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 on-board and off-board components, different data transmission technologies have to be used.
In this project, a hardware has been designed where multiple data inputs and outputs and multiple transmitters can be connected to a serial switch. The designed hardware features an SD card with a configuration file where data routing can be configured.
Data from connected devices will be collected and put into a data package and sent out via the configured transmitter. The corresponding second serial switch hardware receives this package, extracts and verifies the payload, sends it out to the corresponding device and sends an acknowledge back to the package sender.
When data transmission with one transmitter fails, the configuration file lets the user select the order of back up transmitters to be used. Data priority can also be configured because reliability of data transmission is extremely important with information such as exact location of the drone but not as important with information such as state of charge of the battery.
The serial switch hardware designed in the scope of this project features four serial RS232 connections where input and output devices can be connected that process or generate data. There are also four RS232 connectors where transmitters can be connected to send or receive data packages. The routing between data generating devices and transmitters to use can be done in a .ini file saved on an SD card.
There are two SPI to UART converters that act as the interface between the four devices connected and the micro controller respectively the four transmitters and the micro controller.
In a first version of the project, 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 first version of the software ran with more than ten tasks and an overhaul of the complex structure was necessary.
For this reason, an adapter board has been designed so the existing hardware could be used with the more powerful Teensy 3.5 and a hardware debugging interface.
The Teensy 3.5 was then configured to run with FreeRTOS. Task scheduler and queues provided by this operating system have been used to develop software that extracts data from received packages to output them on the configured interface or generates packages from received data bytes to send them out over the configured transmitter. The concept of acknowledges has also been applied so package loss can be detected and lost packages can be resent.
The software concept implemented 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.

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 on-board and off-board components, different data transmission technologies have to be used.
In a previous project, the hardware for a Serial switch has been designed that features four RS232 interfaces to connect data processing and generating devices and four RS232 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. It is a small, inexpensive and powerful USB development board for Arduino applications. The software was flexible and in its header files the user could configure individual baud rates for each RS232 interface, data routing and the use of acknowledges for data packages for each modem side. The application was running with many tasks, 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 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 the on-board 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 received, 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 RS232 interfaces to connect different data generating and processing devices and four RS232 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. 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

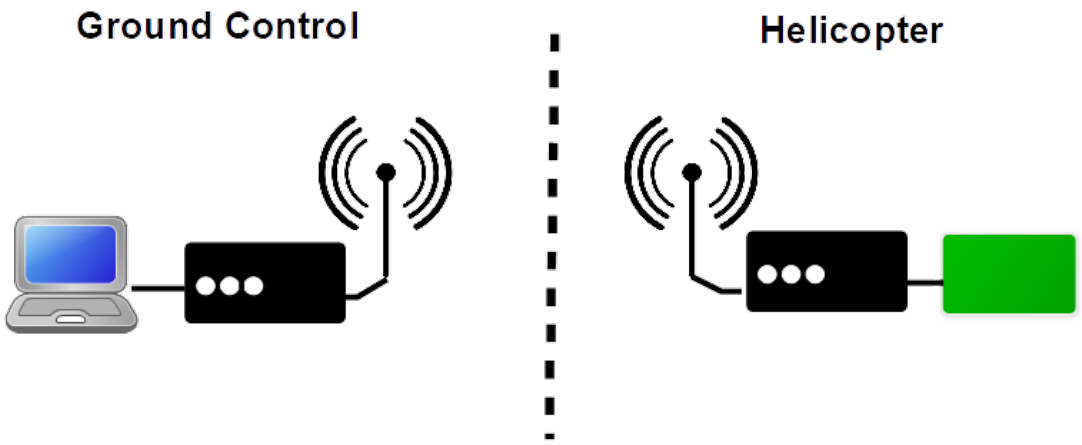

Figure 2.1: Previous system setup for data transmission

The aim of this project is to provide a solution that provides the needed flexibility. The finished product should act as a serial switch with multiple input/output interfaces for connecting devices and sensors and multiple interfaces for connecting transmitters. When one transmission technology fails to successfully transmit data, an other technology can be chosen for the next send attempt. Also, multiple sensors or input streams should be able to send out data over the same wireless connection. A visualization of this set up can be seen in Figure 2.2

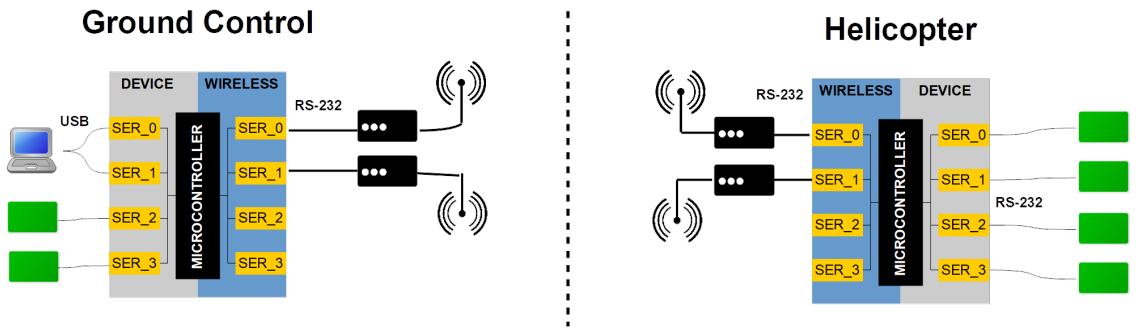

Figure 2.2: New system setup for data transmission

There are various kinds of information exchanged between drone and control device such as state of charge of the battery, exact location of the drone, control commandos etc. Some information such as the exact location of the drone should be prioritized over battery status information when data transmission becomes unreliable. The finished product should therefore take data priority into account.
Encryption should be configurable individually for each modem interface in case sensitive data is exchanged over a connection.
The finished product should be have a hardware debugging interface such as SWD and a shell/command line interface. During run time, the software should log system data and any other relevant information to a file saved on an SD card. The SD card should also contain a configuration file so the behavior of the hardware can be changed easily.
Also, the Free RTOS should go into low power mode during idle state and software performance and memory leaks should be detected with Free RTOS Trace.
Optionally, the application should encrypt sensitive data and use the concept of interleaving to possibly recover data on receiving side in case of package loss during unreliable transmission.
A detailed description of all the requirements can be taken from appendix B.

3 Starting Situation

It was not necessary to start from scratch for this project.
In the beginning of 2017, Andreas Albisser has already started with an implementation and provided a first solution.
He developed a hardware that was used as the interface between input/output data and modem for wireless transmission. He chose the Teensy 3.2 development board as a micro controller and worked with the Arduino IDE and Arduino libraries.
There are various problems still with his work which lead to this follow up project to improve the overall functionality.
More details about the work Andras Albisser has done can be taken from this chapter.

