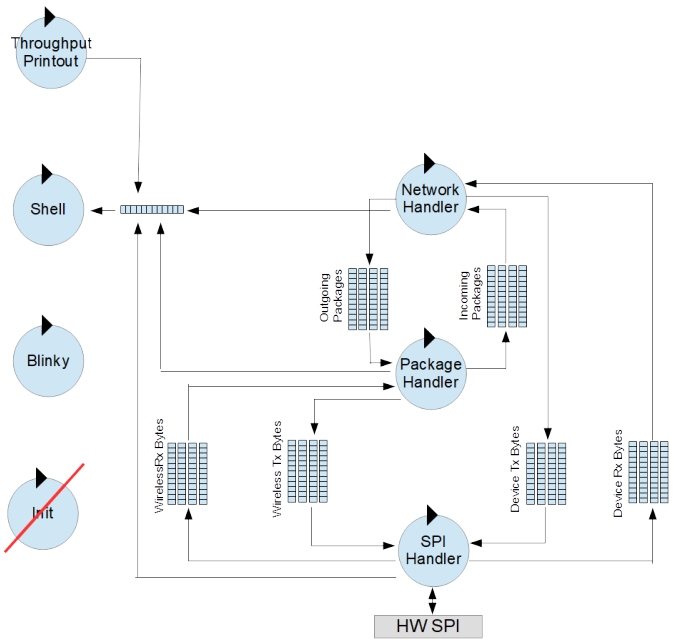

Figure 4.8: Full software concept at the beginning of this project, showing all tasks

be seen in Figure 4.6. Task intercommunication is done with queues. Data is pushed onto the queue by one task and popped from the queue by another task for processing. The ISO-OSI model is taken as a reference guide for the responsibilities of each task. The ISO-OSI Model is a representation of a communication standard, as seen in Figure 4.7. According to [18], the 7 layers have the following responsibilities:

1. Physical Layer: This layer is responsible for the relationship between the application and the physical transmission medium.
2. Data Link Layer: This layer is responsible for error detection and correction, determines the protocol to establish and does network layer protocol encapsulation
3. Network Layer: This layer handles data routing. Message delivery at this layer is not necessarily guaranteed to be reliable
4. Transport Layer: This layer controls the reliability of a given link, e.g. with acknowledges
5. Session Layer: This layer provides the mechanism for opening, closing and managing a session between end-user application processes
6. Presentation Layer: This layer is responsible for the delivery and formatting of information to the application layer for further processing or display
7. Application Layer: This layer standardizes communication and interfaces used in a network

The SPI Handler represents ISO-OSI Layer 1, the Package Handler represents ISO-OSI Layer 2 and the Network Handler represents layer 3 and all upper layers in the ISO-OSI model.
Additional tasks such as the ThroughputPrintout task, the Blinky task or the Shell task are for debug purposes only. There is also an Init task which reads the configuration file saved on the SD card, creates all other tasks and afterwards kills itself. A diagram with the full overview of all tasks and

their interfacing queues can be seen in Figure 4.8.
Details about the purpose of each task can be taken from below. For a full insight into each task, suggested improvements and implementation details, please consult the documentation of the UAV Serial Switch project.

SPI Handler

The SPI Handler represents ISO-OSI layer 1 and is the only task that accesses the serial interfaces on the baseboard. It reads data from the SPI to UART converters and pushes it onto the corresponding queue for the next task to process and it pops data from its interfacing queues to push out to the SPI to UART converters.
This task does byte handling only and knows nothing about data packages or any other data structures. Data routing is also not done within this task, e.g. bytes popped from the queue for serial interface 3 is also pushed out on serial interface 3.

Package Handler

The Package Handler pops bytes from the RxWirelessByte queues (interfacing queue to SPI Handler) and assembles them to full data packages. Package validity is checked here by looking at CRC and session number. The successfully assembled and valid packages are then pushed onto the Incoming-Packages queue for the next task to process, invalid packages are discarded.
The Package Handler also pops packages from the OutgoingPackages queue, adds CRC and session number and disassembles them into bytes to push onto the WirelessTxBytes queue for the SPI Handler to process.
Data routing is not done within this task, e.g. packages assembled with bytes from wireless connection 3 are pushed to the IncomingPackages queue for connection 3.

Network Handler

The Network Handler collects data bytes from the RxDeviceBytes queues, generates packages and pushes them to the OutgoingPackages queue for the Package Handler to disassemble and send out. This task keeps track of the sent packages and received acknowledges. If acknowledges are enabled, resending of packages is done within this task. The Network Handling also does the package routing and extracts the payload from received packages to push down to the TxDeviceBytes queue for the SPI Handler to send out.

Shell

The Shell task is responsible for the RTT interface. As long as there is an J-Link connection to the target, even with no ongoing debug session, either the RTT Viewer or RTT Client can be started on the connecting computer to access the information provided by the Shell task.
The Shell task reads and parses commands supported by the Processor Expert components used. For a list of supported commands, type "help"into the RTT terminal. Components that currently support commands from the Shell are:

- FreeRTOS
- FAT File System
- Green LED pin
- TimeDate component

The same terminal of the Shell task is also used by the Throughput Printout task to provide information about the performance of the application. The Throughput Printout task pushes debug information

onto the interfacing shell queue and upon every execution, the Shell task prints all strings found inside that queue.

Throughput Printout

This task provides information about the bytes read from and written to the hardware buffer. It also provides information about the packages sent and received on wireless side, bytes lost and general errors and warnings of by the application.
All debug information is printed out on the RTT interface and is therefore only available when the Shell task is enabled.

Blinky

This task periodically toggles the green LED on the baseboard.

Init

This task is the only task created upon startup of the application and runs as soon as the scheduler is started. It reads the configuration file on the SD card and fills the global config variable that is later accessed by all other tasks. Only when the configuration file is read and the content of the config variable verified does this task create all other tasks and afterwards kills itself.
It is not possible to read the SD card without the scheduler being started beforehand as the Init task accessed the FatFs file system which requires the scheduler to be running. Furthermore, because all other tasks access the global config variable which is filled within the Init task, the other tasks are created and started later by the Init task.

Idle

This task is running when no other task is. It is provided by the FreeRtos and no changes have been made to it. It is only listed here for completeness and for the reader to have a full overview of all tasks running.

Task Priorities

The application consists of the tasks mentioned above where each task runs with the following priority:

- SPI Handler: tskIDLE_PRIORITY+3
- Package Handler: tskIDLE_PRIORITY+2
- Network Handler: tskIDLE_PRIORITY+2
- Shell: tskIDLE_PRIORITY+1
- Throughput Printout: tskIDLE_PRIORITY+1
- Blinky: tskIDLE_PRIORITY+1
- Init: tskIDLE_PRIORITY+2

Logically high priority tasks have a high priority number and logically low priority tasks have a low priority number. The maximum priority is configurable in the FreeRtos Processor Expert component and is set to 6. The highest priority task can therefore run with priority tskIDLE_PRIORITY+5, because the lowest priority number is 0 which is tskIDLE_PRIORITY. Only the Idle task should be running with tskIDLE_PRIORITY as it is the fallback task for the scheduler when no other task is ready for execution.

Interrupt Priorities

The FreeRtos used allows for 16 interrupt priority levels, with 0 being the logically highest priority and 15 being the logically lowest priority. These priorities are have been implemented with four bits, resulting in the following possible interrupt values:

- Hexadecimal: 0x00, 0x10, 0x20, 0x30, 0x40, 0x50, 0x60, 0x70, 0x80, 0x90 0xA0, 0xB0 0xC0, 0xD0, 0xE0, 0xF0
- Decimal: 0, 16, 32, 48, 64, 80, 96, 112, 128, 144, 160, 176, 192, 208, 224, 240

These values map to priority level 0 to 15.
Negative interrupt priorities are defined by ARM and are part of the core, so they cannot be used. Interrupt priorities with values >= 0 are typically used for devices like UART, SPI etc.
The following components within this project use hardware interrupts:

- RNG: Random Number Generator, used to create session number for Package Handler task
- SPI: Serial interface, used by SPI Handler task to communicate with hardware buffers
- SPI2: Serial interface, used by FatFs which is again used by the Logging task to communicate with SD card
- RTC: Real Time Clock, used by the FatFS but no RTC hardware component or battery attached so the RTC is not consistend across reboots and power outs

The interrupt priorities set inside this application can be see in the file Vector_Config.h and have the following values:

- RNG: 112 or 0x70
- SPI: 48 or 0x30
- SPI2: 112 or 0x70
- RTC: 112 or 0x70

Known Issues and Missing Features

Detailed information about the issues with the provided application can be taken from the documentation of the previous project UAV Serial Switch. Major issues that need to be fixed or added are the following:

- Packages are not numbered consecutively so there is no way for the receiver to know if a package is missing.
- Package reordering is not implemented, the payload of each received package is pushed out immediately. If packages are not received in correct order, e.g. because a package was lost and then retransmitted but in the meanwhile a newer package arrived, the payload sent out on device side is in a wrong order and may therefore be useless for the connected device
- The CRC check of both header and payload is implemented but still commented out because of some issues. Currently, with CRC check enabled, most packages get discarded because of a faulty header.
- It is not possible to transmit redundant data over several modems because only packages are numbered and the payload is not. The receiver would not know when the received payload is redundant and could be discarded because package numbering is handled per wireless connection and the same package receives different package numbers when being sent out over multiple modems.
- Logging has not been implemented yet.

4.1.3 Conclusion

With two Teensy adapter boards successfully assembled and tested, work on the software can be started. In this section, all interrupt and task priorities have been analyzed and are configured correctly.

4.2 Added Features

Before starting with the implementation of data security and reliability of data exchange, the current software needs to undergo refactoring and some features need to be added. The resulting software concept looks as in Figure 4.9. Details about the added features can be found below.

4.2.1 Logging

All exchanged data has to be logged. This was in fact part of the requirements for the previous project but could not be implemented due to a lack of time. Therefore it was implemented in the scope of this project.
A task was added to the software that handles data logging only. The Package Handler assembles packages out of received bytes and pushes the successfully assembled and valid packages to a queue for the Network Handler to process. When logging is enabled, the Package Handler also creates a copy of the assembled package and pushes it to the IncomingPackages queue of the Logger.
The Network Handler puts bytes from the device byte stream into packages and pushes those onto the OutgoingPackages queue of the Package Handler for further processing. When logging is enabled, a copy of the package is created and pushed onto the OutgoingPackages queue of the Logger task.

