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.

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.

Table of Contents

1 Introduction

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 Task Description

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 provides basic functionalities such as routing data between devices and modems and retransmission in case of data loss. The application was still in its first stage but mostly running stable and working correctly.
In the scope of this project, the software should first be refactored and all pending points that are necessary for further development should be implemented. The main goal of this project is to implement two more key features: Reliability and security. Both are elaborated in more detail below.

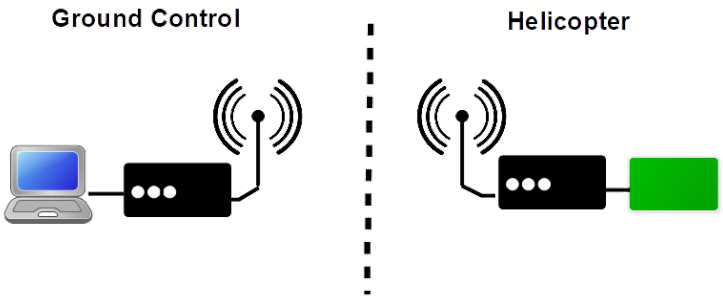

Figure 2.1: Previous system setup for data transmission

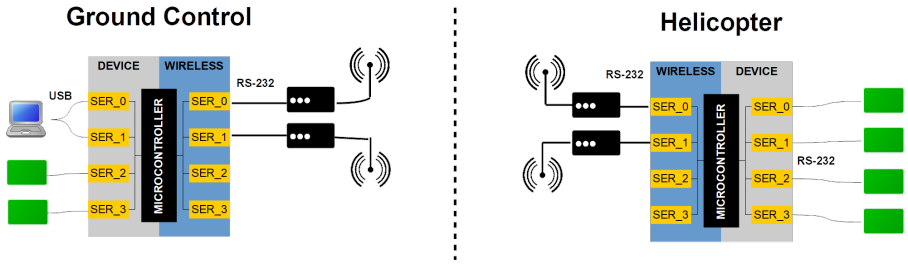

Figure 2.2: New system setup for data transmission

Reliability

Because unmanned aerial vehicles constantly change their position, transmission is not always reliable. About 10% - 20% of the transmitted data are lost and the transmitted 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 as well as implement an error correcting code.

Security

Data communication between the two serial switches should not be interceptable. A suitable encryption algorithm needs to be chosen that requires little computational power and little additional data transfer. A solution for encryption key generation and encryption key storage should be found.
Additionally, data transfer between the two serial switches should be logged, similar to the blackbox concept known from aviation. To ensure that the logged data cannot be tempered with unnoticed, the application should implement some sort of authenticity certificate to the log files.

3 Software Refactoring

In the scope of a previous project, the basic functionality of the Serial Switch was 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 security and reliability.
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 before starting with security and reliability features.

3.1 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.

3.1.1 Hardware

The hardware consists of three main components: a baseboard, the main microcontroller 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. 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 3.1. A block diagram about the components used on the baseboard can be seen

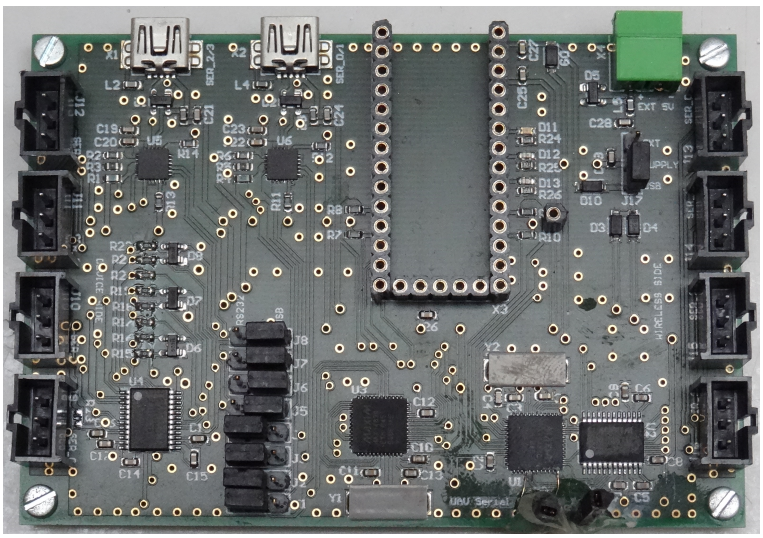

Figure 3.1: Baseboard

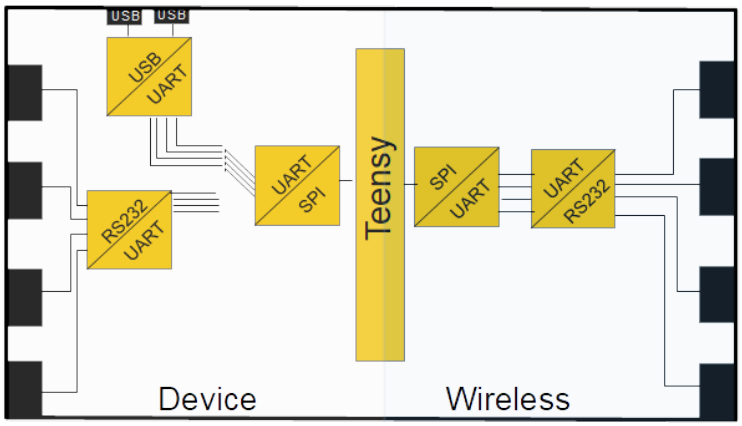

Figure 3.2: Block diagram of the components used on the baseboard

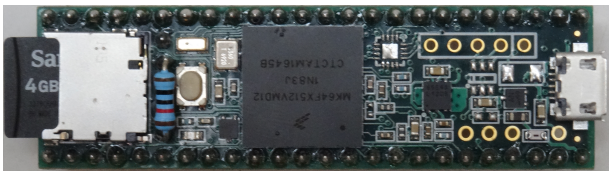

Figure 3.3: Teensy 3.5

in Figure 3.2.
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.