3.1 Hardware

The hardware developed by Andreas Albisser ca be seen in Figure 3.1. It has a total of eight RS232 interfaces where peripheral devices can be connected. Four connections are for control units, sensors or any other devices that process or generate data to be transmitted. On the other side, there are four

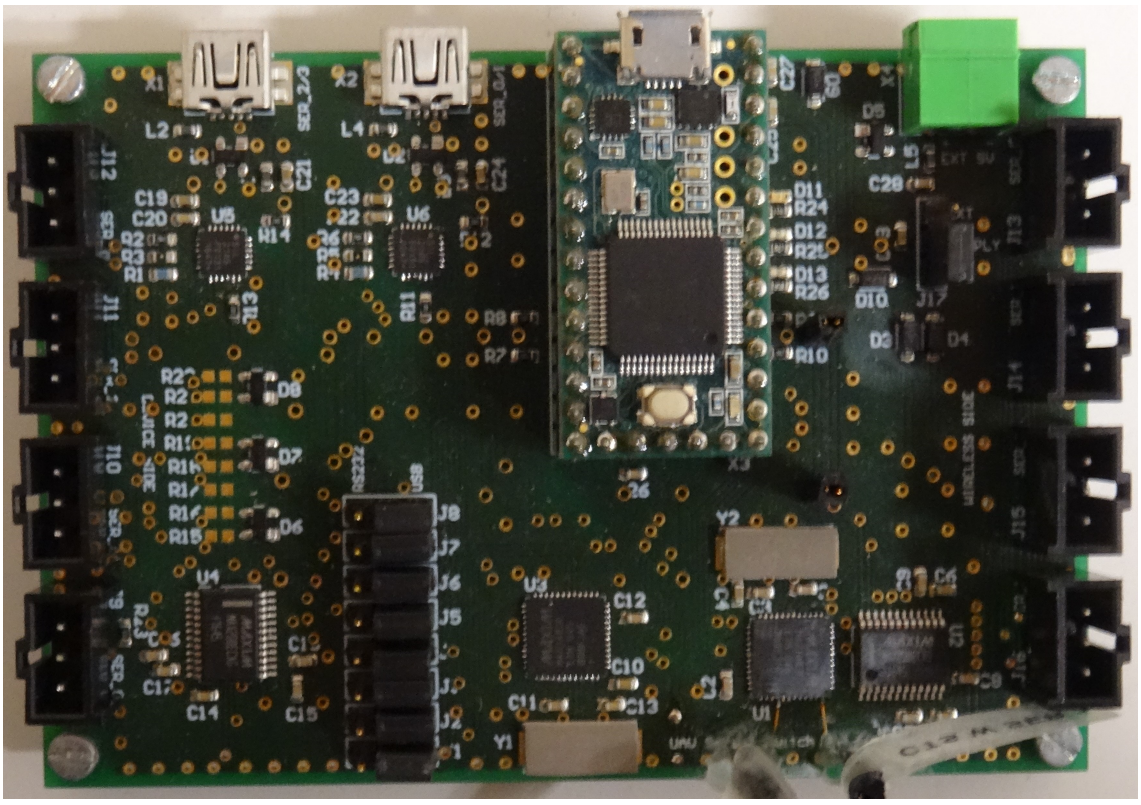

Figure 3.1: Base board with Teensy 3.2 designed by Andreas Albisser

connectors for modems to allow different ways of transmission. An overview can be seen in Figure 3.2.

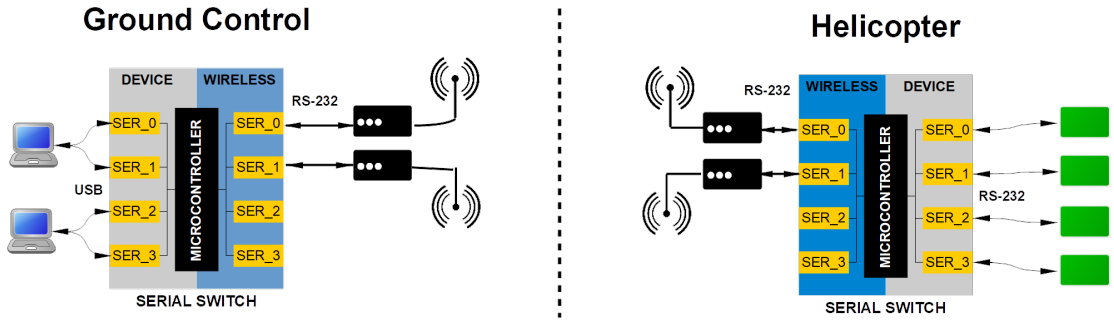

Figure 3.2: Hardware overview

Each interface accessible to the user is bidirectional which means that both sensors and actors can be connected.
From now on, the side where data generating and processing devices can be connected will be referred to as the device side and the side where modems can be connected will be referred to as the wireless side.
On both device side and wireless side, periphery can be connected to the four RS232 serial interfaces. On device side, the user can chose between a RS232 interface and a USB mini interface individually for each interface with jumpers. When selecting the USB mini interface, one USB hub acts as a dual COM interface, allowing two serial COM ports to open up to simulate two serial interfaces.
The serial interfaces are not connected to the Teensy 3.1 development board directly. There is a SPI to UART converter that acts as a hardware buffer between serial input/output and micro controller. All serial connections work on RS232 level which is +-12V. Because the SPI to UART converter is not RS232 level compatible, a voltage regulator is used between the serial interface accessible to the user and the SPI to UART converter.
Details about the components used on this hardware can be taken from the following section. A block diagram of the on-board hardware components can be taken from Figure 3.3.