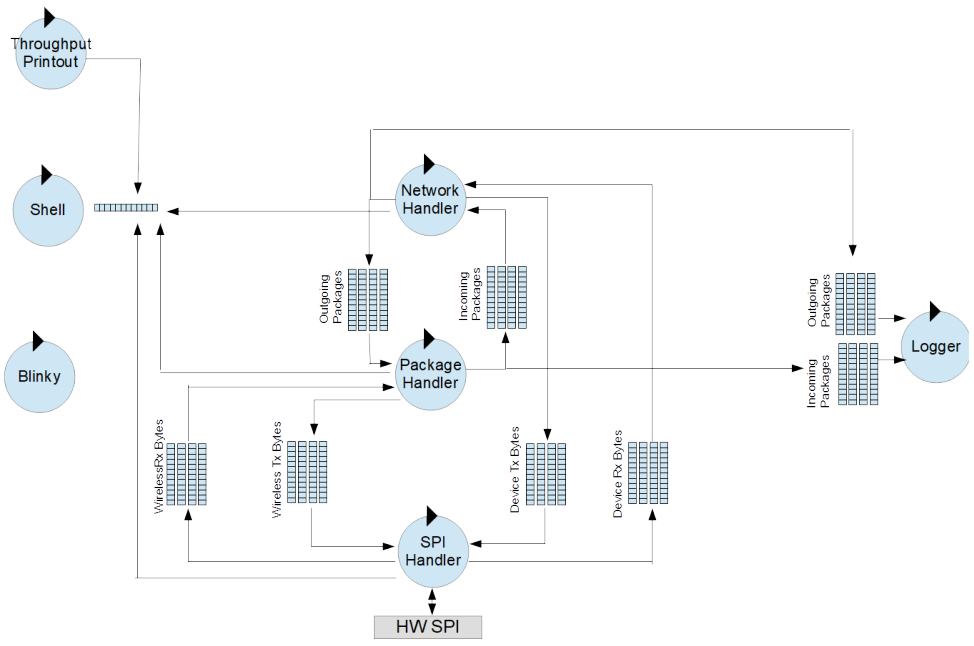

Figure 4.9: Software concept after added features

Logging can be enabled by setting the LOGGING_ENABLED parameter to 1 in the configuration file located on the SD card. There is also a LOGGER_TASK_INTERVAL parameter that determines the execution period of this task.
When logging is enabled, each package in the interfacing queues is converted to a comma separated values string and written into the correct log file. The log files are named accordingly, e.g. all packages in the OutgoingPackages queue for wireless connection 2 are logged into the file txPakWl2.log and all packages in the IncomingPackages queue for wireless connection 1 are logged into the file rxPakWl1.log.
Upon startup of the application, the Logger opens all existing log files, moves to the end of the files and adds the header. Periodically, the Logger task will pull packages from its interfacing queues, convert them into log strings and write them to the SD card. A sample log file could look as follows:

```
1  PackageType;DeviceNumber;SessionNumber;PackageNumber;PayloadNumber;PayloadSize;CRC8_Header
       ;Payload;CRC16_Payload
2
3  01;00;45;01;01;0011;88;FE092DFF0000000000000608C004039640;AD7C
4  01;00;45;02;02;0011;88;FE0930FF0000000000000608C0040371F4FE022EFF00150100;1245
5  01;00;45;03;03;0011;88;FE0931FF0000000000000608C004039B8A;63E4
6  01;00;45;04;04;0011;88;FE0932FF0000000000000608C00403A509;A3F6
7  01;00;45;05;05;0011;88;FE0933FF0000000000000608C004034F77FE2134FF004C0000;FD86
8  01;00;45;06;06;0011;88;FE0935FF0000000000000608C004032279;B5D3
9  01;00;45;07;07;0011;88;FE0936FF0000000000000608C004031CFA;75C1
10 01;00;45;08;08;0011;88;FE0937FF0000000000000608C00403F684;35CF
11 01;00;45;09;09;0011;88;FE0938FF0000803F0000000000000000000000000000000000000000;7822
12 01;00;45;0A;0A;0011;88;FE0939FF0000000000000608C00403F865;1985
13 01;00;45;0B;0B;0011;88;FE093AFF0000000000000608C0040C6E6;D997
```

The log file output can easily be imported into an Excel file, the values separated and visualized.
When the microcontroller writes data to the SD card, the write process does not necessarily take place right away. The main microcontroller on the Teensy has an internal buffer of 512 Bytes and only transfers the content of that buffer down to the SD card when this buffer is full. This write process can be forced by executing a FatFs flush command on the Teensy which results in the internal buffer content being written to the SD card immediately.
To ensure that log data is saved on the SD card periodically and not only when the internal buffer is full, the interval at which the flush command is executed can be specified with the parameter SD_CARD_SYNC_INTERVAL (in seconds) in the configuration file.
The task interval at which elements inside the interfacing queues (OutgoingPackages and IncomingPackages) are processed and converted to a log string can be specified with the parameter LOGGER_TASK_INTERVAL in the configuration file on the SD card.

Next Steps for Logging Task

The standard SD (Secure Digital) format was used inside this application for file access and storage on the memory card. According to [20], SD uses a 9-pin connector as an interface are there are three transfer modes supported:

- SPI mode
- One-bit SD mode with separate command and (bidirectional) data channel and a proprietary transfer format
- Four-bit SD mode (uses extra pins plus some reassigned pins) to support four bit wide parallel transfers. The commands are the same as with one-bit SD mode

All 9 pins of the memory card slot are routed to the main microcontroller on the Teensy 3.5. For communication, the SPI mode is used because there is a byte-oriented hardware SPI component available on the microcontroller. This is the fastest mode for the Teensy because with the one-bit or four-bit

SD mode, bits would have to be manually transferred one at a time in software (bit-banging) and bidirectionality of the data pins would have to handled in software as well.
Also, the SPI-bus interface mode is the only type that does not require a host license for accessing SD cards.
Instead of using the SD format, the application could use the SHDC format. SHDC is an extension of SD, with the same physical connections and dimensions but an increased storage capacity and a FAT32 filesystem instead of FAT12 or FAT16 as supported by SD. According to [5], SD supports data transfer speeds of 0.9 - 20MB/s while the SHDC supports data transfer speeds of 2 - 40MB/s.
Because there was no ready to use Processor Expert component available for SHDC, SD was used. In a later project, the SD component should be replaced with the faster SHDC component to reduce CPU time of the logging task.

4.2.2 CRC

The bug with the faulty CRC check has been found and fixed. Both header and payload CRC are now checked inside the Package Handler task.

4.2.3 Debug Output

The periodic debug text is calculated and assembled by the Throughput Printout task, pushed onto the Shell queue and printed out to the terminal upon the next execution of the Shell task. By adjusting the THROUGHPUT_PRINTOUT_TASK_INTERVAL (in seconds), the frequency of the debug output can be set.

```
1  ********************************************************************************************
2  Device 0 ------>   0 B/s -------> ================== ------->   0 B/s   -------> Wireless 0
3                                    ||             ||            0 Datapack/s
4  Device 1 ------>   0 B/s -------> ||             || ------->   0 B/s   -------> Wireless 1
5                                    ||             ||            0 Datapack/s
6  Device 2 ------>   0 B/s -------> ||             || ------->   0 B/s   -------> Wireless 2
7                                    ||             ||            0 Datapack/s
8  Device 3 ------>   0 B/s -------> ================== ------->   0 B/s   -------> Wireless 3
9                                                                 0 Datapack/s
10
11
12 Device 0 <------   0 B/s <------- ================== <-------   0 B/s   <------- Wireless 0
13                                   ||             ||            0 Datapack/s
14 Device 1 <------   0 B/s <------- ||             || <-------   0 B/s   <------- Wireless 1
15                                   ||             ||            0 Datapack/s
16 Device 2 <------   0 B/s <------- ||             || <-------   0 B/s   <------- Wireless 2
17                                   ||             ||            0 Datapack/s
18 Device 3 <------   0 B/s <------- ================== <-------   0 B/s   <------- Wireless 3
19                                                                0 Datapack/s
20
21 NetworkHandler: Total number of dropped packages per device input: 0,0,0,0
22 NetworkHandler: Total number of dropped acknowledges per wireless input: 0,0,0,0
23 PackageHandler: Total number of invalid packages per wireless input: 0,0,0,0
24 SPI Handler: Total number of dropped bytes per device byte input: 0,0,0,0
25 SpiHandler: Total number of dropped bytes per wireless byte input: 0,0,0,0
26
27 ********************************************************************************************
```

The ThroughputPrintout task does its calculations with float point arithmetics. If CPU occupancy becomes an issue, all variables should be changed to integers so the calculations become easier for the microcontroller.

4.2.4 Package Numbering / Payload Numbering

In the previous implementation, the packages were not numbered consecutively but instead the system tick time (which is the software running time) was used as a package number. There was no means for the receiver to know if a package was missing.
A continuous incrementing package number is now used that replaces the previous system time inside the package header. Each wireless connection now has a separate package counter.
The package header was also expanded by a payload counter because it was not possible to send the same package out on multiple wireless interfaces and discard all redundant packages on receiving side. The receiver would not know if a package was duplicated because of the separate package numbering per wireless connection. Therefore, a payload counter was introduced as well. Now the Network Handler task can not only handle resending of unacknowledged packages and do payload reordering (thanks to the package number), but it can also discard duplicated packages because each device byte stream now has its own continuous payload counter. The wireless package structure now looks as follows:

```
/*! \enum ePackType
*  \brief There are two types of packages: data packages and acknowledges.
*/
typedef enum ePackType
{
    PACK_TYPE_DATA_PACKAGE = 0x01,
    PACK_TYPE_REC_ACKNOWLEDGE = 0x02
} tPackType;


/*! \struct sWirelessPackage
*  \brief Structure that holds all the required information of a wireless package.
*  Acknowledge has the same packNr & devNum in header as the package it is acknowledging.
*  The individual packNr for ACK package is in its payload.
*/
typedef struct sWirelessPackage
{
    /* --- header of package --- */
    tPackType packType;
    uint8_t devNum;
    uint8_t sessionNr;
    uint16_t packNr;
    uint16_t payloadNr;
    uint16_t payloadSize;
    uint8_t crc8Header;
    /* --- payload of package --- */
    uint8_t* payload;
    uint16_t crc16payload;
} tWirelessPackage;
```

Next Steps for Package Numbering / Payload Numbering

Currently, the application allocates two bytes in the package header for package numbering and two bytes for payload numbering. It suffices to use one byte per parameter. This should be changed in the next software revision to keep the package header as small as possible.

4.2.5 Payload Reordering

Now that consecutive payload numbering has been introduced, there are different ways on how to handle packages received in wrong order:

– **Payload number ignored:** the receiver does not process the payload number, the payload is extracted from received packages and sent out in the same order as it was received. Redundant

payloads (a result of the same package being sent out over multiple wireless connections) will not be detected.

- **Payload reordering:** There is an internal array where received payload is stored for reordering. Together with the configuration parameter PACK_REORDERING_TIMEOUT, it determines the reordering behavior of the application. With PACK_REORDERING_TIMEOUT, the user can specify how long a missing payload is waited for before this package is skipped and the payload of the next internally stored package is sent out.
- **Only send out new payload:** There is no internal reordering of received payload but when packages get jumbled, the Network Handler task will discard packages with payload numbers older than the one previously sent out on device side.

All payload reordering is being done by the Network Handler task.

Next Steps for Payload Reordering

If the application is configured to ignore the payload number of send payload out only if it is new, then it is running correctly. The ground work has been done for payload reordering but it cannot be used yet because it does not work correctly. For implementation details and status of payload reordering, see Section 7.2. In a next project phase, more thought should be put into this issue.

4.2.6 Package Transmission Mode

When acknowledges are configured on a wireless connection, there are two ways on how package transmission can be handled:

- **Synchronous transmission:** The next package is only sent out when the previous package has been acknowledged
- **Asynchronous transmission:** Packages are sent out continuously without waiting on the acknowledge of the previous package

Synchronous transmission can be enabled for each wireless interface with the parameter SYNC_MESSAGING_MODE
If no acknowledge is configured for a wireless connection, this parameter is ignored on that interface.

Next Steps for Package Transmission Mode

There is no timeout implemented on this parameter. If a package is never acknowledged, that wireless channel will infinitely be blocked. A timeout should be implemented and the behavior tested with both packages that are acknowledged only after being resent and packages that are never received correctly and therefore never acknowledged.

4.2.7 Static Memory Allocation

Previously, memory for all tasks and queues was allocated dynamically which resulted in .
Static memory allocation can be enabled inside the FreeRtos Processor Expert component. Now only the Init tasks still uses dynamically allocated memory because it later deletes itself. All other tasks including their interfacing queues use static memory allocation.

why????

4.2.8 Conclusion

Adding missing features is a very time consuming task. There is still lots of work to be done and most new features were implemented in a very basic manner. The functionality of the overall software has improved but should not be used in a end product as it is. Time should be invested in refactoring the application first.
The refactored application is now ready to support payload reordering in case packages are not received in correct order. Both consecutive package and payload numbering have been introduced and a logger task has been added. The analysis of the runtime behavior of the logger (as requested in Kapitel 2) will be subject of Section 5.3.

5 System Analysis

Before expanding the application and implementing more features, the system performance needs to be analyzed to ensure sufficient capacity for error correcting codes and encryption. As mentioned in the requirements in Kapitel 2, the runtime behavior and memory usage of the application need to be analyzed.
In the previous project, SEGGER SystemView was used to analyze the runtime behavior and CPU load of the application.
In the scope of this project, Percepio Tracealyzer was used instead of SEGGER SystemView because it provides a more in-depth insight into the runtime behavior of the software. The Tracealyzer not only shows the task execution times and RTOS events, it also shows this information in interconnected views and collects data about the CPU load and memory usage.
Details about the application analysis carried out and their results are provided in the sections below.

5.1 Setup

The system analysis was carried out while the application was running and an autopilot was connected and communicating to the base station. The setup was as can be seen in Figure 5.1. The Pixhawk PX4 was used as an on-board autopilot. It can be controlled by QGroundControl running on a laptop off-board. Both Pixhawk and QGroundControl are open-source applications.
Upon startup, QGroundControl will try to establish a link to the Pixhawk. After successful link establishment, data and a heartbeat are exchanged periodically.
For runtime analysis, there is both a Processor Expert component for the SEGGER SystemView and one for Percepio Tracelyzer. These components are configured and enabled so that the corresponding code is generated. To use either one of these components, they have to be enabled in the FreeRTOS Processor Expert component. Only then will the debugger provide runtime information to the correct tool.