Microcontroller

In a first version of the Serial Switch, the Teensy 3.2 was used but was soon replaced by the Teensy 3.5 which is still used today. The Teensy 3.5 can be seen in Figure 3.3.

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 3.4.

3.1.2 Software

The Teensy 3.5 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 turned on, a package

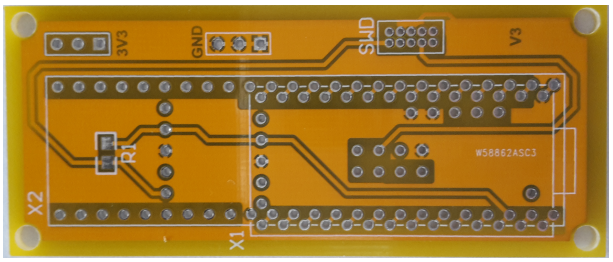

Figure 3.4: Adapter board from Teensy 3.2 to Teensy 3.5 footprint

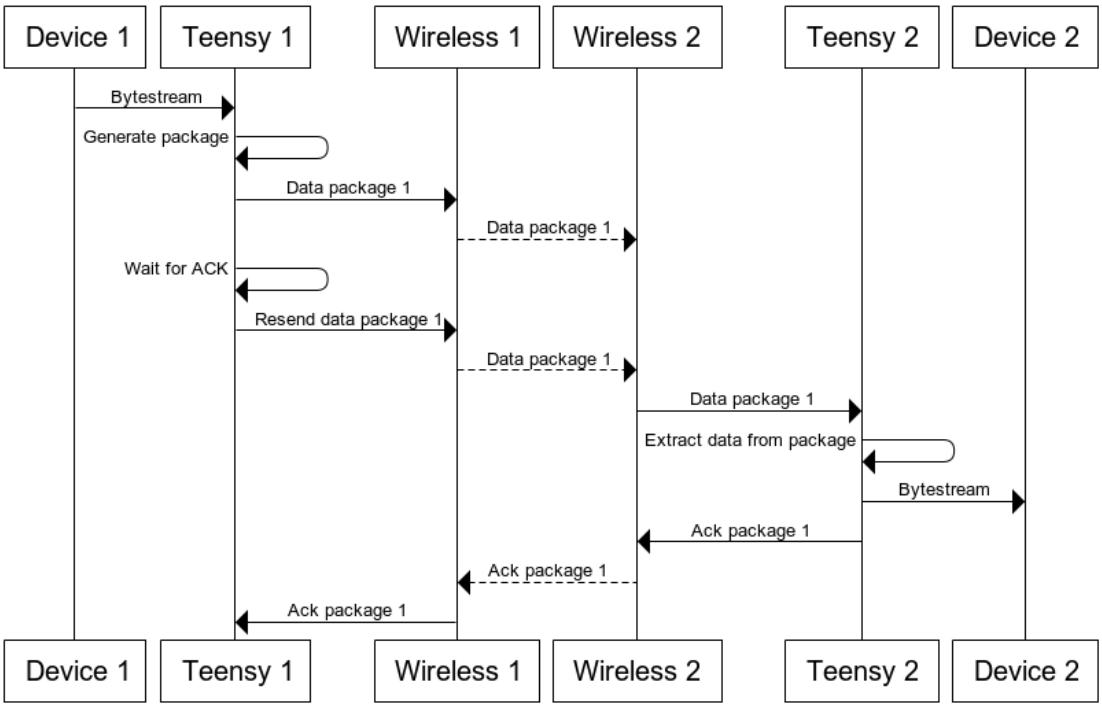

Figure 3.5: Package retransmission in case of no acknowledge received

is resent in case no acknowledge is received within the specified timeout. A visualization of this can be seen in Figure 3.5.
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 be seen in Figure 3.6. Task intercommunication is done with queues where 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 SPI Handler represents ISO-OSI Layer 1, the Package Handler represents ISO-OSI Layer 2 and the Network Handler represents 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 3.7.
Details about the purpose of each task can be taken from below.