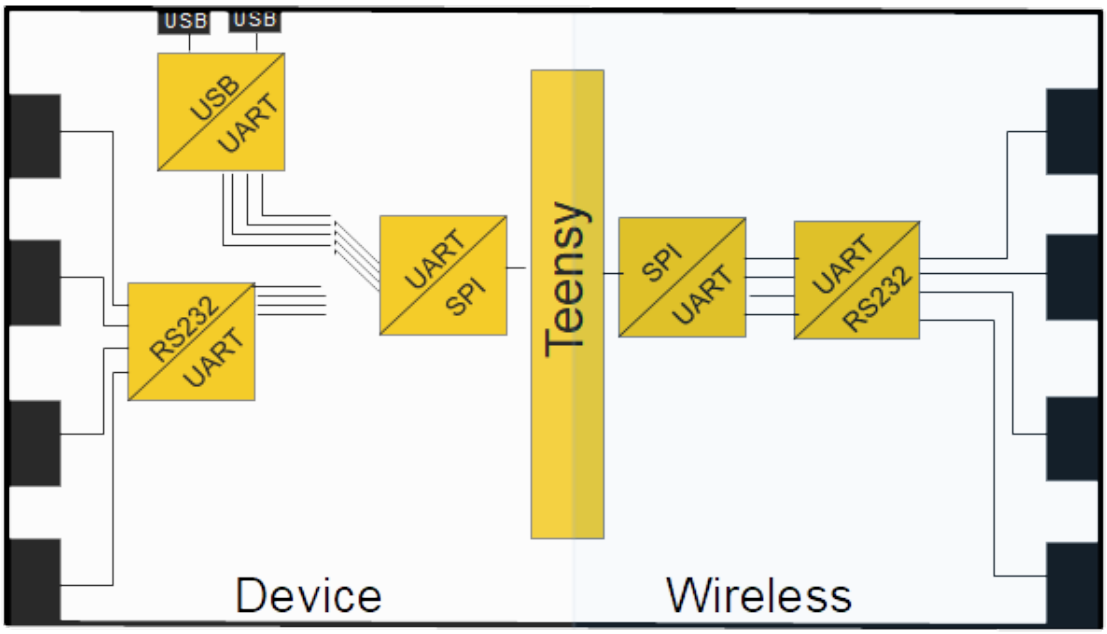

Figure 3.3: Hardware details

3.1.1 User Interface

There are a total of eight RS232 serial connections accessible to the user, four on device side and four on wireless side.
The baud rate for each serial connection can be configured individually.
RS232 is based on the UART protocol but with different voltage level.
UART is an asynchronous serial interface which means that there is no shared clock line between the two components. Both sides need to be configured with the same baud rate so they can communicate correctly.
A UART interface requires three wires: two unidirectional data lines (RX and TX) and a ground connection. Those three wires are accessible to the user on the Serial Switch base board, but with RS232 level, which is +-12V.

3.1.2 RS232 to UART Converter

The serial interfaces accessible to the user work on RS232 level. Just behind the serial interface, there is a level shifter that converts the RS232 level to TTL (5V).
This level shifter is bypassed on the device side in case the USB serial connection is used instead of the RS232 serial interface.

3.1.3 USB to UART Converter

On device side, the user can chose whether data is provided via USB or via RS232 serial connection. A jumper is used to switch between RS232 input and USB input.
In case when the USB input is selected, each USB hub acts as a dual serial COM port which means that when connecting the hardware to a computer, there will be two COM ports available per USB connection.

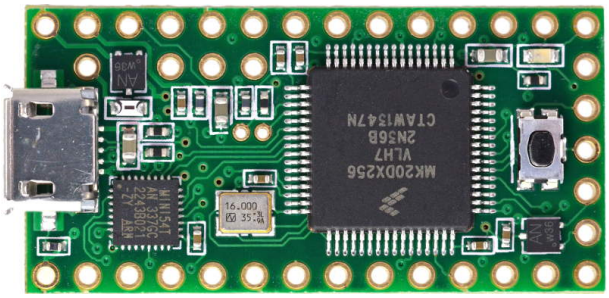

Figure 3.4: Teensy 3.1

The on board USB to UART converter acts as an interface between USB hub and SPI to UART converter.

3.1.4 SPI to UART Converter

UART is an asynchronous serial interface which requires three connections: ground and two unidirectional data lines. If the Teensy was to communicate with each serial port directly, it would require eight of those UART interfaces (which would add up to 16 data lines). To facilitate communication with the serial interfaces, a SPI to UART converter was selected as an intermediate interface.
There are two SPI to UART converters on board, one for the four device serial connections and one for the four wireless serial connections. SPI is a synchronous master-slave communication interface where the unidirectional data lines are shared amongst all participants. The only individual line between master and slave is the Slave Select line that determines, which slave is allowed to communicate to the master at a time.
Those converters are used as hardware buffers and can store up to 128 bytes per serial interface and data direction.

3.1.5 Teensy 32 Development Board

Andreas Albisser used a Teensy 3.2 as a micro controller as can be seen in Figure 3.4.
The Teensy development boards are breadboard compatible USB development boards. They are small, low-priced and yet equipped with a powerful ARM processor.
The Teensy development boards all come with a pre-flashed bootloader to enable programming over USB. They use a less powerful processor as an interface to the developer to enable the use of Arduino libraries and the Arduino IDE.
There is no hardware debugging interface available to the user on the Teensy development boards. Programming is only possible via USB.

3.1.6 Power Supply

The hardware needs 5V as a power supply. This can be achieved by using any of the USB connections or via a dedicated power connector located on the board.

3.2 Software

The following section gives you an overview of the Arduino software written by Andreas Albisser. There was only a brief documentation of the software available but fortunately, the comments in the code were helpful to get an understanding.
Any information provided below has been reverse engineered.

3.2.1 Software Development Tools

The software was written in C++ in Visual Studio. To compile the software, install the Visual Studio Enterprise 2015 version 14, the Arduino IDE extension for Visual Studio and the libraries "Queue by SMFSWänd "TaskScheduler by Anatoli Arkhipenko". Additionally, the old Arduino IDE version 1.8.1 has to be installed as well, together with the software add-on for Teensy support (Teensyduino). A detailed installation manual for all packages and environments needed can be found in the appendix.
.

Link zur Installationsanleitung im Appendix

3.2.2 Basic Functionality

The software written by Andreas Albisser provided a good basis and reference for the software developed in the scope of this project.
The basic functionality provided by his software was the transmission of data packages and acknowledges on wireless side. Generally it can be said that only packages are exchanged over wireless side and bytes are transmitted or received over device side.
The Teensy would frequently poll its SPI to UART hardware buffers for received data. In case the SPI to UART converter had data in its device buffer, the Teensy would read the data in a second SPI command. The read data is then wrapped in a package with header which contained CRC, timestamp and other information and sent out on the wireless side.
The corresponding second hardware would receive this package on its wireless side, extract the payload from it and send the extracted payload out on its device side.
To ensure successful transmission of packages, the concept of acknowledges is applied in the software where the receiver replies with an acknowledge to a successful package reception. A sequence diagram of a successful package transmission can be found in Figure 3.5.
Both Serial Switches continuously do both tasks: poll on device side to generate data packages for sending and poll on wireless side to receive wireless packages and send that payload back out on its device side.