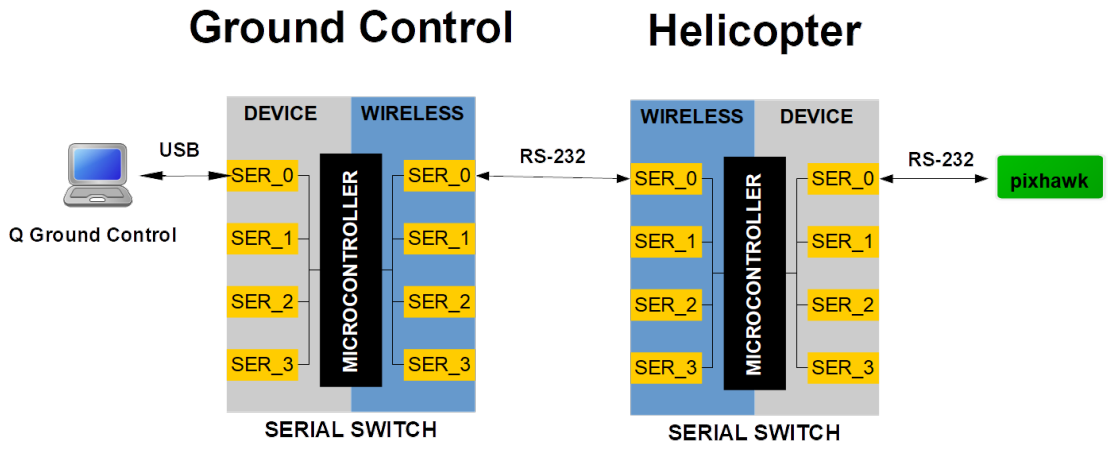

Figure 5.1: Test setup with autopilot and ground station

5.2 SEGGER SystemView

According to [21], SEGGER SystemView is a real-time recording and visualization tool for embedded systems that reveals information about runtime behavior of an application. SystemView can track down inefficiencies and show unintended interactions and resource conflicts.
The application developed in the scope of this project runs with three main tasks (SPI Handler, Package Handler and Network Handler, see Figure 4.9) that perform the main functionality of the software. Task intercommunication is done with queues, where one task always pushes data to a queue and another task pops this data from the queue to process it. This results in lots of queue operations each second when the UAV Serial Switch is busy.
Queue operations are part of the FreeRtos. Because SystemView logs all FreeRtos calls, the additional traffic caused by SystemView when the software is already working to capacity can cause the application to crash. This can be prevented by disabling the logging of queue operations for SystemView. Simply comment the following code lines out in the file SEGGER_SYSTEM_VIEW_FreeRTOS.h (can be found in the GeneratedCode folder):

```
1  //#define traceQUEUE_PEEK( pxQueue )
       SYSVIEW_RecordU32x4(apiID_OFFSET + apiID_XQUEUEGENERICRECEIVE,
       SEGGER_SYSVIEW_ShrinkId((U32)pxQueue), SEGGER_SYSVIEW_ShrinkId((U32)pvBuffer),
       xTicksToWait, xJustPeeking)
2  //#define traceQUEUE_PEEK_FROM_ISR( pxQueue )
       SEGGER_SYSVIEW_RecordU32x2(apiID_OFFSET + apiID_XQUEUEPEEKFROMISR,
       SEGGER_SYSVIEW_ShrinkId((U32)pxQueue), SEGGER_SYSVIEW_ShrinkId((U32)pvBuffer))
3  //#define traceQUEUE_PEEK_FROM_ISR_FAILED( pxQueue )
       SEGGER_SYSVIEW_RecordU32x2(apiID_OFFSET + apiID_XQUEUEPEEKFROMISR,
       SEGGER_SYSVIEW_ShrinkId((U32)pxQueue), SEGGER_SYSVIEW_ShrinkId((U32)pvBuffer))
4  //#define traceQUEUE_RECEIVE( pxQueue )
       SYSVIEW_RecordU32x4(apiID_OFFSET + apiID_XQUEUEGENERICRECEIVE,
       SEGGER_SYSVIEW_ShrinkId((U32)pxQueue), SEGGER_SYSVIEW_ShrinkId((U32)pvBuffer),
       xTicksToWait, xJustPeeking)
5  //#define traceQUEUE_RECEIVE_FAILED( pxQueue )
       SYSVIEW_RecordU32x4(apiID_OFFSET + apiID_XQUEUEGENERICRECEIVE,
       SEGGER_SYSVIEW_ShrinkId((U32)pxQueue), SEGGER_SYSVIEW_ShrinkId((U32)pvBuffer),
       xTicksToWait, xJustPeeking)
6  //#define traceQUEUE_RECEIVE_FROM_ISR( pxQueue )
       SEGGER_SYSVIEW_RecordU32x3(apiID_OFFSET + apiID_XQUEUERECEIVEFROMISR,
       SEGGER_SYSVIEW_ShrinkId((U32)pxQueue), SEGGER_SYSVIEW_ShrinkId((U32)pvBuffer), (U32)
       pxHigherPriorityTaskWoken)
7  //#define traceQUEUE_RECEIVE_FROM_ISR_FAILED( pxQueue )
       SEGGER_SYSVIEW_RecordU32x3(apiID_OFFSET + apiID_XQUEUERECEIVEFROMISR,
       SEGGER_SYSVIEW_ShrinkId((U32)pxQueue), SEGGER_SYSVIEW_ShrinkId((U32)pvBuffer), (U32)
       pxHigherPriorityTaskWoken)
8  #define traceQUEUE_REGISTRY_ADD( xQueue, pcQueueName )
       SEGGER_SYSVIEW_RecordU32x2(apiID_OFFSET + apiID_VQUEUEADDTOREGISTRY,
       SEGGER_SYSVIEW_ShrinkId((U32)xQueue), (U32)pcQueueName)
9  #if ( configUSE_QUEUE_SETS != 1 )
10 // #define traceQUEUE_SEND( pxQueue )
       SYSVIEW_RecordU32x4(apiID_OFFSET + apiID_XQUEUEGENERICSEND, SEGGER_SYSVIEW_ShrinkId((
       U32)pxQueue), (U32)pvItemToQueue, xTicksToWait, xCopyPosition)
11 #else
12 #define traceQUEUE_SEND( pxQueue )                                    SYSVIEW_RecordU32x4(
       apiID_OFFSET + apiID_XQUEUEGENERICSEND, SEGGER_SYSVIEW_ShrinkId((U32)pxQueue), 0, 0,
       xCopyPosition)
13 #endif
14 #define traceQUEUE_SEND_FAILED( pxQueue )                             SYSVIEW_RecordU32x4(
       apiID_OFFSET + apiID_XQUEUEGENERICSEND, SEGGER_SYSVIEW_ShrinkId((U32)pxQueue), (U32)
       pvItemToQueue, xTicksToWait, xCopyPosition)
15 //#define traceQUEUE_SEND_FROM_ISR( pxQueue )
       SEGGER_SYSVIEW_RecordU32x2(apiID_OFFSET + apiID_XQUEUEGENERICSENDFROMISR,
       SEGGER_SYSVIEW_ShrinkId((U32)pxQueue), (U32)pxHigherPriorityTaskWoken)
16 #define traceQUEUE_SEND_FROM_ISR_FAILED( pxQueue )
       SEGGER_SYSVIEW_RecordU32x2(apiID_OFFSET + apiID_XQUEUEGENERICSENDFROMISR,
       SEGGER_SYSVIEW_ShrinkId((U32)
```

The output of the SystemView will then not contain any information about queue events but only visualize task execution times and other FreeRtos calls.

5.3 Percepio Trace Analyzer

Just like the SEGGER SystemView, the Percepio Trace Analyzer logs all FreeRtos function calls, including queue operations. The Percepio Trace component can log data in two ways:

- **Streaming mode:** All log data is transferred from the microcontroller to the Tracelyzer desktop application in real-time
- **Snapshot mode:** There is an on-board buffer that is filled with log data. This buffer can be imported into the Tracelyzer to analyze the performance of the application. No live streaming is possible.

When starting with the application analysis with Percepio Trace, it seemed that the software had some major issues. The Tracelyzer showed that queue operations sometimes take up to 5 milliseconds for apparently no reason. The executed code was the following:

```
1  for (unsigned int cnt = 0; cnt < nofReadBytesToProcess; cnt++)
2  {
3      if (xQueueSendToBack(queue, &buffer[cnt], ( TickType_t ) pdMS_TO_TICKS(
           SPI_HANDLER_QUEUE_DELAY) ) != pdTRUE)
4      {
5          char infoBuf[100];
6          XF1_xsprintf(infoBuf, "Warning: Not all read bytes could be sent to queue, losing
               %u bytes on wl %u\r\n", (nofReadBytesToProcess-cnt), uartNr);
7          pushMsgToShellQueue(infoBuf);
8          break;
9      }
10 }
```

As can be seen, queue operations are done inside a while loop and the should be no reason for the application to stall inside this loop when the parameter SPI_HANDLER_QUEUE_DELAY is set to 0. This dleay parameter specifies how long the application should wait for a queue operation to finish successfully in case of a full or empty queue.
When looking at the timestamps in the output of the Percepio Trace Analyzer in Figure 5.2, the highlighted queue operation takes 4 milliseconds while all other queue operations take about 15-30 micro seconds in this example. This suggests that the application is stalling inside the queue operation. Furthermore, look at the system tick event. During normal operation of the application, the FreeRtos is configured to generate a system tick event every millisecond. When the application stalls inside the queue operation, the Percepio Trace output suggests that there is no system tick event generated. To verify this, code is added inside the system tick event handler function. An output pin is toggled so that periodic system ticks and correct FreeRtos execution can be checked. The code for the system tick event handler then looks as follows:

```
1  void FRTOS_vApplicationTickHook(void)
2  {
3      /* Called for every RTOS tick. */
4      TMOUT1_AddTick();
5      TmDt1_AddTick();
6      Pin33_NegVal(); /* toggle Pin 33  on Teensy */
7  }
```