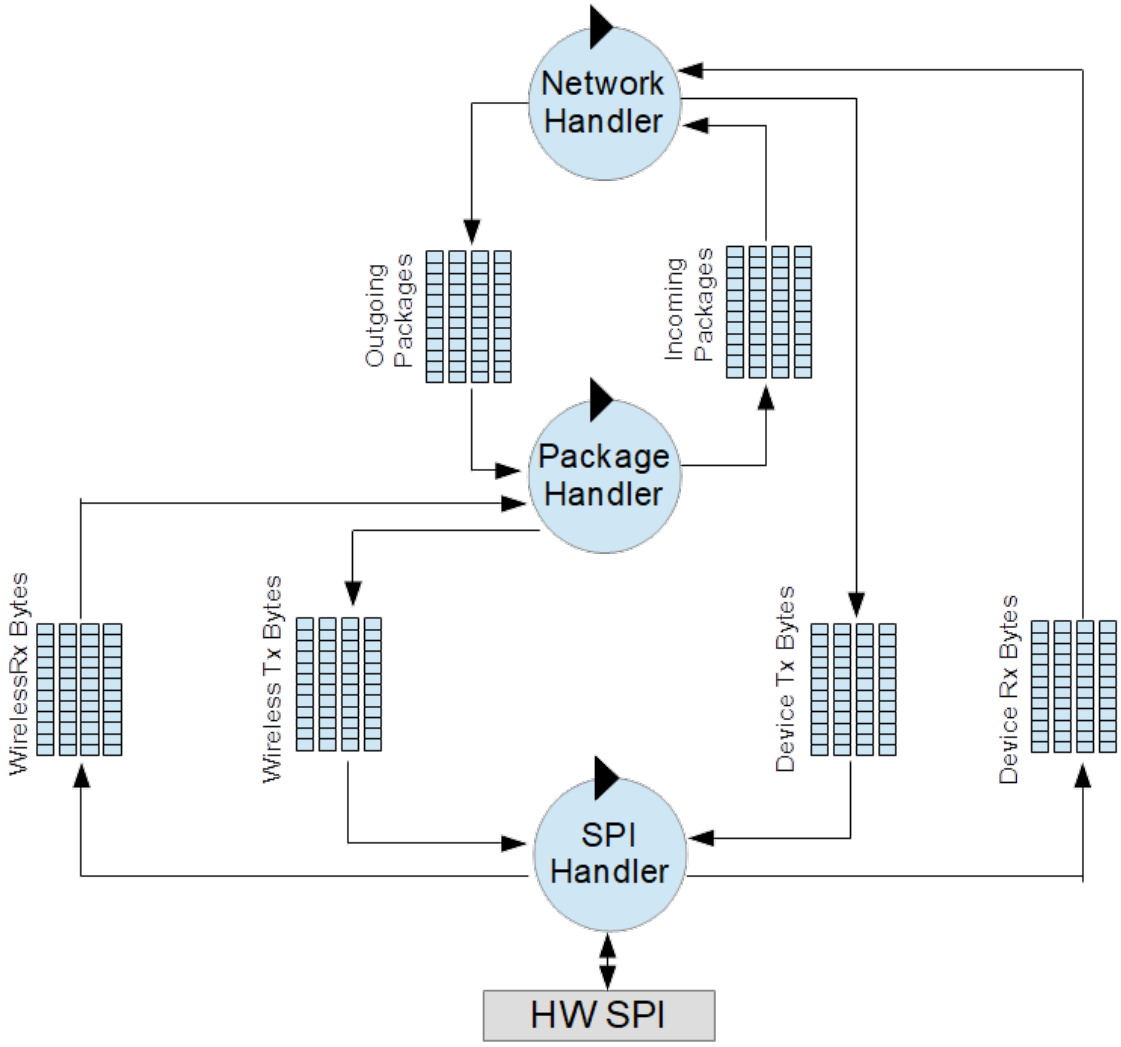

Figure 3.6: Simplified software concept at the beginning of this project, showing only the three main tasks

SPI Handler

The SPI Handler 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 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 IncomingPackages queue for the next task to process.
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.

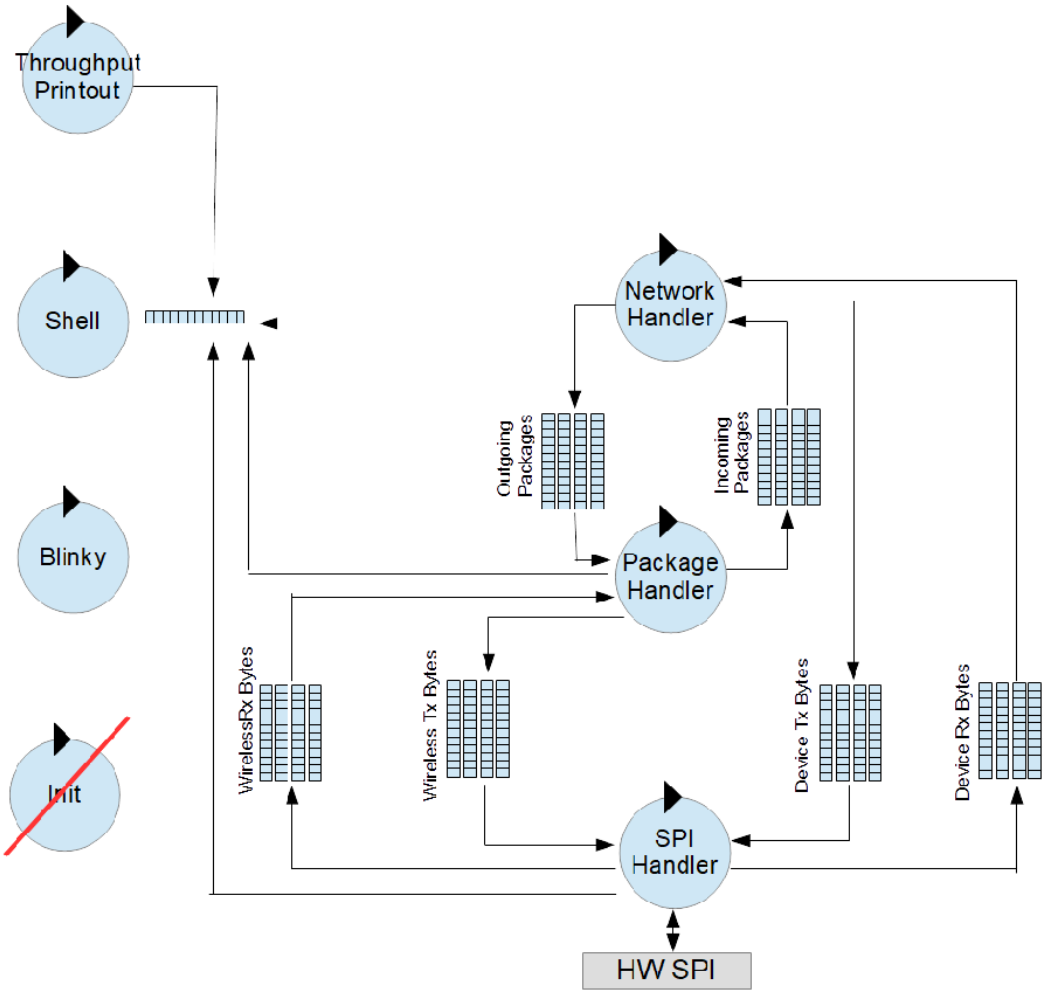

Figure 3.7: Full software concept at the beginning of this project, showing all tasks

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 "helpïnto the RTT terminal. The same terminal is also used by the Throughput Printout task to provide information about the performance of the application.