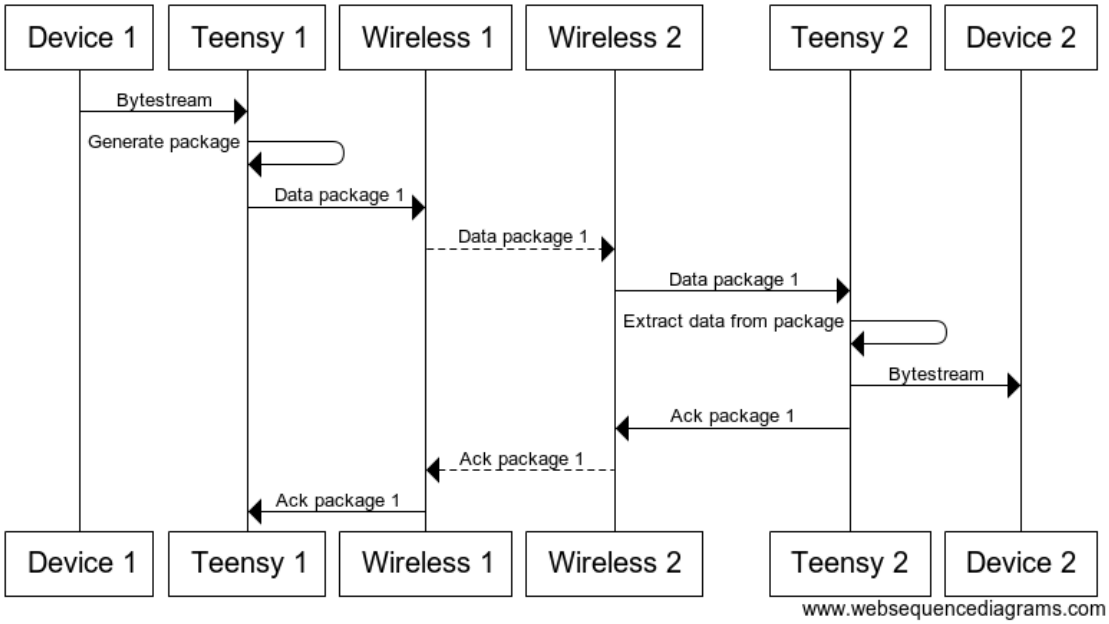

Figure 3.5: Successful package transmission

The maximum number of payload bytes per package can be configured in the software, just like the maximum amount of time the application should wait for a package to fill up until it will be sent anyway.
In case the package transmission was unsuccessful, either if the package got lost or corrupted, the receiving hardware will not send an acknowledge back. The application that sent the package will wait for a configurable amount of time before trying to send the same package again. Details can be found in **??**.
The maximum time to wait for an acknowledge before resending the same package can be configured in the software. The maximum number of resends per package can be configured for each wireless connection.

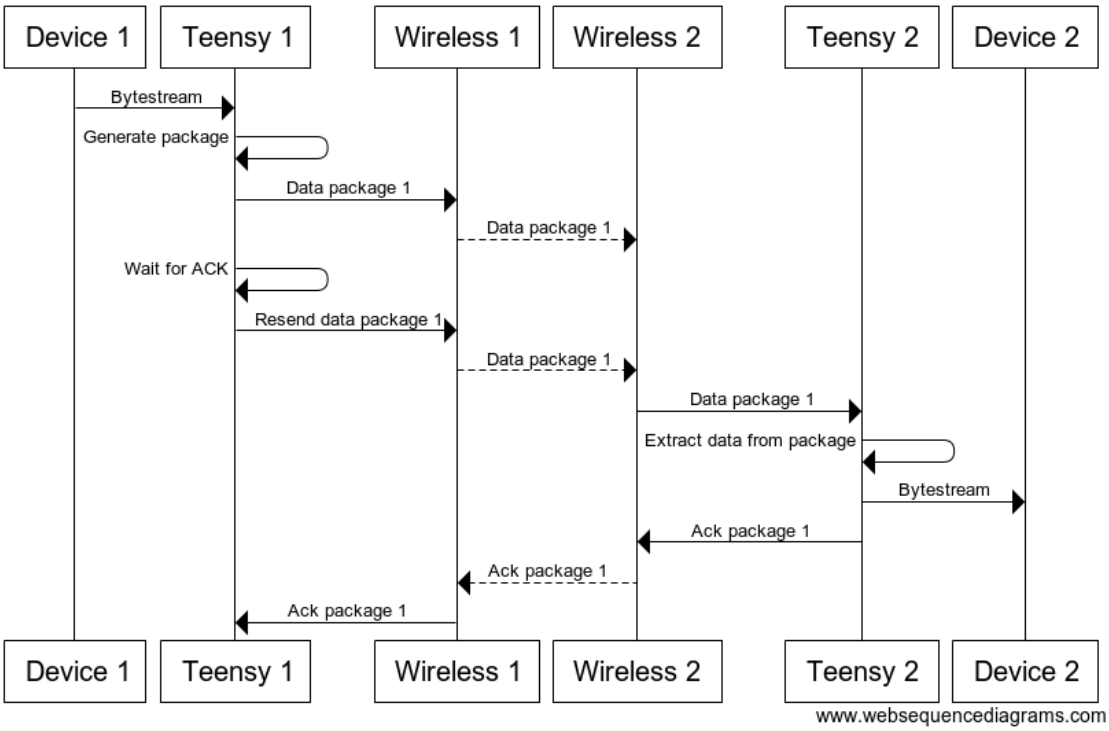

Figure 3.6: Unsuccessful package transmission

3.2.3 Configuration

All basic configuration parameters of the Arduino software are in the file serialSwitch_General.h For changes to be executed, the software has to be recompiled and uploaded. In order to do so, the necessary environment and all packages used by the software have to be installed on the computer as described in the user manual . All configuration possibilities of Andreas Albissers software can be taken from the Table 3.1:

Link zum user manual FW installation von Andreas

Configuration parameter	**Possible values**	**Description**
BAUD_RATES_WIRELESS_CONN	9600, 38400, 57600, 115200	Baud rate to use on wireless side, configurable per wireless connection. Example: 9600, 38400, 57600, 115200 would result in 9600 baud for wireless connection 0, 38400 baud for wireless connection 1 etc.
BAUD_RATES_DEVICE_CONN	9600, 38400, 57600, 115200	Baud rate to use on device side, configurable per cevice connection. Example: 9600, 38400, 57600, 115200 would result in 9600 baud for device connection 0, 38400 baud for device connection 1 etc.

PRIO_WIRELESS_CONN_DEV_X	0, 1, 2, 3, 4	This parameter determines over which wireless connection the data stream of a device will possibly be sent out. 0: this wireless connection will not be used. 1: Highest priority, data will be tried to send out over this connection first. 2: Second highest priority, data will be tried to send out over this connection should transmission over the first priority connection fail. 3: Third highest priority. 4: Lowest priority for data transmission. Example: 0, 2, 1, 0 would result in data being sent out over wireless connection 2 first and only sent out over wireless connection 1 in case of failure. All other wireless connections would not be used. Replace the X in the parameter name with 0, 1, 2 or 3.
SEND_CNT_WIRELESS_ CONN_DEV_X	0...255	Determines how many times a package should tried to be sent out over a wireless connection before moving on to retrying with the next lower priority wireless connection. Example: 0, 5, 4, 0 would result in the package being sent out over wireless connection 1 five times and four times over wireless connection 2. Together with PRIO_WIRELESS_CONN_DEV_X, this parameter determines the number of resends per connection. Replace the X in the parameter name with 0, 1, 2 or 3.
RESEND_DELAY_WIRELESS_ CONN_DEV_X	0...255	Determines how many milliseconds the software should wait for an acknowledge per wireless connection before sending the same package again. Example: 10, 0, 0, 0 would result in the software waiting for an acknowledge for 10ms when having sent a package out via wireless connection 0 before attempting a resend. Together with PRIO_WIRELESS_CONN_DEV_X, this parameter determines the delay of the resend behaviour Replace the X in the parameter name with 0, 1, 2 or 3.