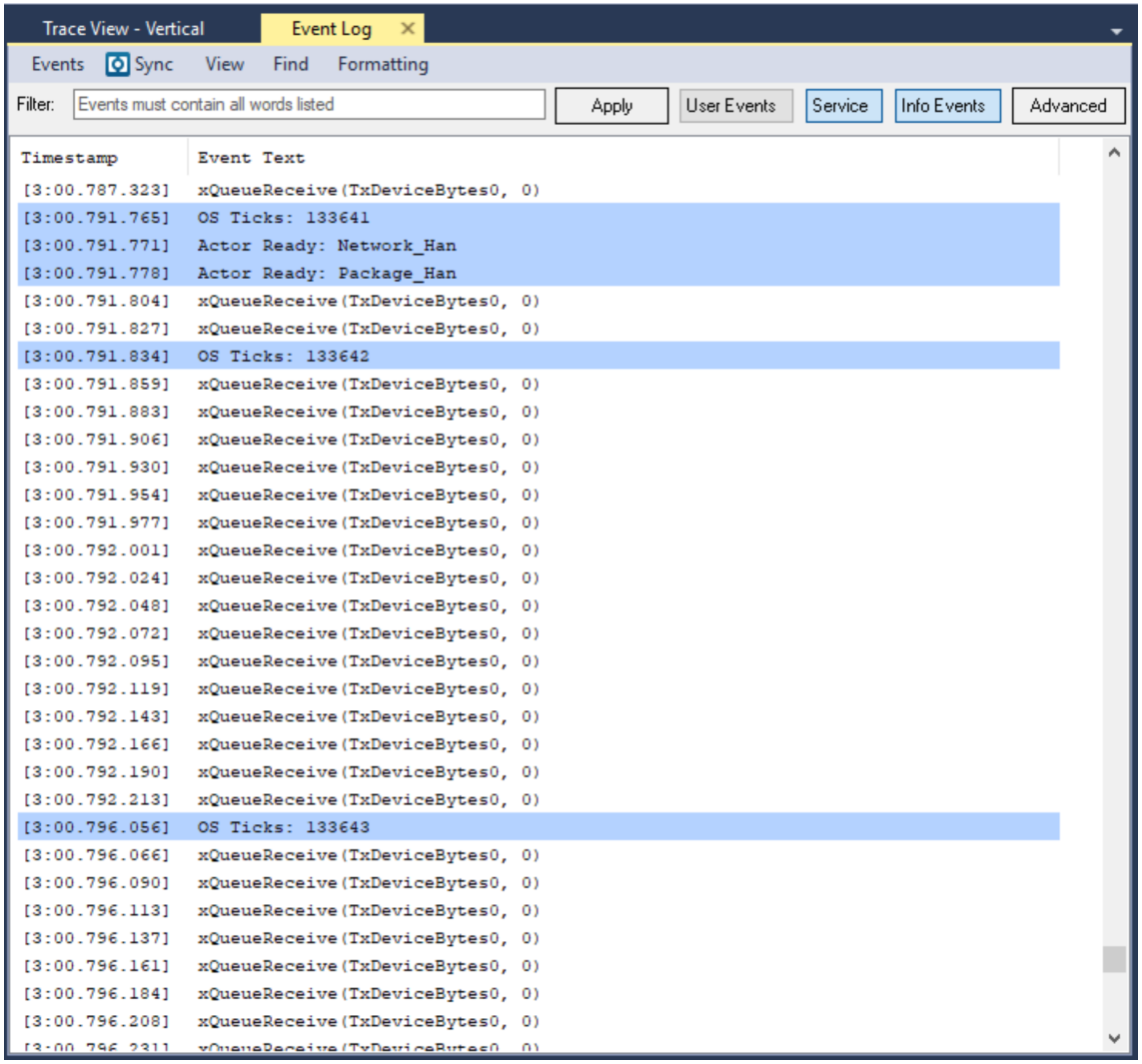

Figure 5.2: A queue operation taking 4 milliseconds

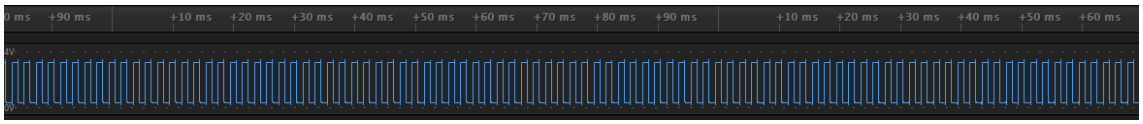

Figure 5.3: Periodic system ticks when Percepio disabled

Now periodic system tick events can be observed by monitoring the toggled pin (Pin33). When little to no data is exchanged between the Serial Switches, the system tick event is called periodically every millisecond and the output then looks similar to Figure 5.3. When lots of data is exchanged between two Serial Switches, the system tick event is indeed not called regularly as can be seen when measuring the output of Pin33 (see Figure 5.4). This suggests that there really is a problem that needs to be solved.
The issue can possibly have two sources: it is either a bug in the implementation of the queue operation within the FreeRtos operating system or it is a bug caused by Percepio Trace.

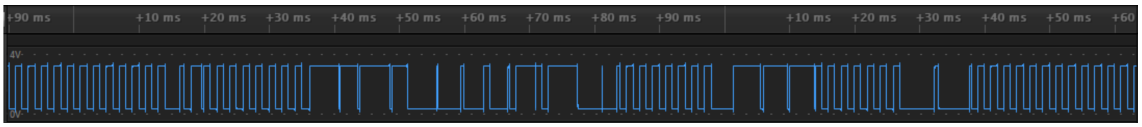

Figure 5.4: Irregular system ticks when Percepio enabled

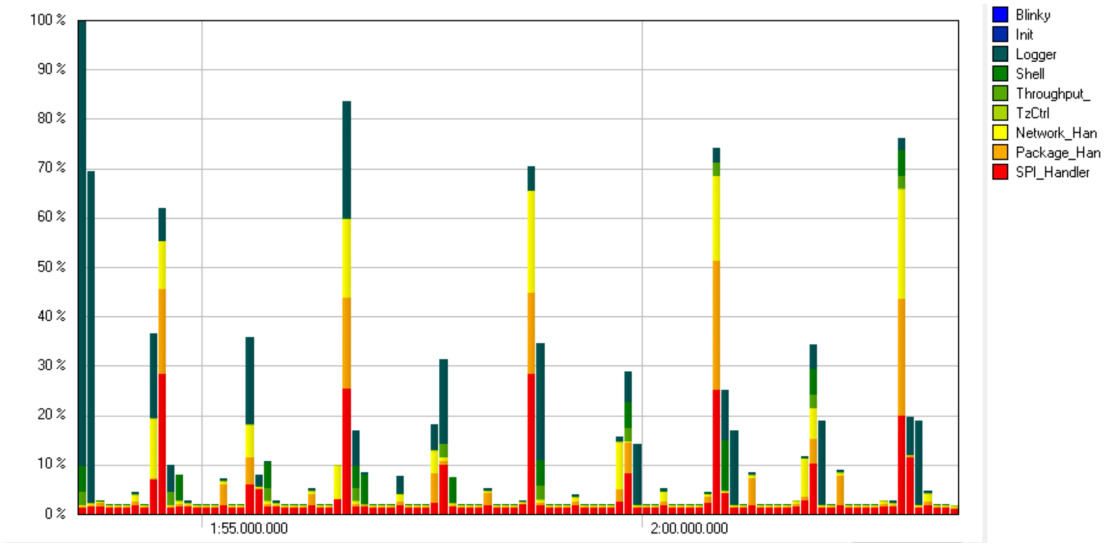

Figure 5.5: System load during periodic heartbeat exchange

5.3.1 Option 1: FreeRtos Issue

To rule out a bug in the FreeRtos queue implementation, the changelog was looked at and there were no known issues with the FreeRtos version used (V9.0.1) concerning queue operations. The FreeRtos component was updated anyway and the application now runs with FreeRtos V10.0.1. The same test was repeated and the application still randomly stalls inside queue operations and the Pin33 output still looks similar to Figure 5.4.
Therefore the source of the issue must be located with Percepio Trace.

5.3.2 Option 2: Percepio Trace Issue

To check if the bug is caused by Percepio Trace, this component was disabled. The results was a working system and with a regular system tick event, verified by a periodically toggled Pin33. This outcome suggests that the extra traffic caused by Percepio Trace results in a faulty behavior of the application.
It turned out that the Percepio Trace component can neither be used in snapshot not streaming mode, both options result in a faulty application.
Because Percepio Trace logs all FreeRtos calls and there are many queue operations when the Serial Switches are busy communicating, the extra traffic is most likely the source of the issue. Erich Styger then updated the Percepio Trace Processor Expert component to have an option for disabling the logging of queue events. Whether disabling the logging of queue events can prevent irregular system ticks is to be evaluated.
The conclusion is that Percepio Trace can provide a good indicator for general system performance, task execution times and task intercommunication but is not ideal when lots of FreeRtos functions are used due to the additional traffic it generates.
Nevertheless, various inefficiencies were found with thanks to Percepio Trace and it provides good

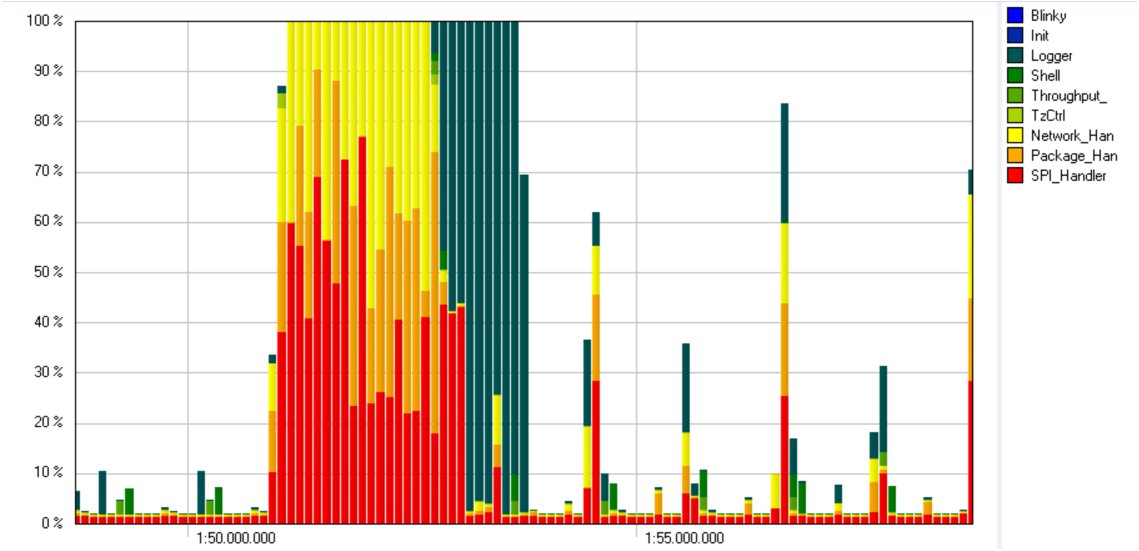

Figure 5.6: System load during link establisment

help with finding the cause of Hard Faults because the general area where the application stops can easily be made visible.

5.3.3 System Analysis with Percepio Trace

The Percepio Trace output may not be 100% accurate but it provides a good general idea about the system performance. The application has been tested with the hardware setup as in Figure 5.1.
Upon link establishment, about 5000 bytes of data are sent down from Pixhawk to QGroundControl and about 500 Bytes are sent from QGroundControl to Pixhawk. When the link is established and only periodic data is exchanged, the Pixhawk sends about 250 Bytes per second to the QGroundControl and QGroundControl sends about 50 Bytes per second to the Pixhawk.
The system load is therefore heaviest upon link establishment and with periodic load peaks during heartbeat exchange.
The Trazelyzer output during periodic heartbeat exchange looks as in Figure 5.5. During link establishment the output looks as in Figure 5.6.
The system load with all tasks enabled is visible in those figures. During periodic heartbeat exchange, the CPU occupancy is about 10% which means that there are enough resources left to implement encryption. During link establishment, the CPU load is 100% for about one second which results in only the high priority task being executed by the scheduler. Because the three main tasks (SPI Handler, Package Handler, Network Hander) are running with higher priorities than the other tasks, they are the only ones running during high capacity of the application. See Unterunterabschnitt 4.1.2 for details about task priorities.
The logger task takes up a lot of CPU time when lots of packages are exchanged. As mentioned in Section 4.2.1, the main microcontroller on the Teensy has an internal buffer of 512 Bytes and information is only written to the SD card if either this buffer is full or upon a FatFs sync command. Because lots of data is exchanged during high occupancy of the CPU, all the packages need to be logged and the internal buffer fills up fast which results in frequent forced write processes to the SD card. But because the Logger task is running with lower frequency than the three main tasks, it will not disturb the data exchange.
Dynamic memory usage has also been analyzed with Percepio Trace and no memory leak was found,

memory usage stayed consistent across link establishment and after two hours runtime with periodic heartbeat exchanges.

5.4 Conclusion

Several weeks have been spent with finding the cause of a software problem that appeared when doing a system analysis with Percepio Trace. It turned out that the reason for the faulty behavior was that the Tracelyzer caused a high amount of traffic when the application was already running at its capacity. Only after finding the source of this error could the system analysis be carried out.
The system analysis showed that there is enough CPU capacity left for encryption and an error correcting code as the average occupancy during periodic heartbeat exchange is about 10%.
Also, no memory leak was found in the application.
Now all requirements concerning refactoring and improvements of the application are implemented (see Kapitel 2).

6 Wireless Modems

The goal of this project is to create a flexible platform for data routing between devices and modems. But because not all modems behave the same and have equal configuration possibilities, further research needed to be done on the two modems that Aeroscout GmbH plans on using:

- RF686 RFD900x
- ARF868URL

This is also part of the requirements as seen in Kapitel 2.
This chapter provides an overview of the configuration possibilities and transmission behavior of both modems.

6.1 RFD900x

According to he datasheet [2], the RFD900x is a long distance radio modem to be integrated into custom projects.
Its features include:

- 3.3V UART interface
- The RTS and CTS pins are available to the user
- 5V power supply, also via USB
- Default serial port settings: 57600 baud, no parity, 8 data bits, 1 stop bit
- MAVLink radio status reporting available (RSSI, remote RSSI, local noise, remote noise)
- MAVLink protocol framing can be turned on and off.
- The RFD900x has two antenna ports and a firmware which supports diversity operation of antennas. During the receive sequence the modem will check both antennas and select the antenna with the best receive signal.
- There are three different communication architectures and node topologies selectable: Peer-to-peer, multipoint network and asynchronous non-hopping mesh.
- The RFD900x Radio Modem is configurable with methods like the AT Commands and APM Planner.
- Golay error correcting code can be enabled (doubles the over-the-air data usage)
- Encryption level either off or 128bit
- Adapted version of open source firmware SiK used for the modem

6.1.1 RTS and CTS Pins

The modem supports hardware flow control. Ready To Send (RTS) and Clear To Send (CTS) signals are part of the UART communication standard. The transmitter lets the receiver know that it is now ready to transmit data with the RTS line and the receiver lets the transmitter know when it is ready to receive data with the CTS line.