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 printed out 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 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 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.

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.

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 values possible:

- 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. 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

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. Only the major issues that need to be fixed or added are described shortly here.
The software does not yet support reordering of received packages. 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, 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. 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.

3.2 Added Features

Before starting with the implementation of new features, the current software needs to undergo refactoring and some features need to be added. The resulting software concept looks as in Figure 3.8. Details about the added features can be found below.

3.2.1 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

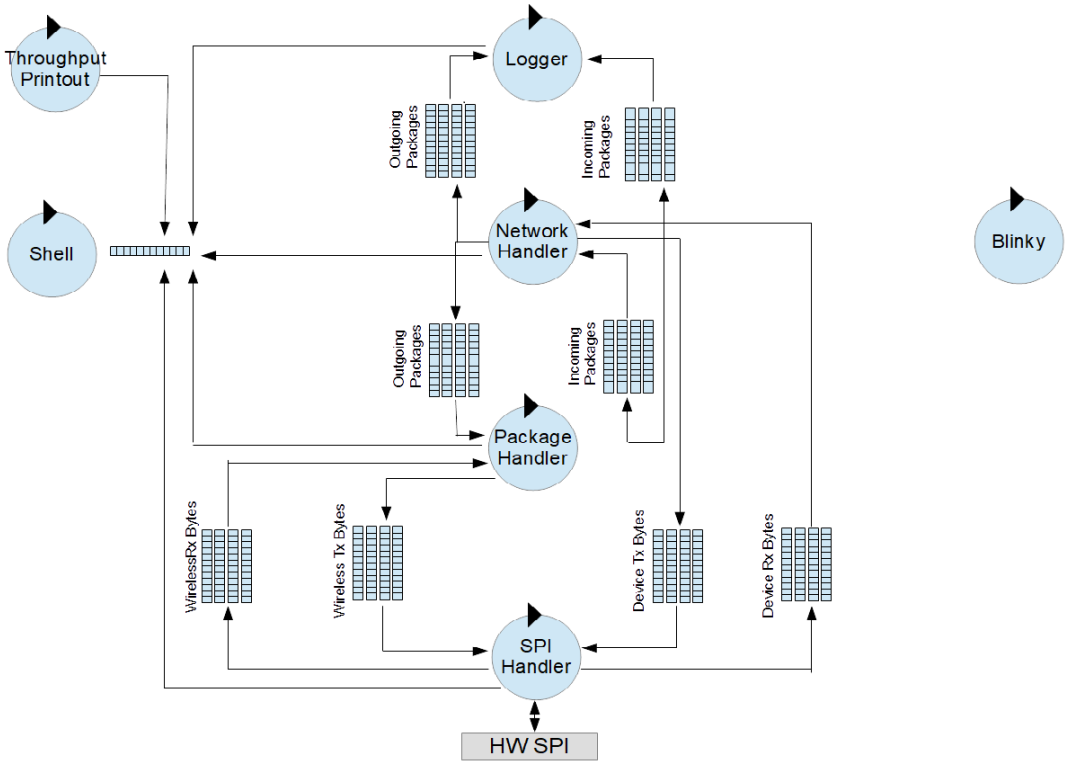

Figure 3.8: Software concept after added features

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.
Logging can be enabled by setting the LOGGING_ENABLED parameter to 1 in the configuration file located on the SD card.
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:

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

01;00;45;01;01;0011;88;FE092DFF0000000000000608C004039640;AD7C
01;00;45;02;02;0011;88;FE0930FF0000000000000608C0040371F4FE022EFF00150100;1245
01;00;45;03;03;0011;88;FE0931FF0000000000000608C004039B8A;63E4
01;00;45;04;04;0011;88;FE0932FF0000000000000608C00403A509;A3F6
01;00;45;05;05;0011;88;FE0933FF0000000000000608C004034F77FE2134FF004C0000;FD86
01;00;45;06;06;0011;88;FE0935FF0000000000000608C004032279;B5D3
01;00;45;07;07;0011;88;FE0936FF0000000000000608C004031CFA;75C1
01;00;45;08;08;0011;88;FE0937FF0000000000000608C00403F684;35CF
01;00;45;09;09;0011;88;FE0938FF0000803F0000000000000000000000000000000000000000;7822
01;00;45;0A;0A;0011;88;FE0939FF0000000000000608C00403F865;1985
01;00;45;0B;0B;0011;88;FE093AFF0000000000000608C0040C6E6;D997
```

The log file output can easily be imported into an Excel file, 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.

improvement by using sdhc or sw buffer

3.2.2 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.

3.2.3 Debug Output

The debug output on the shell looks as follows:

```
————————— add rtt output ————————-
```

Debug output

3.2.4 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 was then used that replaced the system time inside the package header. Each wireless connection now has a separate package counter. This was not sufficient because it was not possible to send the same package out on multiple wireless interfaces and discard all redundant packages on receiving side because the receiver would not know if a package was redundant because of the separate package counters per wireless connection. Therefore payload numbering

was introduced as well. Now the Network Handler can not only handle resending of unacknowledged packages and do payload reordering (thanks to the package number), but it can also discard redundant packages because each device byte stream now has its own continuous payload counter.

3.2.5 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 can be 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 will discard packages with payload numbers older than the one previously sent out on device side.

All payload reordering and package reordering is being done by the Network Handler.

3.2.6 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 has been acknowledged
- **Asynchronous transmission:** Packages are sent out continuously without waiting on the acknowledge of the previous package

3.2.7 Static Memory Allocation

Previously, memory for all tasks and queues was allocated dynamically which resulted in blablabla. 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.

4 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.
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 4.1. 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.
In the previous project, SEGGER SystemView was used to analyze the runtime behavior and CPU load of the application.
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.[15].
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.
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.

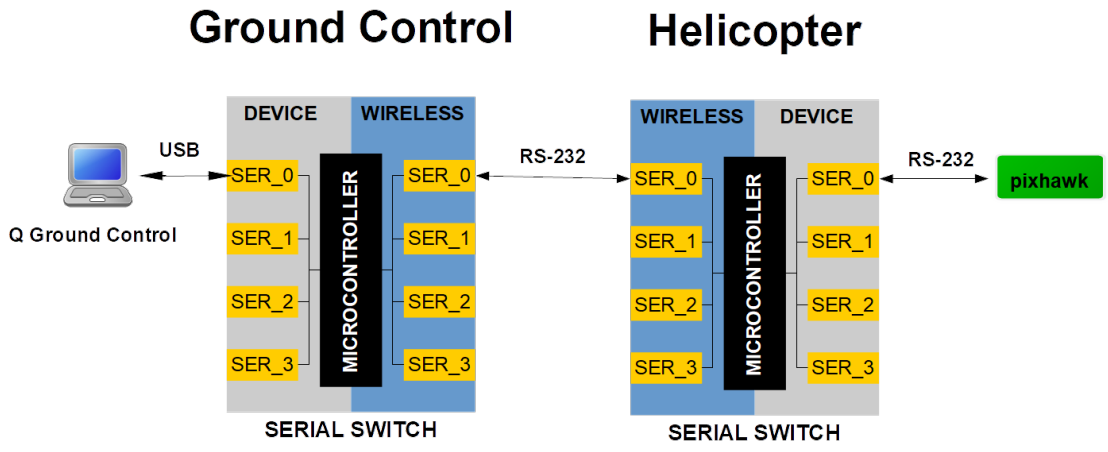

Figure 4.1: Test setup with autopilot and ground station

4.1 SEGGER SystemView

The application developed in the scope of this project runs with four main tasks 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 thousands 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.

4.2 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 looked as though 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:

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

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 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 4.2, 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:

```
void FRTOS_vApplicationTickHook(void)
{
    /* Called for every RTOS tick. */
    TMOUT1_AddTick();
    TmDt1_AddTick();
    Pin33_NegVal(); /* toggle Pin 33  on Teensy */
}
```

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 4.3. When lots of data is exchanged, the system tick event is indeed not called regularly as can be seen when measuring the output of Pin33 (see Figure 4.4). 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.

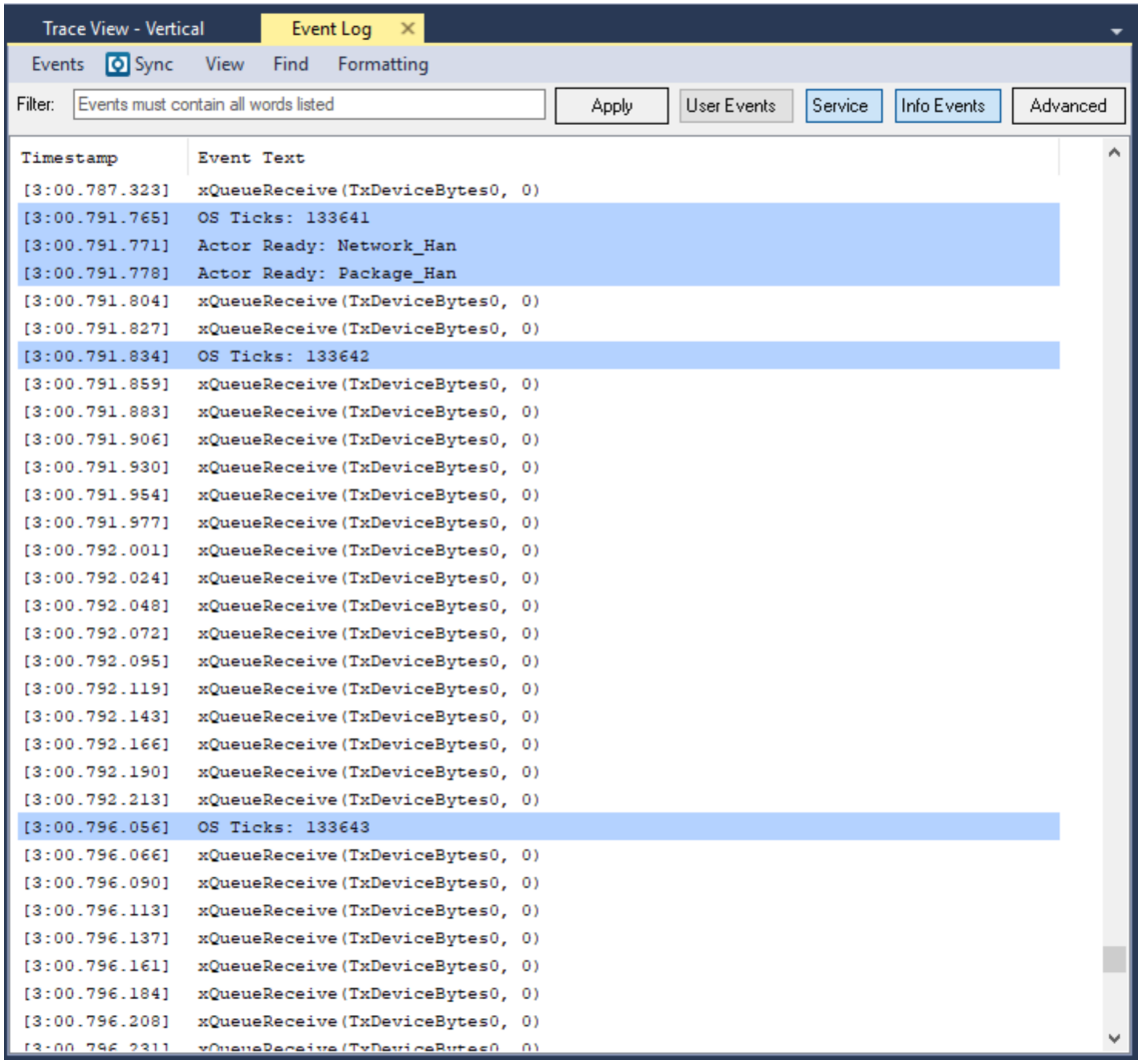

Figure 4.2: A queue operation taking 4 milliseconds

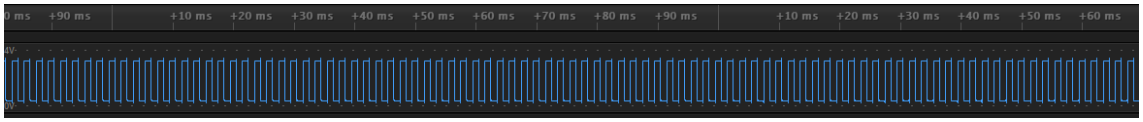

Figure 4.3: Periodic system ticks when Percepio disabled

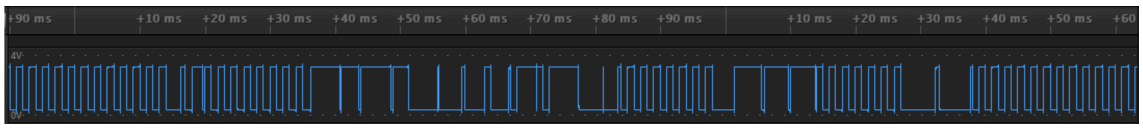

Figure 4.4: Irregular system ticks when Percepio enabled

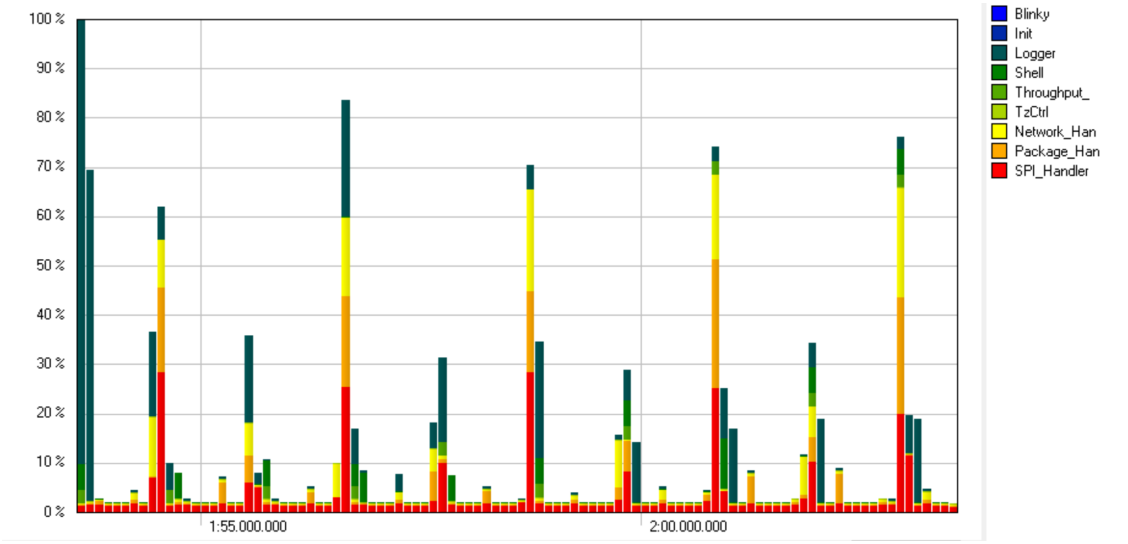

Figure 4.5: System load during periodic heartbeat exchange

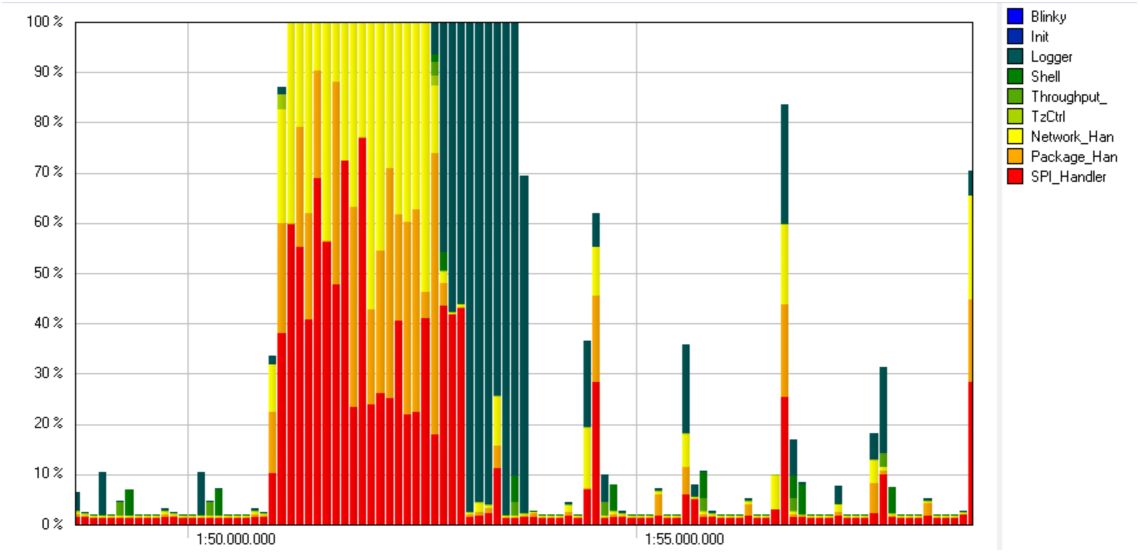

Figure 4.6: System load during link establisment

4.2.1 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 4.4.
Therefore the source of the issue must be located with Percepio Trace.

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

4.2.3 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 4.1.
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 4.5. During link establishment the output looks as in Figure 4.6.
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 3.1.2 for details about task priorities.
The logger task takes up a lot of CPU time when lots of packages are exchanged. The reason for this can be found in the FatFs. As mentioned in Section 3.2.1, 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 flushed or upon a FatFs sync command. A write process to the SD card is an expensive operation, especially when data is written across different sectors.

SD card!

5 Reliability

The outcome of the software refactoring and the system analysis was an application that runs stable and 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
- Algorithm for choosing the optimal modem to (re)send packages

5.1 Foreward Error Correction

The general idea for achieving error detection and correction is to add some redundancy. Error-detection and correction schemes can be either systematic or non-systematic: 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. 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.
The aim of encoding a message is to get the content to the recipient with minimal errors. 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.

5.1.1 Error Detection

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. Example: CRC.[12]

5.1.2 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

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, $n\breve{}k = 2t$ 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.

[4]

Golay

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.

5.1.3 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 per 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 ([6])) who provides an implementation that can be used without restrictions as long as his copyright is reproduced. This 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.
The Golay library provides the following interface:

```
/*!
* \fn void golay_encode(uint8_t n, uint8_t* in, uint8_t* out)
* \brief Encodes n bytes of original data into n*2 bytes of encoded data
* \param n: number of bytes to encode, must be multiple of 3
* \param in: pointer to n bytes that will be encoded
* \param out: pointer to memory location where encoded data will be stored
*/
void golay_encode(uint8_t n, uint8_t* in, uint8_t* out);

/*!
* \fn uint8_t golay_decode(uint8_t n, uint8_t* in, uint8_t* out)
* \brief Decodes n bytes of coded data into n/2 bytes of original data
* \param n: number of bytes to decode, must be multiple of 6
* \param in: pointer to n bytes that will be decoded
* \param out: pointer to memory location where decoded data will be stored
* \return number of 12bit words that required correction
*/
uint8_t golay_decode(uint8_t n, uint8_t* in, uint8_t* out);
```

Because this implementation uses global variables to save 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:

```
/* read byte data from hw buffer */
if(spiSlave == MAX_14830_WIRELESS_SIDE && config.UseGolayPerWlConn[uartNr]) /* read and
    decode if Golay enabled */
{
    /*
    There's a tradeoff here: the number of data to be decoded needs to be a multiple of 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.
    Or we can read out all bytes, fill up with pseudo chars and destroy some of the
        codewords this way.
    => decided to read out only multiples of 6
    */
    if((nofReadBytesToProcess % 6) > 0) /* nof data that will be read is NOT a multiple of
        six */
    {
        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 4.2.3, this should not be a problem.
The application was tested by trying to establish a link between QGroundControl and the autopilot, analogous to Figure 4.1. 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 transmission 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 data loss or the communication performance in general, so no more time has been invested in it since then.

5.2 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 aeral 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).

copy text from arq

Improvements on retransmission behavior have been done during the first software refactoring in this project (see Section 3.2). Package numbering and payload numbering has been introduced, as well as synchronous transmission handling.

6 Modems

6.1 RF686x RFD900x

The RF686x 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)
- The RFD900x has two antenna ports and 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 compatible with many configuration methods like the AT Commands and APM Planner.
- Golay error correcting code can be enabled (doubles the over-the-air data usage)
- MAVLink framing and reporting can be turned on and off.
- Encryption level either off or 128bit

The 128bit AES data encryption may be set by AT command. 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 modems.

6.1.1 Peer to peer network

Abb: P2P
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.

6.1.2 Asynchronous non-hopping mesh

It is a straight foreward connection between two and more nodes. As long as all the nodes are within range and have compatible parameters, communication between them will succeed.

6.1.3 Multipoint network

Abb: P2MP, PTMP or PMP
In a multipoint connection, the link is between a sender and multiple receivers.

6.1.4 MAVLink

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 and speed.

6.1.5 AT Commands

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, you 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 such as RSSI signal report.

6.1.6 SiK

A SiK Telemetry Radio is a small, light and inexpensive open source radio platform that typically allows ranges of better than 300m "out of the box" (the range can be extended to several kilometres with the use of a patch antenna on the ground). The radio uses open source firmware which has been specially designed to work well with MAVLink packets and to be integrated with the Mission Planner, Copter, Rover and Plane.
SiK radio is a collection of firmware and tools for telemetry radios.
Hardware for the SiK radio can be obtained from various manufacturers/stores in variants that support different range and form factors. Typically you will need a pair of devices - one for the vehicle and one for the ground station.
A SiK Telemetry Radio is one of the easiest ways to setup a telemetry connection between your APM/Pixhawk and a ground station.
You can use the MAVLink support in the SiK Radios to monitor the link quality while flying, if your ground station supports it

6.2 ARF868URL

All information from datasheet:
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 into a single radio packet can reach 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. To prevent from input fifo overrun, the hardware flow control may be activated. In this case, the RTS signal will be set when the incoming fifo is almost full to prevent the host controller from sending new data.
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.