MAX_THROUGHPUT_WIRELESS_CONN	0...4294967295	Limit of the maximum data throughput in bytes/s per wireless connection. If two devices use the same wireless connection with the same priority but the maximum throughput is reached, data of the lower priority device will be redirected to its wireless connection with the next lower priority or discarded (in case this was the wireless connection with lowest priority already). Example: 0, 10000, 10000, 10000 means that wireless connection 0 will not be used.
USUAL_PACKET_SIZE_DEVICE_CONN	0...512	Maximum number of payload bytes per wireless package. 0: unknown payload, the PACKAGE_GEN_MAX_TIMEOUT parameter always determines the payload size. Example: 128, 0, 128, 128 results in a maximum payload of 128 bytes per package and an unknown maximum payload size for wireless connection 0.
PACKAGE_GEN_MAX_TIMEOUT	0...255	Maximum time (in milliseconds) that the software should wait for a package to fill up before sending it out anyway. Together with USUAL_PACKET_SIZE_DEVICE_CONN, this parameter determines the size of a package. Example: 50, 50, 50, 50 will result in data being sent out after a maximum wait time of 50ms.
DELAY_DISMISS_OLD_PACK_PER_DEV	0...255	Maximum time (in milliseconds) an old package should be tried to resend while the next package with data from the same device is available for sending. Example: 5, 5, 5, 5 results in a package being discarded 5ms after the next package is available in case it has not been sent successfully until then.
SEND_ACK_PER_WIRELESS_CONN	0, 1	Acknowledges turned on/off for each wireless connection. Example: 1, 1, 0, 0 results in acknowledges being expected and sent over wireless connection 0 and 1 but not over wireless connection 2 and 3.
USE_CTS_PER_WIRELESS_CONN	0, 1	Hardware flow control turned on/off for each wireless connection. Example: 1, 1, 0, 0 results in hardware flow control (CTS) for wireless connection 0 and 1 only.

Table 3.1: Configuration parameters of arduino software

Further configuration parameters can be found in the file serialSwitch_General.h
There, the you can modify task interval of all tasks, enable hardware loopback and debug output or edit the preamble for a package start.

Configuration parameter	Possible values	Description
TEST_HW_LOOPBACK_ONLY	0, 1	This parameter enables local echo. Any data received over any serial connection will be returned over the same connection immediately.
ENABLE_TEST_DATA_GEN	0, 1	Random test data will be generated instead of waiting for device data to arrive to fill a package.
GENERATE_THROUGHPUT_OUTPUT	0, 1	Information about the data throughput on wireless side will be printed out on the serial terminal.
X_INTERVAL	0 ... 65535	Task interval in milliseconds for each task. Replace X with the name of the task.

Table 3.2: General software configuration

3.2.4 LED Status

There is a separate task that handles blinking of the green LED. While this LED blinks, the software is running and all threads are executed.
The orange LED is turned on when a warning is printed out on the serial interface and the red LED is turned on when an error occurs.

3.2.5 Software Concept

The software written by Andreas Albisser runs with ten main tasks that make up the basic functionality and several minor task that are responsible for debug output, blinking of LEDs and any other functionalities that can be enabled through the configuration header. The software concept can be seen in Figure 3.7.

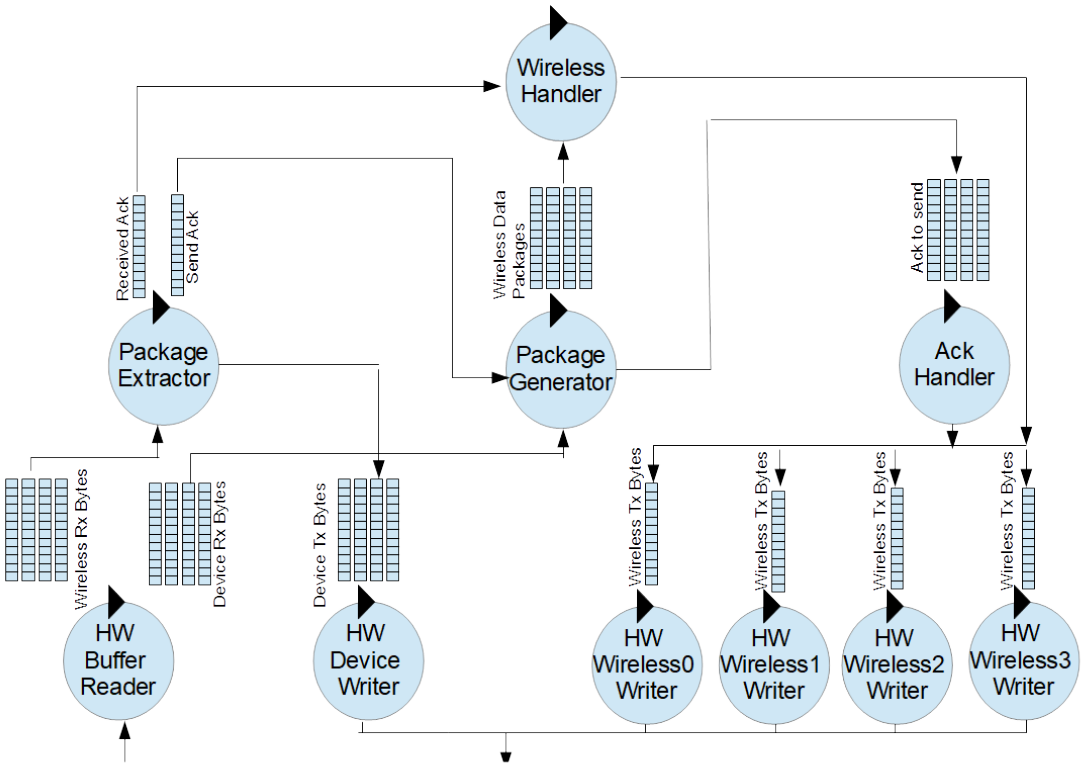

Figure 3.7: Arduino software concept

In the following sections, an overview will be given on the functionality performed by each task.

SPI Interface not in figure of SW concept

HW Buffer Reader

This task periodically polls the SPI to UART converters for new data. In case the converters have received data, the HW Buffer Reader will read and store the data in the corresponding queue.
The HW Buffer Reader does not know anything about packages or any data structure. It simply reads bytes and stores them in a queue.
The HW Buffer reader is responsible for the input data of both SPI to UART converters, the one on device side and on wireless side. This task has eight queues where the read data is stored, one queue for each UART interface accessible to the user.
If there is more data available in the hardware buffer (SPI to UART converter) than can be stored in the corresponding output queue of the HW Buffer Reader, the HW Buffer Reader will flush the queue to discard all information previously read from the SPI to UART converter and store its read data in the now empty queue.

```
1  /* send the read data to the corresponding queue */
2  /*const char* buf = (const char*) &buffer[0]; */
3  for (unsigned int cnt = 0; cnt < dataToRead; cnt++)
4  {
5      if ((*queuePtr).push(&buffer[cnt]) == false)
6      {
7          /* queue is full - flush queue, send character again and set error */
8          (*queuePtr).clean();
9          (*queuePtr).push(&buffer[cnt]);
10         if (spiSlave == MAX_14830_WIRELESS_SIDE)
11         {
12             char warnBuf[128];
13             sprintf(warnBuf, "cleaning full queue on wireless side, UART number %u", (
                   unsigned int)uartNr);
```

```
14              showWarning(__FUNCTION__, warnBuf);
15          }
16          else
17          {
18              /* spiSlave == MAX_14830_DEVICE_SIDE */
19              char warnBuf[128];
20              sprintf(warnBuf, "cleaning full queue on device side, UART number %u", (
                    unsigned int)uartNr);
21              showWarning(__FUNCTION__, warnBuf);
22          }
23      }
24  }
```

HW Device Writer

This task transmits data to the SPI to UART converter on device side. It takes bytes from its queues and passes them to the SPI to UART converter.
Communication to other tasks has been realized through four byte queues, one for each device interface accessible to the user.
This task does not know anything about data packages or other data structures, it only takes bytes from the queues and writes them to the SPI to UART converter on device side.