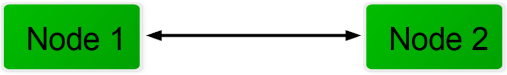

Figure 6.1: Peer To Peer Network

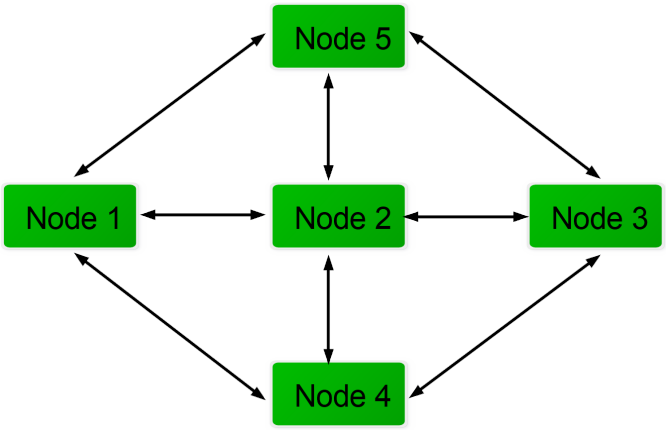

Figure 6.2: Asynchronous Non-Hopping Mesh

6.1.2 MAVLink Protocol

MAVLink or Micro Air Vehicle Link is a protocol for communicating with small unmanned vehicle. It is designed as a header-only message marshaling library. It is used mostly for communication between a Ground Control Station (GCS) and unmanned vehicles, and in the inter-communication of the subsystem of the vehicle. It can be used to transmit the orientation of the vehicle, its GPS location, speed and many more measurements.

6.1.3 Topology Options

The modem supports different network setups.

Peer To Peer Network

Peer to peer network is a straight forward connection between any two nodes. Whenever two nodes have compatible parameters and are within range, communication will succeed after they synchronize. If your setup requires more than one pair of radios within the same physical pace, you are required to set different network ID's to each pair. See Figure 6.1.

Asynchronous Non-Hopping Mesh

It is a straight foreward connection between two and more nodes. It allows data transfer across great distances if their settings match. See Figure 6.2.

Multipoint Network

In a multipoint connection, the link is between a sender and multiple receivers as long as their configuration matches and they are within range. See Figure 6.3.

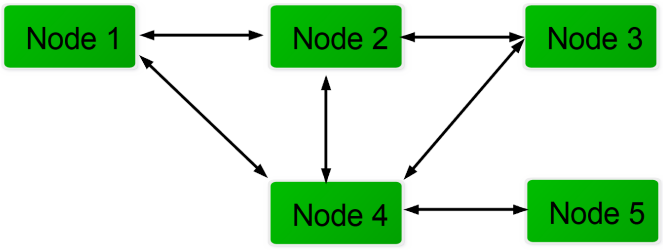

Figure 6.3: Multipoint Network

6.1.4 Configuration Methods

There are multiple options available on how the RFD900x can be configured.

AT Commands

The AT command set, formerly known as the Hayes command set, is a command language designed by Dennis Hayes for communication with a 300 baud modem in 1981. The original command set has since been expanded to meet various new needs.
In the RFD900x modem, the AT commands can be used to change parameters such as power levels, air data rates, serial speeds etc.
The AT command mode can be entered by using the '+++' sequence in a serial terminal connected to the radio. When doing this, the user must allow at least 1 second after any data is sent out to be able to enter the command mode. This prevents the modem to misinterpret the sequence as data to be sent out.
The modem will reply with 'OK' as a feedback to the user. Then commands can be entered to set or get modem and transmission parameters.

APM Planner

APM Planner is an open-source ground station application for MAVlink based autopilots including APM and PX4/Pixhawk that can be run on Windows, Mac OSX, and Linux. It cannot only be used as a mission planner and control application for the autopilot, but it also supports configuration of the connected modems.
The APM Planner is an alternative to the QGroundControl used by Aeroscout GmbH.

6.1.5 Error Correcting Code

The modem can encode the data stream in such a way so that up to 3 bit error in any 24 bit word of encoded data can be recovered and up to 7 bit error can be detected. The applied encoding algorithm is the binary Golay G24.

6.1.6 Encryption

The 128bit AES data encryption may be set by AT command or any other supported configuration tool. The encryption key can be any 32 character hexadecimal string and less and must be set to the same value on both receiving and sending modem.

6.1.7 SiK Firmware

The RFD900x runs with an improved and adapted version of the SiK firmware.
The SiK Telemetry Radio is a light and inexpensive open source radio hardware platform that uses open source firmware which has been specially designed to work well with MAVLink packets. It is not only for copters, but also for rovers and planes and is well integrated with the Mission Planner.
A SiK Telemetry Radio is one of the easiest ways to setup a telemetry connection between an autopilot (such as Pixhawk or APM) and a ground station (such as QGroundControl).
If the ground station supports it, the SiK radios together with the MAVLink protocol can be used to monitor the link quality while flying. This means that the modem can decode the MAVLink protocol and attach radio status parameters like RSSI and noise level to it for the ground station to display to the user.

6.2 ARF868URL

The ARF868 radio modem is a long-distance radio modem with the following features:

- +-12V RS232 interface
- RTS and CTS pins available to the user
- 12V power supply
- Default serial port settings: 9600 baud, no parity, 8 data bits
- MAVLink not decoded
- Radio status reports can be retrieved by AT command but not added to MAVLink protocol automatically
- One antenna port
- There are two different communication architectures and node topologies selectable: Peer-to-peer and multipoint networks
- The ARF868 Radio Modem is configurable with the AT command set over serial link and a dedicated configuration software called Ädeunis RF - Stand Alone Configuration Manager"which can be downloaded on the Adeunis website
- Packet mode can be enabled
- Encryption and Golay not supported

Most features are similar to the RFD900x and explained above. The main difference to this modem is that the ARF868 does not support encryption, Golay error correction and MAVLink decoding. This modem has not specifically been optimized for MAVLink communication and usage within unmanned aeral vehicles.

6.2.1 Packet Mode

ARF868 modem uses a packet oriented protocol on its RF interface. The data coming from the UART interface are accumulated in an internal fifo in the module and then encapsulated in an RF frame. The maximum amount of data that can be transferred in a single radio packet is 1024 Bytes.
The maximum packet size can be set up in S218 register from 1 to 1024 bytes. Each new packet introduces some latency in the transmission delay caused by the RF protocol overhead. The RF protocols encapsulate the data payload with the following elements:

- A preamble pattern required for receiver startup time
- A bit synchronization pattern to synchronize the receiver on the RF frame
- Other protocol field such as source address and destination address, payload length, optional CRC and internal packet type field.

The incoming fifo may accumulate up to 1024 data byte. No more data has to be set in the fifo while a 1024 bytes block of data has not been released by the radio transmission layer.
The user can configure the modem to run in non-secure packet mode where no acknowledges are sent out. The modem can also run in secure packet mode where acknowleges are expected and packages can be retransmitted two times before they are dropped.
RF protocol includes a 16 bit CRC. Each data extracted from an RF packet with an invalid CRC is silently discarded by the state machine module. The CRC ensures that all data received are valid. It can be disabled by the user whose protocols already have a control mechanism integrity or when some bug fixes user protocols are implemented.

6.3 Conclusion

The RFD900x is specifically designed for use in unmanned vehicles. It can add information about the link status to the commonly used MAVLink protocol and Golay error correction and detection can be enabled in settings.
The ARF868URL provides no such options but can run in packet mode with a very basic resend behavior configurable.

7 Reliability

The outcome of the software refactoring (see Section 4.2) and the system analysis (see Section 5.3) was an application that runs stable and has enough CPU resources left for further features such as encryption and error correcting codes.
The main focus of this chapter is therefore the implementation of all features concerning the reliability of the data exchange. As mentioned in the Kapitel 2, Aeroscout GmbH estimates that about 10% to 20% of the packages sent are lost and that packages received contain errors due to interference. The application should be expanded to ensure reliability of the data transfer.
The implementation of reliability within this project is split into the following parts:

- Foreward Error Correction: Error detecting and error correcting code in case of interference
- Retransmission: Resend behavior in case of data loss
- Smart Wireless Selection: Algorithm for choosing the optimal modem to (re)send packages

7.1 Foreward Error Correction

The general idea for achieving error detection and correction is to add redundancy by encoding. The ground work for error detection and error correction methods has been done by Shannon in the 1940s. Shannon showed that every communication channel can be described by a maximal channel capacity with which information can be exchanged successfully. As long as the transmission rate is smaller or equal to the channel capacity, the transmission error could be arbitrarily small. When redundancy is added, possible errors can be detected or even corrected. Error-detection and correction schemes can be either systematic or non-systematic and it can either be a block code or a convolutionary code.
An example of a systematic block code is CRC, as the original information is unaltered and only accompanied by some redundancy.

Systematic Code

In a systematic scheme, the transmitter sends the original data, and attaches a fixed number of check bits (or parity data), which are derived from the data bits by some deterministic algorithm.

Non-Systematic Code

In a system that uses a non-systematic code, the original message is transformed into an encoded message that has at least as many bits as the original message.

Block Code

If the decoder only uses the currently received block of bits to decode the received message into an output, it is called an (n, k) block code with n being the number of output symbols and k the number of input symbols. Thereby, the last received block code does not matter for the decoding and is not stored.

Convolutionary Code

With convolutionary codes, the history of messages received is used to decode the next message.

7.1.1 Error Detection

According to [16], error detection is most commonly realized using a suitable hash function (or checksum algorithm). A hash function adds a fixed-length tag to a message, which enables receivers to verify the delivered message by recomputing the tag and comparing it with the one provided. The most widely known example is the CRC.

7.1.2 Error Correction

An error-correcting code (ECC) or forward error correction (FEC) code is a process of adding redundant data, or parity data, to a message, such that it can be recovered by a receiver even when a number of errors.
Error-correcting codes are usually distinguished between convolutional codes and block codes:

- Convolutional codes are processed on a bit-by-bit basis. They are particularly suitable for implementation in hardware, and the Viterbi decoder allows optimal decoding.
- Block codes are processed on a block-by-block basis. Examples of block codes are repetition codes, Hamming codes, Reed Solomon codes, Golay, BCH and multidimensional parity-check codes. They were followed by a number of efficient codes, Reed–Solomon codes being the most notable due to their current widespread use.

Reed Solomon

The following information has been taken from [7].
Reed Solomon is an error-correcting coding system that was devised to address the issue of correcting multiple errors, especially burst-type errors in mass storage devices (hard disk drives, DVD, barcode tags), wireless and mobile communications units, satellite links, digital TV, digital video broadcasting (DVB), and modem technologies like xDSL. Reed Solomon codes are an important subset of non-binary cyclic error correcting code and are the most widely used codes in practice. These codes are used in wide range of applications in digital communications and data storage.
Reed Solomon describes a systematic way of building codes that could detect and correct multiple random symbol errors. By adding t check symbols to the data, an RS code can detect any combination of up to t erroneous symbols, or correct up to $t/2$ symbols. Furthermore, RS codes are suitable as multiple-burst bit-error correcting codes, since a sequence of $b+1$ consecutive bit errors can affect at most two symbols of size b. The choice of t is up to the designer of the code, and may be selected within wide limits.
RS are block codes and are represented as RS (n, k), where n is the size of code word length and k is the number of data symbols, $n\check{}k = 2t$ is the number of parity symbols.
The properties of Reed-Solomon codes make them especially suited to the applications where burst error occurs. This is because

- It does not matter to the code how many bits in a symbol are incorrect, if multiple bits in a symbol are corrupted it only counts as a single error. Alternatively, if a data stream is not characterized by error bursts or drop-outs but by random single bit errors, a Reed-Solomon code is usually a poor choice. More effective cods are available for this case.
- Designers are not required to use the natural sizes of Reed-Solomon code blocks. A technique known as shortening produces a smaller code of any desired size from a large code. For example, the widely used (255,251) code can be converted into a (160,128). At the decoder, the same portion of the block is loaded locally with binary zero s.

- A Reed-Solomon code operating on 8-bits symbols has $n = 2^8 - 1 = 255$ symbols per block because the number of symbol in the encoded block is $n = 2^m - 1$
- For the designer its capability to correct both burst errors makes it the best choice to use as the encoding and decoding tool.

Golay

According to [12], there are two types of Golay codes: binary golay codes and ternary Golay codes. The binary Golay codes can further be devided into two types: the extended binary Golay code: G24 encodes 12 bits of data in a 24-bit word in such a way that any 3-bit errors can be corrected or any 7-bit errors can be detected, also called the binary (23, 12, 7) quadratic residue (QR) code.
The other, the perfect binary Golay code, G23, has codewords of length 23 and is obtained from the extended binary Golay code by deleting one coordinate position (conversely, the extended binary Golay code is obtained from the perfect binary Golay code by adding a parity bit). In standard code notation this code has the parameters [23, 12, 7]. The ternary cyclic code, also known as the G11 code with parameters [11, 6, 5] or G12 with parameters [12, 6, 6] can correct up to 2 errors.