6.2.1 RSSI

This modem can also transmit RSSI values

7 Channel Parametrization

Die wichtigste Eigenschaft eines Empfängers ist seine Empfindlichkeit. Bei digitalen Systemen wird sie über die Bitfehlerrate (BER) bestimmt.
Hierzu wird dem Empfänger ein Testsignal mit einer Pseudo-Random-Bitfolge und einem definierten Pegel zugeführt und an seinem Ausgang die Anzahl der Bitfehler gemessen. Das Grundprinzip der BER-Testmodi ist einfach: Der Funkmessplatz sendet einen Datenstrom an das Mobiltelefon, der vom diesem wieder zurückgesendet wird (Loop). Der Messplatz vergleicht gesendete und empfangene Datenströme und ermittelt daraus die Anzahl der Bitfehler. [1]

8 Error Correcting Codes

In information theory and coding theory with applications in computer science and telecommunication, error detection and correction or error control are techniques that enable reliable delivery of digital data over unreliable communication channels. Many communication channels are subject to channel noise, and thus errors may be introduced during transmission from the source to a receiver. Error detection techniques allow detecting such errors, while error correction enables reconstruction of the original data in many cases.
Common channel models include memory-less models where errors occur randomly and with a certain probability, and dynamic models where errors occur primarily in bursts. Consequently, error-detecting and correcting codes can be generally distinguished between random-error-detecting/correcting and burst-error-detecting/correcting. Some codes can also be suitable for a mixture of random errors and burst errors.
The general idea for achieving error detection and correction is to add some redundancy. Error-detection and correction schemes can be either systematic or non-systematic: 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. 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.
The aim of encoding a message is to get the content to the recipient with minimal errors. 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.
Generally, there are two ways to correct a faulty message once received:

- Foreward error correction
- Retransmission of the faulty message

Hier Bild einfügen von INKT Vorlesung: Info -> Encoder with Code Rate R -> Modulation -> Versand -> Empfang -> Demodulation -> Decoding -> Output

Block Codes and Convolution Codes

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. With convolutionary codes, the history of messages received is used to decode the next message.
Cyclic Codes
Cyclic codes are a subgroup of block codes. A random sequence of 1 and 0 are shifted to form other codes and inverted for even more codes. They can be represented as polynomes where the power of X represents the 1 in a sequence, starting with X^0 as the leftmost bit.
Systematic Block Codes
When the original information bits are unaltered and only accompanied by some redundancy such as CRC, the block code word is called systematic.

8.1 Error Detection

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. Example: CRC.

8.2 Error Correction

8.2.1 Automatic Repeat Request (ARQ)

ARQ is an error control method for data transmission that makes use of error-detection codes, acknowledgment and/or negative acknowledgment messages, and timeouts to achieve reliable data transmission. An acknowledgment is a message sent by the receiver to indicate that it has correctly received a data frame. Usually, 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.

8.2.2 Error Correcting Code

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

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 – 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

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 tis 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.

8.2.3 Hybrid

When both Foreward correction and automatic repeat requests are enabled, it is called hybrid. [5]

9 Information Security

Information security is the practice of preventing unauthorized access, use or modification of information [7]. It key concepts are:

- 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.
- Availability: All systems are functioning correctly and information is available when it is needed.

To ensure these key concepts, different measurements can be taken, such as:

- Authentication: A means for one party to verify another's identity, e.g. by entering a password
- Authorization: Process by which one party verifies that the other party has access permission
- Encryption: Process of transforming data so that it is unreadable by anyone who does not have a decryption key

A mathmatical way to demonstrate authenticity and integrity of data is by using a digital signature. Each of these terms is elaborated in more details in the following sections.

9.1 Authentication

In information security, to verify if the data provider really is the one intended source, respectively if the data recepient really is the intended recepient

9.2 Authorization

9.3 Integrity

Integrity of data can be assured with a hash function. A hash is a string or number generated from a string of text. The resulting string or number is a 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.

9.4 Confidentiality

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.

9.5 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 it at the right times.

9.6 Encryption

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. [13]

9.6.1 Encryption Algorithms Supported by Teensy 3.5

The Teensy 3.5 uses the ARM Coretex-M4 micro controller MK64FX512VMD12. In the MK64FX512xxD12 data sheet (see [3], p.1), it sais that this family supports hardware encryption such as DES, 3DES, AES, MD5, SHA-1, SHA-256. The supported encryption standards are elaborated in more detail in the following subsections.
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 [2].

DES

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). [9] While the small key size was generally sufficient when the algorithm was designed, the increasing computational power made brute-force attacks feasible. [10]

3DES

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. [10]

AES

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. [8]

MD5

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. [14]

SHA-1

The Secure Hash Algorithm 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. [16]

SHA-256

The Secure Hash Algorithm 256 is, just like the SHA-1, a cryptographic hash function. It generates a fixed size 256-bit (32-byte) hash output.

9.7 Digital Signature

Digital signature is a mathematical scheme to demonstrate authenticity of a digital message. A valid digital signature gives a recipient reason to believe that the message was created by a known sender (authentication), that the sender cannot deny having sent the message (non-repudiation) and that the message was not altered in transit (integrity) [11].
A digital signature usually consists of three algorithms:

- Key generation algorithm: selects a random private key, outputs it and the corresponding public key.
- Signing algorithm: produces a signature when given a message and a private key.
- Signature verification: Verifies the authenticity of a message, when given the message, its signature and public key.

Non-repudiation can only be assured when an asymmetric encryption algorithm has been used, which means when the message was signed using a private key and can be verified using the corresponding public key. Symmetric encryption is not used for digital signatures because the sender and receiver are both in possession of the same key, therefore the receiver knows the scheme and cannot prove that

it has not been the creator of the message and its signature.

10 Conclusion

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

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

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

[5] *title*.

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

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

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

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

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

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

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

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

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

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

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

Abbrevations

ALOHA	System for coordinating access to a shared communication channel
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
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
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