HW WirelessX Writer

There are four tasks responsible for writing data to the SPI to UART converter on wireless side. Each task has its byte queue where data will be taken from and transmitted to the corresponding wireless user interface.
These tasks do not know anything about data packages or other data structures, they only take bytes from the queues and write them to the SPI to UART converter on wireless side.
Data is only taken from the queue and written to the hardware buffer if there is space available on the hardware buffer.

Package Extractor

This task reads the wireless bytes from the output queue of the HW Buffer Reader and assembles them to wireless packages.
There are two types of wireless packages, acknowledges and data packages. The Package Extractor detects of which type an assembled package is and puts it on the corresponding queue.
This task also verifies the checksums of both header and payload of a package and discards the package in case of incorrect checksum.
In case of full output queues, new packages will be dropped and not stored in the corresponding queue.

Package Generator

This task reads the incoming device bytes from the output queue of the HW Buffer Reader and generates data packages with this device data as payload. The generated packages are then stored in the corresponding queue for further processing.
The Package Generator also generates acknowledge packages when being told so by the output queue of the Package Extractor. The generated acknowledge are then put into the correct queue for a wireless connection.
If the queue is full, the package is dropped, no matter if acknowledge package or data package.

```
/* check if there are some receive acknowledge packages that needs to be created */
while (queueSendAck.pull(&ackData))
{
    if (generateRecAckPackage(&ackData, &wirelessAckPackage))
    {
        queueWirelessAckPack.push(&wirelessAckPackage);
    }
}

/* generate data packages and put those into the package queue */
if (generateDataPackage(0, &queueDeviceReceiveUart[0], &wirelessPackage))
{
    queueWirelessDataPackPerDev[0].push(&wirelessPackage);
}
if (generateDataPackage(1, &queueDeviceReceiveUart[1], &wirelessPackage))
{
    queueWirelessDataPackPerDev[1].push(&wirelessPackage);
}
if (generateDataPackage(2, &queueDeviceReceiveUart[2], &wirelessPackage))
{
    queueWirelessDataPackPerDev[2].push(&wirelessPackage);
}
if (generateDataPackage(3, &queueDeviceReceiveUart[3], &wirelessPackage))
{
    queueWirelessDataPackPerDev[3].push(&wirelessPackage);
}
```

The queue function call for push() returns false if unsuccessful there is no handling of an unsuccessful push in this code.

Ack Handler

This task takes the acknowledge package generated by the Package Generator, splits it into bytes and puts those bytes into the corresponding wireless queue for the HW WirelessX Writer to send out.

Wireless Handler

This task handles the correct sending of wireless packages. It takes wireless packages from the output queues of the Package Generator, splits them into bytes and puts those bytes to the queue of the correct HW WirelessX Writer.
This task has an internal buffer where packages with an expected acknowledge are stored. The Wireless Handler keeps track of the send attempts per wireless connection and handles the resending of packages.
The Wireless Handler also does the prioritizing of data packages.

3.2.6 Wireless Package Structure

There are two types of packages that are exchanged over wireless side: acknowledges and data packages. Each package consists of a header and a payload of arbitrary size. Acknowledges and data packages can be distinguished by a parameter in the header of a package.
More information about the structure of header and payload can be found in the following section.

Header

The structure of a header can be found in Table 3.3.

Parameter name	Description	Value range	Length, bytes
PACK_START	Preamble for a package header start, indicates the beginning of a header.	0x1B	1
PACK_TYPE	Determines weather it is a data package or acknowledge.	1: data pack, 2: acknowledge	1
SESSION_NR	A random number generated upon startup of the software to keep the receiver from discarding all packages in case a reset has been made.	0...255	1
SYS_TIME	Milliseconds since start of the software. This parameter is used as a substitute for package numbering.	0...4294967295	4
PAYLOAD_SIZE	Number of bytes in payload of this package	0...65535	2
CRC8_HEADER	8 bit CRC of this header	0...255	1

Table 3.3: Header structure

When the software receives a package, it checks the system time to decide if it should be discarded. If the system time of a received package is lower than the last one received, it will be discarded.
In case of a hardware reset, the system time starts over with 0 which means that all packages would be discarded on receiving side. This is the reason for the session number. When the session number changes, the receiver knows to start over with the system time and not to discard all received packages because of a lower system time.

Payload

Talk about payload and CRC32

Parameter name	Description	Value range	Length, bytes
PAYLOAD	Data bytes to send out	0...255	0...65535
CRC16_PAYLOAD	16 bit CRC of the payload	0...65535	2

Table 3.4: Payload structure

3.3 Discussion and Problems

There was no documentation available on tests conducted nor on issues discovered with Andreas Albissers work.
All information below has been recalled by the people involved.