7.1.3 Software Implementation of Error Detection and Error Correction

Because the Reed Solomon error correcting code requires a lot of CPU power and is intended for microcontrollers with more resources, the Golay error correcting code was chosen for this project.
Encoding and decoding is done inside the SPI Handler task. The use of the binary Golay error correting code can be enabled separately for each wireless connection by setting the USE_GOLAY_ERROR_CORRECTION to 1 in the configuration file located on the SD card. The Golay source code has been taken from Andrew Tridgell (see [9]) who provides an implementation that can be used without restrictions as long as his copyright is reproduced. The same Golay library has also been used in the ArduPilot, an alternative autopilot that is at least as popular as the Pixhawk used by Aeroscout GmbH.
Andrew Tridgells Golay library provides the following interface:

```
1  /*!
2  * \fn void golay_encode(uint8_t n, uint8_t* in, uint8_t* out)
3  * \brief Encodes n bytes of original data into n*2 bytes of encoded data
4  * \param n: number of bytes to encode, must be multiple of 3
5  * \param in: pointer to n bytes that will be encoded
6  * \param out: pointer to memory location where encoded data will be stored
7  */
8  void golay_encode(uint8_t n, uint8_t* in, uint8_t* out);
9
10 /*!
11 * \fn uint8_t golay_decode(uint8_t n, uint8_t* in, uint8_t* out)
12 * \brief Decodes n bytes of coded data into n/2 bytes of original data
13 * \param n: number of bytes to decode, must be multiple of 6
14 * \param in: pointer to n bytes that will be decoded
15 * \param out: pointer to memory location where decoded data will be stored
16 * \return number of 12bit words that required correction
17 */
18 uint8_t golay_decode(uint8_t n, uint8_t* in, uint8_t* out);
```

Because this Golay library uses global variables to save temporary data during encoding and decoding, the library is not reentrant. This is not an issue because the only task using the library is the SPI Handler.
The Golay library provides an interface to encode blocks of 3 bytes into blocks of 6 bytes and to decode blocks of 6 bytes into blocks of 3 bytes (see interface from code snippet above). There are two possibilities on how to use the library itself:

- Encoding only multiple of 3 bytes and decoding only multiple of 6 bytes, with the risk of delaying some bytes quite long
- Adding fill bytes if length of data to encode is not multiple of 3 bytes and adding fill bytes if length of data to decode is not multiple of 6 bytes. This could possibly destroy some code words

The option of encoding only multiple of 3 bytes respectively decoding only multiple of 6 bytes has been chosen within this application. Decoding encoded wireless packages then looks as follows:

```
1  /* read byte data from hw buffer */
2  if(spiSlave == MAX_14830_WIRELESS_SIDE && config.UseGolayPerWlConn[uartNr]) /* read and
       decode if Golay enabled */
3  {
4      /*
5      There's a tradeoff here: the number of data to be decoded needs to be a multiple of 6.
6      So we can either just read out as many bytes as there is multiple of 6, risking that
           we delay some of the bytes quite long.
7      Or we can read out all bytes, fill up with pseudo chars and destroy some of the
           codewords this way.
8      => decided to read out only multiples of 6
9      */
10     if((nofReadBytesToProcess % 6) > 0) /* nof data that will be read is NOT a multiple of
           six */
11     {
12         while((nofReadBytesToProcess % 6) > 0)      nofReadBytesToProcess--; /* read out
               multiples of 6 */
13     }
14
15     /* read byte data from the HW buffer and decode it */
16     spiTransfer(spiSlave, uartNr, MAX_REG_RHR_THR, READ_TRANSFER, encodedBuf,
           nofReadBytesToProcess); /* read out multiples of 6 */
17     nofErrors = golay_decode(nofReadBytesToProcess, &encodedBuf[1], &buffer[1]); /* decode
           */
18     nofReadBytesToProcess = nofReadBytesToProcess / 2; /* Golay doubled the data rate ->
           after decoding, only half is actual data */
19 }
```

Testing

The Golay error correcting code doubles the data. But after looking at the system performance in Section 5.3.3, this should not be a problem.
The application was tested by trying to establish a link between QGroundControl and the autopilot, analogous to Figure 5.1. The link could be established successfully and a periodic heartbeat was exchanged.
After consultation with Aeroscout GmbH, it was implied that their main focus does not lie with the error correcting code because they do not expect many bit errors in their data exchange but rather full package losses. Therefore no more time was invested in testing the implementation of the Golay algorithm. Before enabling this feature in a final product, more field tests need to be carried out.
In fact, the Golay error correcting code can also be enabled with one of the two modems used by Aeroscout GmbH. They reported that they have used a modem with Golay error correction enabled but have not seen a difference in the communication performance in general, so no more time has been invested in it since then.

7.1.4 Conclusion

The Golay error detecting and correcting code has been implemented and tested in a very basic manor. Golay can also be enabled on the RF900x modem used by Aeroscout. Because Aeroscout found

out that the use of an error correcting code does not affect the reliability of the data transmission and most data loss is due to packages not arriving at the receiver, no more time has been invested into foreward error correction.

7.2 Retransmission

According to Aeroscout GmbH, the most common issue with data exchange is not interference and the resulting bit errors but data loss. Because unmanned aerial vehicles constantly change their position, the data link per modem is not always reliable. Lost packages need to be retransmitted to ensure an uninterrupted data stream. This concept is also known as the Automatic Repeat Request (ARQ). According to [16], ARQ is an error control method for data transmission that makes use of error-detection codes, acknowledgment and/or negative acknowledgment messages (receiver detects missing package and requests retransmission), and timeouts to achieve reliable data transmission.
An acknowledgment (ACK) is a message sent by the receiver to indicate that it has correctly received a data package. When the transmitter does not receive the acknowledgment before the timeout occurs, it retransmits the frame until it is either correctly received or the error persists beyond a predetermined number of retransmissions. Acknowledges are often used in a synchronous transmission mode (as implemented in Section 4.2.6) so that the sender has to keep track of sent packages and no reordering is required on receiver side. This is typically used on wireless links with 10-20% package loss and a fast physical layer with a fast response time / acknowledge generating time.
Instead of sending an acknowledgement for all frames received correctly, the receiver could send negative acknowledgements (NACK) for all missing and/or faulty packages. Synchronous transmission

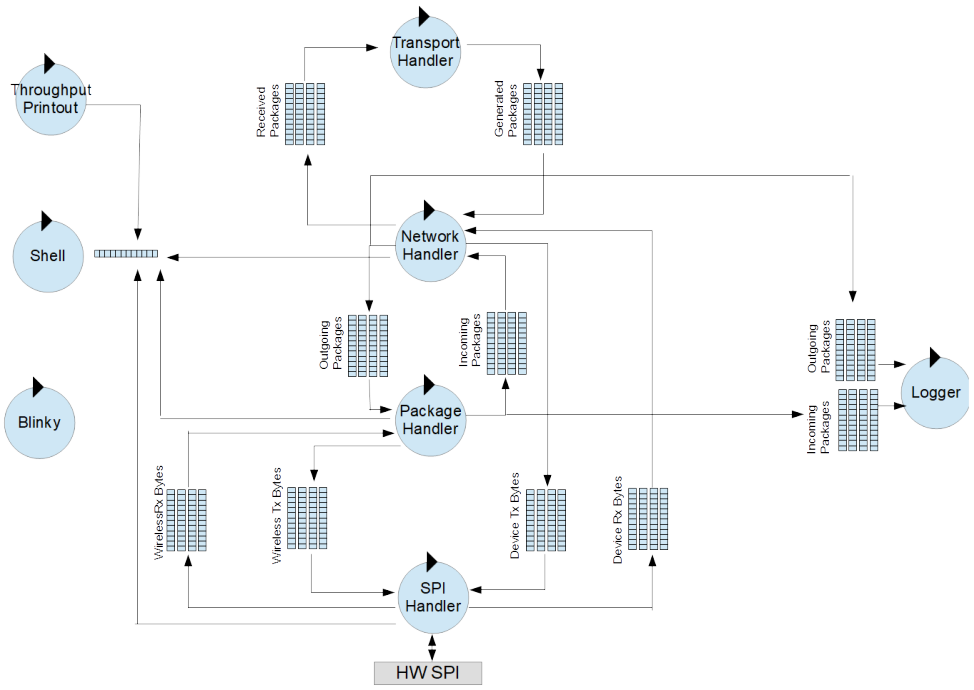

Figure 7.1: Full software concept with 4 ISO OSI layer tasks

(as implemented in Section 4.2.6) is not supported and payload reordering can only be done on receiver side. In practice, the receiver may not be able to reliably identify whether a package has been received, and the transmitter will usually also need to implement a timer to recover from the condition where the receiver does not respond.
Improvements on the ARQ behavior of this application have been done during the initial software refactoring in this project (see Section 4.2). Package numbering and payload numbering have been introduced, as well as a configuration parameter to enable/disable synchronous transmission handling (see Section 4.2.6).
This section focuses on a more sophisticated retransmission behavior. In the previous project, the application was implemented to iterate through the configured priorities and consecutively try to send out packages on the next lower priority modem. The Network Handler task was responsible for data routing and retransmission.
Within the scope of this project, retransmission behavior has been overhauled which resulted in an expanded software concept, replacing the concept mentioned in Figure 4.9.
The new software concept splits the responsibilities of the former Network Handler into two tasks: a Network Handler and a Transport Handler. As mentioned in Section 4.1.2, the software concept was based on the ISO-OSI model (see autoreffig:picIsoOsiModel), with each task representing a layer. The Network Handler was responsible for all upper layers but because the software grew in size, it now makes sense to add a task for ISO-OSI layer 4 and all upper layers and make the Network Handler responsible for ISO-OSI layer 3 only. The resulting software concept then looks as in Figure 7.1.
The new Network Handler is responsible for routing of the wireless packages and the Transport Layer does the retransmission. The ARQ and routing behavior of the application can be modified by changing configuration parameters inside the configuration file. Find information about how the responsibilities of the two new tasks was split and information about the configuration parameters that affect the routing and ARQ behavior of the application below.
The SPI Handler remains unchanged and the Package Handler now generates acknowledges. Find details about the changes made on each task below.

7.2.1 Package Handler

The Package Handler now generates acknowledges for data packages received successfully. Even though the responsibility of acknowledges received lies with the Transport Handler, the time it takes for a package to be processed by all lower layers and a generated acknowledge to traverse all lower layers again before being pushed out adds to the latency. Acknowledges should be generated as quickly as possible so the wait time if the application is configured for synchronous mode is as small as possible.
Acknowledge generation has previously been handled by the Network Handler but the latency proved to be too great as the interval at which each task is executed (which can be set in the configuration file) adds to this latency as well.

7.2.2 Network Handler

The new Network Handler task now only handles routing of the packages generated by the Transport Handler. Packages assembled by the Package Handler will be passed to the Network Handler but nothing is being done to them, they are passed onto the next queue to be processed by the Transport Handler. It is still necessary for the packages to transit through the Network Handler so that this task can process the throughput and assess how reliable one wireless link currently is. This information is needed to calculate the cost function to chose the optimal modem used for package transmission.

Packages generated by the Transport Handler are passed to the Network Handler and will be routed to the correct OutgoingPackages queue by this task. There are several configuration possibilities that determine the routing behavior of the Network Handler. See Section 7.2.4 for a detailed overview.

Next Steps for Network Handler

Currently, the Network Handler still handles the resending of unacknowledged packages. The responsibilities of the former Network Handler have not yet fully been split into two new tasks due to a lack of time. Currently, the only responsibility handled by the Transport Handler is the generation of data packages to be sent out on wireless side and the extraction of payload bytes from packages and pushing those to the TxDeviceBytes queue for the SPI Handler to send out.
With the overhaul of the software concept, the application does not work correctly with acknowledges enabled. More time should be invested into finding the source of this issue.

7.2.3 Transport Handler

The Transport Handler should keep track of the acknowledges received and the packages sent that expect an acknowledge back. This task should have an array to internally store the unacknowledged packages so they can be resent if they are not acknowledged within the timeout.
The Transport Handler is responsible for generating wireless packages from the incoming device stream. It accesses the DeviceRxBytes queue to remove bytes and put them into packages to be sent out.
The Transport Handler is responsible for extracting the payload from a wireless package and sending it out on device side by pushing the bytes to the correct DeviceTxBytes queue. Payload reordering should also be done in this task in case the packages arrive in the wrong order.
There are several configuration possibilities that determine behavior of the Transport Handler. See Section 7.2.4 for a detailed overview.

Next Steps for Transport Handler

Currently, the Transport Handler only generates wireless data packages and extracts device bytes from received wireless packages. In a next step, the software should be expanded so the Transport Handler keeps track of all pending acknowledges and does the payload reordering.

7.2.4 Configuration File

The configuration possibilities have been expanded in the scope of this project. The following section gives you an overview of all configuration options available that concern reliability of data exchange. A sample configuration file can be seen in .
Note that not all configuration possibilities have been implemented yet. Currently, the Network Handler only allows for LOAD_BALANCING = 1. The application has been refactored and expanded to soon support the other load balancing options as well. This following section describes how the application should behave with each configuration parameter, even if the configuration parameter is not supported yet.

ref zu config file im appendix

SEND_ACK_PER_WIRELESS_CONN

With this parameter, the use of acknowledges can be enabled/disabled. Only when enabled for one particular wireless connection will the package be stored internally for possible resending in case no acknowledge is received within the timeout.

All configuration parameters listed below are only applicable if acknowledges are enabled.
Currently, the application does not work correctly with this parameter enabled.

LOAD_BALANCING_MODE

This configuration parameter has been introduced in the scope of this project. It can possibly hold three values:

1. The wireless connection for the (re)send attempt is chosen according to the configuration file, first x attempts on the wireless connection with priority 1 , then y attempts on wireless connection with priority 2, etc. This is the behavior as implemented in the previous project.
2. The wireless connection for the (re)send attempt is chosen according to received acknowledges. Once an acknowledge is not received on one wireless connection, the next send attempts will be done with an other wireless connection. The memory of which wireless connection is currently reliable is done per device input. E.g. if one package generated from device input X does not get acknowledged on wireless connection with priority 1, all later packages from this device will be sent over wireless connection with priority 2 until one package fails to get acknowledged there, then it is back to wireless connection with priority 1. Note that this load balancing mode will only take wireless connections with priority 1 and 2 into account.
3. Smart Wireless Selection: The wireless connection for the (re)send attempt is chosen according to an algorithm that takes various aspects into account. More information in Section 7.3.

Currently, load balancing can only be set to 1 as the other two options have not been implemented yet. Software refactoring and improvements have been done to soon support the other two load balancing options.

PRIO_WIRELESS_CONN_DEV_X

This parameter determines the order at which wireless connections are used for resending unacknowledged packages.
It can be configured for each device and must hold values within the range 0...4. The wireless connection with highest priority has the value 1, second highest priority has the value 2 etc. Unused wireless connections for this device stream need to be configured to the value 0.
Priorities need to be configured continuously, starting with priority value 1 up to the desired maximum priority. No value can be skipped between 1 and the maximum.
If load balancing is set to 1 then resending is done on wireless connections in the order configured by this parameter. When configuring the same priority to multiple wireless connections, the same package will be sent out on multiple modems when (re)sending is done on this priority.
If load balancing is set to 2 then all resend attempts are done on wireless connections with priority 1 and 2, alternating only if a send attempt is unsuccessful.
If load balancing is set to 3 then .
This parameter is currently only supported when load balancing is set to 1.

how does this parameter affect resending algorithm?

SEND_CNT_WIRELESS_CONN_DEV_X

This parameter determines the number of transmission retries per wireless connection for each device data stream.
If load balancing is set to 1 then the priority order of wireless connections is traversed where SEND_CNT_WIRELESS_CON determines how many times a package is retransmitted on each wireless connection.
If load balancing is set to 2 then this parameter is not applicable because the wireless connection is switched upon first absence of an acknowledge.
If load balancing is set to 3 then .
This parameter is currently only supported when load balancing is set to 1.

does this parameter affect resending algorithm?

RESEND_DELAY_WIRELESS_CONN

Each generated package holds the payload from one device and is then sent out on wireless side. For each connected device, the user can configure over which wireless connection the generated package should be sent out first and the backup wireless connections used in case the package does not get acknowledged by the receiver.
This parameter specifies the acknowledge timeout for each wireless connection, how long the application waits for an acknowledge on a particular connection before retransmitting or discarding the package.
If load balancing is set to 1, the data stream will be forwarded to one wireless connection as long as the acknowledge is received within this timeout. If no acknowledge is received within this timeout, the data stream will switch to the other wireless connection. If load balancing is set to 2, the data stream of one device will be routed to one wireless side as long as the acknowledges arrive within this timeout. If a package is not acknowledged, the data stream will be forewarded to the other wireless connection configured.
If load balancing is set to 3, .
This parameter is currently only supported when load balancing is set to 1.

how does this affect algorithm?

DELAY_DISMISS_OLD_PACK_PER_DEV

This parameter has different meanings depending on the load balancing scenario chosen.
If load balancing is set to 1, then here are three possibilities how an internally stored package can get discarded:

- Acknowledge received, package transmission successful
- All configured retransmissions have been carried out but no acknowledge was received, package is dropped
- Timeout on the parameter DELAY_DISMISS_OLD_PACK_PER_DEV for a generated package from a particular device connection

If load balancing is set to 2, then this parameter determines how long one package is tried to be retransmitted before it is dropped. The number of retransmissions can be calculated by dividing this dismission timeout by the resend delays of the wireless connections used.
If load balancing is set to 3,

how does this parameter affect algorithm?

Configuration Validation

The Init task (see **??**) reads the configuration file on the SD card and sets the content of the global config variable. Afterwards, the configuration plausibility is verified and invalid parameter combinations are reset to their default value.

```
1  void validateSwConfiguration(void)
2  {
3      for(int devNr=0; devNr < NUMBER_OF_UARTS; devNr++)
4      {
5          // todo: validate software configuration!
6          if(!config.SendAckPerWirelessConn[devNr])
7          {
8              config.SyncMessagingModeEnabledPerWlConn[devNr] = 0;
9          }
10     }
11     /* constrain task execution intervals */
12     UTIL1_constrain(config.SdCardSyncInterval_s, 1, 1000); /* 1sec...1000sec */
13     UTIL1_constrain(config.SpiHandlerTaskInterval, 1, 1000); /* 1ms...1sec */
14     UTIL1_constrain(config.PackageHandlerTaskInterval, 1, 1000); /* 1ms...1sec */
15     UTIL1_constrain(config.NetworkHandlerTaskInterval, 1, 1000); /* 1ms...1sec */
16     UTIL1_constrain(config.TransportHandlerTaskInterval, 1, 1000); /* 1ms...1sec */
17     UTIL1_constrain(config.ShellTaskInterval, 1, 1000); /* 1ms...1sec */
```

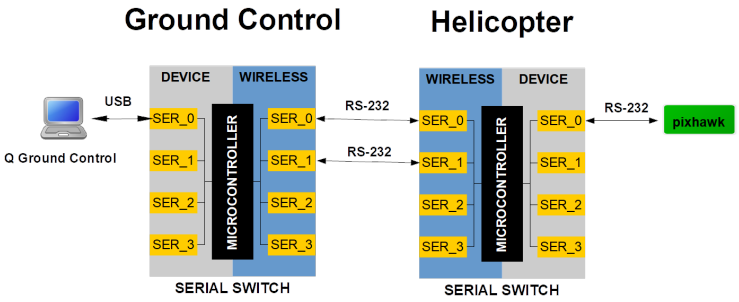

Figure 7.2: Setup for testing

```
18      UTIL1_constrain(config.LoggerTaskInterval, 1, 1000); /* 1ms...1sec */
19      UTIL1_constrain(config.ThroughputPrintoutTaskInterval_s, 1, 1000); /* 1sec...1000sec
            */
20      UTIL1_constrain(config.ToggleGreenLedInterval, 1, 1000); /* 1ms...1sec */
21  }
```

ex-
pand!

7.2.5 Testing

All tests have been carried out with QGroundControl and the Pixhawk autopilot set up as in Figure 5.1 and with load balancing configured to 1 because this is the only load balancing option that has been implemented so far.

Resending when Wireless Connection Lost

When load balancing is set to 1, the application handles retransmission as configured. Depending on the retransmission delay and the number of retries for each configured priority, the package is sent out again. All wireless connections are traversed in order of their configured priority, starting with the wireless connection priority 1.
The first test scenario was with the Pixhawk connected to serial interface 0 and two cable connections between wireless 0 and wireless 1, see Figure 7.2.The resend behavior was configured as follows:

```
1  SEND_ACK_PER_WIRELESS_CONN =    1, 1, 1, 1
2  LOAD_BALANCING_MODE =       1
3  PRIO_WIRELESS_CONN_DEV_0 =      1, 2, 0, 0
4  SEND_CNT_WIRELESS_CONN_DEV_0 =    4, 4, 0, 0
5  RESEND_DELAY_WIRELESS_CONN =    100, 100, 0, 0
6  DELAY_DISMISS_OLD_PACK_PER_DEV =  1000, 0, 0, 0
```

Four send attempts will therefore be done on wireless connection 0 before retrying four times on wireless connection 1.
When disconnecting the cable on wireless connection 0, the application first tried to resend the packages on wireless connection 0 and then switched to wireless connection 1 where they were received and acknowledges successfully.
When disconnecting both wireless 0 and wireless 1, the packages were dropped after the last unsuccessful send attempt.
When wireless connection 0 was only unplugged shortly and payload reordering was disabled in the configuration file, the link between QGroundControl and the autopilot was never dropped. The payload on receiver side was not in proper order (according to the log file) due to the resending of old packages with new packages arriving as well. QGroundControl seems to handle this error internally by reordering the data stream itself or just discards any data that cannot be processed.

Redundant Data Transmission

The second test scenario was with Pixhawk connected on serial interface 0 and two cable connections between wireless 0 and wireless 1, as in Figure 7.2. The resend behavior was configured as follows:

```
SEND_ACK_PER_WIRELESS_CONN =    1, 1, 1, 1
LOAD_BALANCING_MODE =       1
PRIO_WIRELESS_CONN_DEV_0 =      1, 1, 0, 0
SEND_CNT_WIRELESS_CONN_DEV_0 =    4, 4, 0, 0
RESEND_DELAY_WIRELESS_CONN =    100, 100, 0, 0
DELAY_DISMISS_OLD_PACK_PER_DEV =  1000, 0, 0, 0
```

Unlike in the first test scenario, duplicate packages are sent out on wireless connection 0 and 1. The receiver will acknowledge both packages but should only stream out the payload once on device side. This has been tested first with serial terminals connected to both Serial Switches and only string characters exchanged and should later be tested the setup of Pixhawk and QGroundControl.

7.3 Smart Wireless Selection

Blabla

8 Security

The outcome of the software refactoring (see Section 4.2) and the system analysis (see Section 5.3) was an application that runs stable and has enough CPU resources left for further features such as encryption and error correcting codes.
The main focus of this chapter is therefore the implementation of all features concerning the security of the data exchange. As mentioned in the Kapitel 2, Aeroscout GmbH would like the data exchange between the two Serial Switches to be encrypted and the exchanged data to be stored in a log file that cannot be modified without notice. Therefore, some means of information security has to be implemented.
According to [10], information security is the practice of preventing unauthorized access, use or modification of information. Its key concepts are:

- **Availability:** All systems are functioning correctly and information is available when it is needed.
- **Integrity:** The same data is received as was transmitted, it cannot be modified without detection.
- **Confidentiality:** Data can only be read by the intended receiver. This can be ensured by encrypting the data exchanged so that it is unreadable by anyone who does not have the decryption key.

Each key concept and its implementation is elaborated in more detail in this chapter.

8.1 Availability

Availability of information refers to ensuring that authorized parties are able to access the information when needed.
Information only has value if the right people can access the data at the right times.
Ensuring availability is not part of the project requirements (see Kapitel 2), it is only listed here for completeness.

8.2 Integrity

Integrity of data can be assured with a hash function. A hash is a string or number generated from byte data. The resulting hash is of fixed length, and will vary widely with small variations in input.
The only way to recreate the input data from an ideal cryptographic hash function's output is to attempt a brute-force search of possible inputs to see if they produce a match, or use a rainbow table of matched hashes. Therefore, hashing is a one-way function that scrambles plain text to produce a unique message digest.
A CRC is an example of a simple hash function and is used to check if the message received matches the message transmitted.
Integrity only does not provide security against tempering with the message itself. If someone knows the hash algorithm used, a message can be modified and its hash value recalculated without the receiver knowing about it.