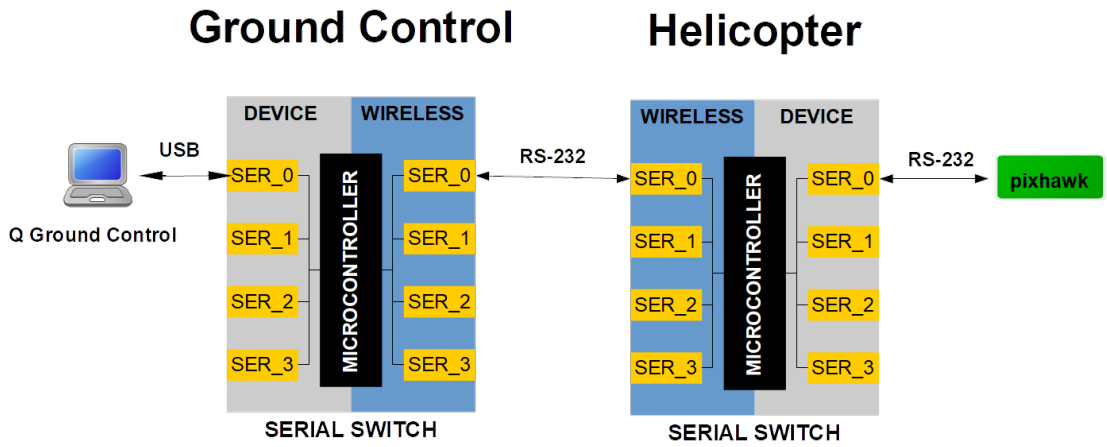

Figure 3.8: End test with hardware directly connected

3.3.1 Tests Results

Aeroscout uses a Pixhawk PX4 as an autopilot on-board. This is an open-source and flexible platform for UAV projects. The PX4 can be controlled with QGroundControl, an open source software that runs off-board on a computer and provides full flight control and vehicle setup.
As an end test, the PX4 and QGroundControl were connected to the Serial Switch to check if data transmission between those two components works as expected.
First, the QGroundControl needs to be configured with the correct COM port and baud rate. When connecting on the COM port for the first time, QGroundControl will try to establish a link to the PX4 and gather all data available from the on-board system. Only then will the autopilot show up as a valid link on the computer.
This is also a stress test for the software to check for memory leaks because QGroundControl and PX4 exchanges lots of data upon first setup (about 3000 bytes/sec from PX4 to computer and about 100 bytes/sec from computer to PX4, for about 1sec).
Afterwards, they will only transmit about 150 bytes/sec from PX4 to computer and 20bytes/sec from computer to PX4 during idle mode.
These numbers are best case conditions, with no resending of packages and no acknowledge configured.
Source and verification of this throughput rate can be found in Section 6.2.4.

Directly connected wireless sides

To test the software under semi real circumstances, the PX4 and QGroundControl is connected to the serial switches.
The setup can be seen in Figure 3.8.
The PX4 was connected on device side of the on-board Serial Switch with UART. QGroundControl was running on a computer that was connected to the off-board Serial Switch on device side via USB. The two Serial Switches were directly connected on wireless side with RX and TX crossed.
After setting up the correct baud rates for device side on both Serial Switches (57600 baud) and disabling acknowledges, the data link between QGroundControl and the PX4 was established successfully within about one second.
The outcome was also successful when acknowledges were configured.

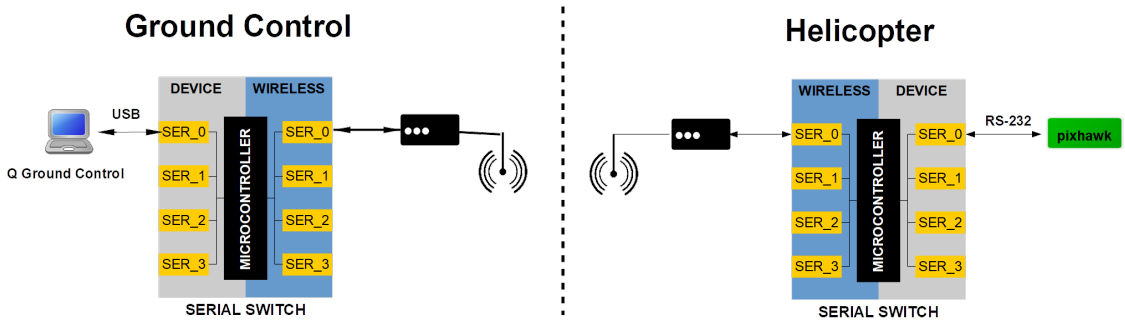

Figure 3.9: End test with wireless communication

Wireless sides with modem

A real testcase was provided when the same configuration as above was tested but the Serial Switches were not connected directly but through a modem and antenna. The setup was as can be seen in Figure 3.9 This time, no data link could be established between PX4 and QGroundControl. One possible reason for this faulty behavior could be the lack of package numbering. The header of a package only contains session number and system time but no variable with a monotonic increasing counter. The system time does not provide any information about missing packages because it might jump up in case no package was generated for some amount of time. The parameter system time corresponds to the runtime of the software since start up in milliseconds.
The following code section is copied from the software Andreas Albisser developed. It shows that packages will be discarded when their time stamp (sysTime) is older than the one of the last package.

```
1  /* make sure to not send old data to the device - but also make sure overflow is handled
       */
2  if ((currentWirelessPackage[wirelessConnNr].sysTime > timestampLastValidPackage[
       currentWirelessPackage[wirelessConnNr].devNum]) ||
3  ((timestampLastValidPackage[currentWirelessPackage[wirelessConnNr].devNum] -
       currentWirelessPackage[wirelessConnNr].sysTime) > (UINT32_MAX / 2)))
4  {
5      /* package OK, send it to device */
6      timestampLastValidPackage[currentWirelessPackage[wirelessConnNr].devNum] =
           currentWirelessPackage[wirelessConnNr].sysTime;
7      Queue* sendQueue;
8      sendQueue = &queueDeviceTransmitUart[currentWirelessPackage[wirelessConnNr].devNum];
9      for (uint16_t cnt = 0; cnt < currentWirelessPackage[wirelessConnNr].payloadSize; cnt
           ++)
10     {
11         if (sendQueue->push(&data[wirelessConnNr][cnt]) == false)
12         {
13             char warnBuf[128];
14             sprintf(warnBuf, "Unable to send data to device number %u, queue is full",
                   currentWirelessPackage[wirelessConnNr].devNum);
15             showWarning(__FUNCTION__, warnBuf);
16             break;
17         }
18     }
19 }
20 else
21 {
22     /* received old package */
23     /* also can happen when we have two redundant streams.. */
24     static char infoBuf[128];
25     sprintf(infoBuf,
26     "received old package, device %u - check configuration if this message occurres often"
           ,
27     currentWirelessPackage[wirelessConnNr].devNum);
28     showInfo(__FUNCTION__, infoBuf);
29 }
```

This theory has not been tested because the software Andreas Albisser has writte has not been tested or developped further in the scope of this project.

3.3.2 Issues

Dropping Data when Transmission unreliable

Task intercommunication has been done with queues, as can be seen in Figure 3.7. When data arrives, it will be pushed to the byte queue from one task to be processed and assembled to a full package by an other task.
When data arrives too fast, any of the queues may be full and pushing the most recent data to the queue might fail. Currently, only the hardware interfaces (HW Device Writer and HW WirelessX Writer) deal with this scenario. When they read data from the SPI to UART converter but can not push it to the RX byte queue, they will flush the byte queue (which results in all unprocessed data bytes being lost) and push the new data to the now empty queue. A warning will be printed on the serial interface afterwards and the orange LED will be turned on.
A snippet from the code handling a full byte queue can be seen below.

```
1 if ((*queuePtr).push(&buffer[cnt]) == false)
2 {
3     /* queue is full - flush queue, send character again and set error */
4     (*queuePtr).clean();
5     (*queuePtr).push(&buffer[cnt]);
```

When any of the other queues are full or pushing new data to a queue is not possible, this data will be lost and no further action will be taken except for printing a warning on the serial interface.
This results in packages being lost when lots of data is read and packages generated on device side. The package generator does not check if its package output queue is full before popping bytes from the received byte queue and putting them into the payload of a generated package. The generated package will be lost together with all its data when trying to push it onto a full queue.
All tasks should take the state of their interface queues into account during runtime so that dropping of data can be carried out controlled and purposely.