8.2.1 Implementation of Integrity in Data Exchange

Data integrity during communication is implemented with a CRC value in the package header and payload to detect bit errors (see **??**). The main microcontroller (K64F) on the Teensy 3.5 has a hardware CRC module that was used for faster CRC calculations. There is a Processor Expert component that uses the hardware CRC module and this component can be configured and used for easier CRC calculation. In the source code, it looks as follows:

```
/* calculate CRC payload */
uint32_t crc16;
CRC1_ResetCRC(CRC1_DeviceData);
CRC1_SetCRCStandard(CRC1_DeviceData, LDD_CRC_MODBUS_16);
CRC1_GetBlockCRC(CRC1_DeviceData, pPackage->payload, pPackage->payloadSize, &crc16);
pPackage->crc16payload = (uint16_t) crc16;

/* calculate crc header */
CRC1_ResetCRC(CRC1_DeviceData);
CRC1_GetCRC8(CRC1_DeviceData, startChar);
CRC1_GetCRC8(CRC1_DeviceData, pPackage->packType);
CRC1_GetCRC8(CRC1_DeviceData, pPackage->devNum);
CRC1_GetCRC8(CRC1_DeviceData, pPackage->sessionNr);
CRC1_GetCRC8(CRC1_DeviceData, *((uint8_t*)(&pPackage->packNr) + 1));
CRC1_GetCRC8(CRC1_DeviceData, *((uint8_t*)(&pPackage->packNr) + 0));
CRC1_GetCRC8(CRC1_DeviceData, *((uint8_t*)(&pPackage->payloadNr) + 1));
CRC1_GetCRC8(CRC1_DeviceData, *((uint8_t*)(&pPackage->payloadNr) + 0));
CRC1_GetCRC8(CRC1_DeviceData, *((uint8_t*)(&pPackage->payloadSize) + 1));
pPackage->crc8Header = CRC1_GetCRC8(CRC1_DeviceData, *((uint8_t*)(&pPackage->payloadSize)
    + 0));
```

The CRC1 component uses the hardware CRC module for accelerated CRC computation.

8.2.2 Implementation of Integrity for Log Data

As seen in Kapitel 2, the application must also log the exchanged data and implement means to detect unauthorized modification of the log data. This can be achieved by implementing data integrity for the logged data. The easiest way to implement integrity in the Logger task is by adding a hash log value for every package logged. The log file then looks similar to this:

```
PackageType;DeviceNumber;SessionNumber;PackageNumber;PayloadNumber;PayloadSize;
CRC8_Header;Payload;CRC16_Payload;Hash

01;00;45;01;01;0011;88;FE092DFF0000000000000608C004039640;AD7C;154987
```

The Hash value was simply added after the CRC16_Payload entry.
The problem is that, knowing the hash algorithm used, data can still be tempered with by adding or deleting log lines inside the log file. To detect this misusage, a hash value should not only be calculated for every package but also over the entire log file so that it becomes more difficult to modify log data undetected. But because it would again be easy for malusers to detect the hash algorithm applied, modify the log file and recalculate the hash not only over single log lines but also over the entire modified file, an additional item of security has to be added. Instead of only calculating the hash over the entire file (in addition to the hash calculation over single log lines), a secret random number is also put into the hash algorithm of the entire file before the final hash output is extracted and logged. It lies within the nature of a hash algorithm that the hash output changes drastically with just a small variation of the input value. It is therefore nearly impossible for unauthorized users to find the secret random number used by looking at other hash outputs that were calculated over the entire file.
Each K64F microcontroller has a 128-bit unique identification number per chip. This number can be used as the secret random number to be put into the hash algorithm.

There is a hardware encryption module on the K64F microcontroller that supports the following hash functions:

- **MD5:**
  According to [19], the MD5 algorithm is a widely used hash function producing a 128-bit hash value. Although MD5 was initially designed to be used as a cryptographic hash function, it has been found to suffer from extensive vulnerabilities. It can still be used as a checksum to verify data integrity, but only against unintentional corruption.Like most hash functions, MD5 is neither encryption nor encoding. It can be cracked by brute-force attack and suffers from extensive vulnerabilities.
- **SHA-1:**
  According to [22], the Secure Hash Algorithm 1 (SHA-1) is a cryptographic hash function which takes an input and produces a 160-bit (20-byte) hash value as an output. Since 2005 SHA-1 has not been considered secure anymore and it is recommended to use SHA-2 or SHA-3 instead.
- **SHA-256:**
  The Secure Hash Algorithm 256 (SHA-256) is, just like the SHA-1, a cryptographic hash function. It generates a fixed size 256-bit (32-byte) hash output.

Although there are no Processor Expert components for either of these hash functions, there is a Crypto Acceleraction Unit (CAU) provided by NXP which is a encryption software library especially designed for the ARM Coretex-M4. The CAU is used for ColdFire and ColdFire+ devices while the mmCAU is for Kinetis devices (ARM Coretex-M4). For more information, consult the user guide and software API in [4].
This library can be imported into the Kinetis Design Studio project and used inside the software.

code snippet on how hash was calculated

8.3 Confidentiality

According to [3], encryption ensures that only authorized individuals can decipher data. Encryption turns data into a series of unreadable characters, that are not of a fixed length.
There are two primary types of encryption: symmetric key encryption and public key encryption.
In **symmetric key encryption**, the key to both encrypt and decrypt is exactly the same. There are numerous standards for symmetric encryption, the popular being AES with a 256 bit key.
Public key encryption has two different keys, one used to encrypt data (the public key) and one used to decrypt it (the private key). The public key is made available for anyone to use to encrypt messages, however only the intended recipient has access to the private key, and therefore the ability to decrypt messages.
Symmetric encryption provides improved performance, and is simpler to use, however the key needs to be known by both the systems, the one encrypting and the one decrypting data.
According to [17], information security uses cryptography to transform usable information into a form that renders it unusable by anyone other than an authorized user; this process is called encryption. Information that has been encrypted (rendered unusable) can be transformed back into its original usable form by an authorized user who possesses the cryptographic key, through the process of decryption.

8.3.1 Implementation of Encryption

The Teensy 3.5 uses the ARM Coretex-M4 microcontroller MK64FX512VMD12. In the MK64FX512xxD12 data sheet (see [6], p.1), it sais that this family supports the following hardware encryption algorithms:

- **DES:**
  According to [13] and [14], the Data Encryption Standard is a symmetric-key encryption algorithm developed in the early 1970s. It is now considered an insecure encryption standard and can be deciphered quickly, mostly due to its small key size (only 56 bit). While the small key size was generally sufficient when the algorithm was designed, the increasing computational power made brute-force attacks feasible.
- **3DES:**
  According to [14], the Triple Data Encryption Standard is a symmetric-key encryption algorithm which applies the DES cipher algorithm three times to each data block. Thereby, all three encryption keys can be identical or independent Theoretically, the resulting key length can be up to 3 x 56 bits = 168 bits. No matter if the keys are independent or identical, the resulting key has a shorter length due to vulnerability to different attacks.
- **AES:**
  According to [11], the Advanced Encryption Standard is, just like the DES, a symmetric-key algorithm and has been established in 2001. It superseded the DES and is now one of the most widely used encryption protocols. Its key sizes can either be 128 bits, 192 bits or 256 bits.

There is a Crypto Acceleraction Unit (CAU) provided by NXP which is a encryption software library especially designed for the ARM Coretex-M4. The CAU is used for ColdFire and ColdFire+ devices while the mmCAU is for Kinetis devices (ARM Coretex-M4). For more information, consult the user guide and software API in [4].
Because the RF900x modem already supports encryption and uses the AES algorithm with a 128bit key, this algorithm was also implemented in the application. The encryption key can be stored inside a header file so it is only visible to anyone who has access to the source code. Aeroscout claimed that it is not necessary for them to have a unique encryption key per Serial Switch and that they do not require the key to be changeable during runtime.

9 Conclusion

Usually, software is implemented in stages where each stage is followed by a refactoring phase. Step-like progress!

References

[1] "Bitfehlerraten-Messungen an GSM-Mobiltelefonen", in: *Neues von Rohde und Schwarz* 169 (2000), [20-Mar-2018, web page read], S. 11–13.

[2] Design, R., *RFD900x Radio Modem Data Sheet*, [25-Mai-2018, web page read], 2016.

[3] Europe, S. I., *Difference between hashing and encrypting*, [24-Apr-2018, web page read], 2016.

[4] Freescale, *ColdFire/ColdFire+ CAU and Kinetis mmCAU Software Library User Guide*, [10-Apr-2018, web page read], 2018.

[5] Labs, S., *FAT on SD Card*, [31-Mai-2018, web page read], 2018.

[6] NXP, *Kinetis K64F Sub-Family Data Sheet*, [10-Apr-2018, web page read], 2017.

[7] Priyanka Shrivastava, U. P. S., *Error Detection and Correction Using Reed Solomon Codes*, [24-Mai-2018, web page read], 2013.

[8] *title*.

[9] Tridgellh, A., *Golay Library*, [24-Mai-2018, web page read], 2012.

[10] Unknown, *Understanding Authentication, Authorization and Encryption*, [10-Apr-2018, web page read], 2018.

[11] Various, *Advanced Encryption Standard*, [10-Apr-2018, web page read], 2018.

[12] Various, *Binary Golay Code*, [11-Apr-2018, web page read], 2018.

[13] Various, *Data Encryption Standard*, [10-Apr-2018, web page read], 2018.

[14] Various, *Data Encryption Standard*, [10-Apr-2018, web page read], 2018.

[15] Various, *Digital Signature*, [11-Apr-2018, web page read], 2018.

[16] Various, *Error Detection and Correction*, [24-May-2018, web page read], 2018.

[17] Various, *Information Security*, [11-Apr-2018, web page read], 2018.

[18] Various, *ISO OSI Model*, [27-May-2018, web page read], 2018.

[19] Various, *MD5*, [10-Apr-2018, web page read], 2018.

[20] Various, *Secure Digital*, [31-Mai-2018, web page read], 2018.

[21] Various, *SEGGER SystemView*, [24-Apr-2018, web page read], 2018.

[22] Various, *SHA-1*, [10-Apr-2018, web page read], 2018.

Abbrevations

ACK	Acknowledgement, receiver sends confirmation that package has been received successfully
ARQ	Automatic Repeat Request
COM port	Simulated serial interface on computer
FatFS	File Allocation Table File System
GND	Ground reference, usually 0V
HSLU	Lucerne University of Applied Sciences and Arts
HW	Hardware
ISO/OSI	7 Layers Model
MCU	Micro Controller Unit
NACK	Negative acknowledgement, receiver sends message to sender that a package should be retransmitted
PC	Personal Computer
PCB	Printed Circuit Board
RS-232	Serial interface with +-12V
RTT	Real Time Transfer, Segger terminal
RTOS	Realtime Operating System
RX	Received signal
SD	Secure digital, memory card format
SDHC	Secure digital high capacity, memory card format
SPI	Serial Peripheral Interface, synchronous communication standard
SW	Software
SWD	Serial Wire Debug, hardware debugging interface
TX	Transmitted signal
TTL	Transistor Transistor Level, 5V level
UART	Universal Asynchronous Receiver Transmitter
UAV	Unmanned Aeral Vehicle
USB	Universal Serial Bus

Appendix A A

Lebenslauf

Personalien

Name	Stefanie Schmidiger
Adresse	Gutenegg 6125 Menzberg
Geburtsdatum	30.06.1991
Heimatort	6122 Menznau
Zivilstand	ledig

Ausbildung

August 1996 - Juli 2003	Primarschule, Menzberg
August 2003 - Juli 2011	Kantonsschule, Willisau
August 2008 - Juni 2009	High School Exchange, Plato High School, USA
August 2011 - Juli 2013	Way Up Lehre als Elektronikerin EFZ bei Toradex AG, Horw
September 2013 - Juli 2016	Elektrotechnikstudim Bachelor of Science Vertiefung Automation & Embedded Systems Hochschule Luzern - Technik & Architektur, Horw
Juli 2015 - Januar 2016	Austauschsemester, Murdoch University, Perth, Australien
September 2016 - jetzt	Elektrotechnikstudium Master of Science Hochschule Luzern - Technik & Architektur, Horw

Berufliche Tätigkeit

Juli 2003 - August 2003	Produktionsarbeiten bei Schmidiger GmbH, Menzberg
Juli 2004 - August 2004	Produktionsarbeiten bei Schmidiger GmbH, Menzberg
Juli 2005 - August 2005	Produktionsarbeiten bei Schmidiger GmbH, Menzberg
Juli 2006 - August 2006	Produktionsarbeiten bei Schmidiger GmbH, Menzberg
Juli 2007 - August 2007	Produktionsarbeiten bei Schmidiger GmbH, Menzberg
Juli 2009 - August 2009	Produktionsarbeiten bei Schmidiger GmbH, Menzberg
Juli 2010 - August 2010	Produktionsarbeiten bei Schmidiger GmbH, Menzberg
Juli 2014 - August 2014	Schwimmlehrerin bei Matchpoint Sports Baleares, Mallorca
September 2016 - jetzt	Entwicklungsingenieurin bei EVTEC AG, Kriens