Debugging

Arduino does not support hardware debugging, only prints on a serial connection are available.
This is fine as long as the software is relatively small and straightforward.
As software complexity increases it becomes problematic, especially when multiple tasks are involved.
As can be seen in Figure 3.7, the software concept implemented by Andreas Albisser is complex and runs with many tasks. In order to expand the functionality of it even further, a real hardware debugging interface is inevitable.

Software Concept

The software concept implemented by Andreas Albisser as can be seen in Figure 3.7 and it was attempted to visualize it split into three layers similar to the ISO/OSI model.

All hardware reader and writer tasks access the SPI to UART interface and represent layer 1. They deal with bytes and do not know anything about packages.
Each wireless interface runs with its own task that handles the bytes to be sent out. There is one task that writes bytes to the SPI to UART converter on device side and one task that reads incoming bytes from both SPI to UART converters. These five tasks all access the same SPI interface which results in possible conflicts and the need to define critical sections. The software concept would be simpler if there was only one task that accessed the SPI interface.
The package generator, package extractor and ack handler tasks are the interface between bytes and packages and represent layer 2. They assemble wireless packages popped from the output queue of layer 1 and split generated packages into bytes. They also deal with the sending of acknowledges in case a received package expects one.
The package handlers have two separate queues where assembled data packages and acknowledges are stored. The wireless handler will then not process packages in the order they were received but will first iterate through the data package queue and then through the acknowledge queue.
The wireless handler represents layer 3 and deals with sending and resending of packages. While it keeps track of acknowledges received, it does not handle the acknowledges sent because this is contradictory done by layer 2.
Generally, the software could be kept much smaller with less tasks and less queues.

Data Priority

There is no way to prioritize data of one device over data of an other device connected. When data transmission becomes unreliable, the software needs to know which data is most important to be exchanged. An additional configuration parameter should be added to represent data priority.

Reassembling Device Data from Received Packages

The receiver will check the system time of the package header to decide if it will extract the payload to send it out on device side or discard the package. If the system time of the package received is lower than the system time of the last package received, it will discard the package completely.
This results missing data in case packages are not received in the right order even though data transmission might still be reliable.

4 Hardware

The task description for this project as seen in appendix B requires the following hardware changes:

- Optimization of size and weight
- Optimization for outdoor use
- Usage of more powerful processor with more memory and RNG or encryption support
- SWD/JTAG debugging interface
- UART hardware flow control
- On-board SD card (regular or micro)

There are several options on how to proceed with the implementation. The pro and cons of these choices are listed in this chapter, followed by the chosen solution and its execution.

4.1 Hardware Redesign Options

The hardware Andreas Albisser designed is working as expected. The desired modifications as listed in the task description are merely optimizations. Only the replacement of the Teensy 3.1 is absolutely required for this project because software will later be written for a specific micro controller.
Therefore there are two options on how to proceed.

- Redesign entire hardware for/with a new processor.
- Redesign hardware step by step, starting with just an adapter board to use existing hardware with new micro controller.

4.1.1 Complete Hardware Redesign

A redesign of the entire hardware requires careful component selection, adaption of the schematic and footprints and redesign of the PCB.
Changes for the next complete hardware redesign include:

- New RS232 level to TTL level converter with more inputs to convert hardware flow control pins (CTS/RTS) as well. The converter currently used has no more free pins available. Alternatively add more components of the already used converter for the hardware flow control signals.
- A way to switch between RS232 level and TTL level for all serial interfaces accessible to the user to allow for TTL modems to be connected as well.
- More powerful micro controller with support for encryption and SD card slot.
- Hardware debugging interface.
- Use of different connector as user interface, with more pins to include hardware flow control signals.

Implementing all these changes would require about three weeks, followed by one week of manufacturing and one week of assembly. In case of a faulty hardware, it would be extremely difficult start

with the software implementation because there is no hardware to work with and no way to work standalone. In this case, producing a second version of the hardware would take up a considerable amount of time because of manufacturing time, assembly and testing.
There is simply not enough time to redesign the entire hardware in the scope of this project.
A possible project plan for this scenario can be found in Figure 4.1:

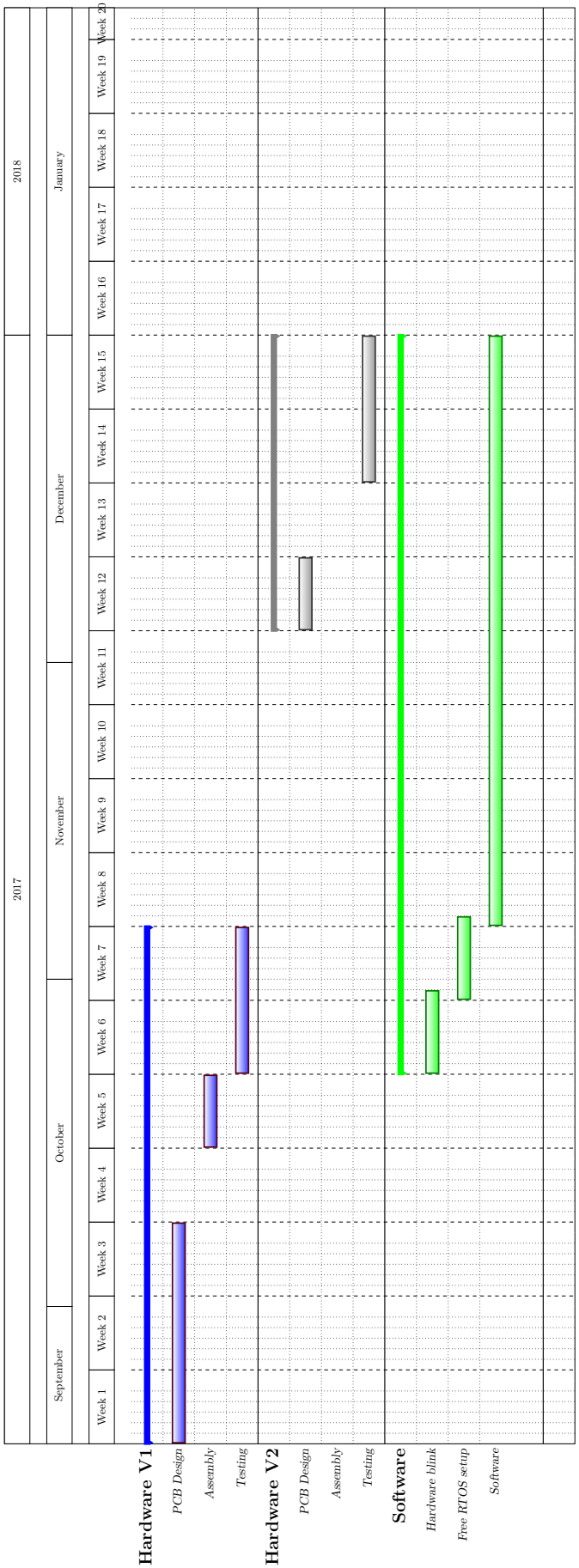

Figure 4.1: Possible project plan of a complete hardware redesign

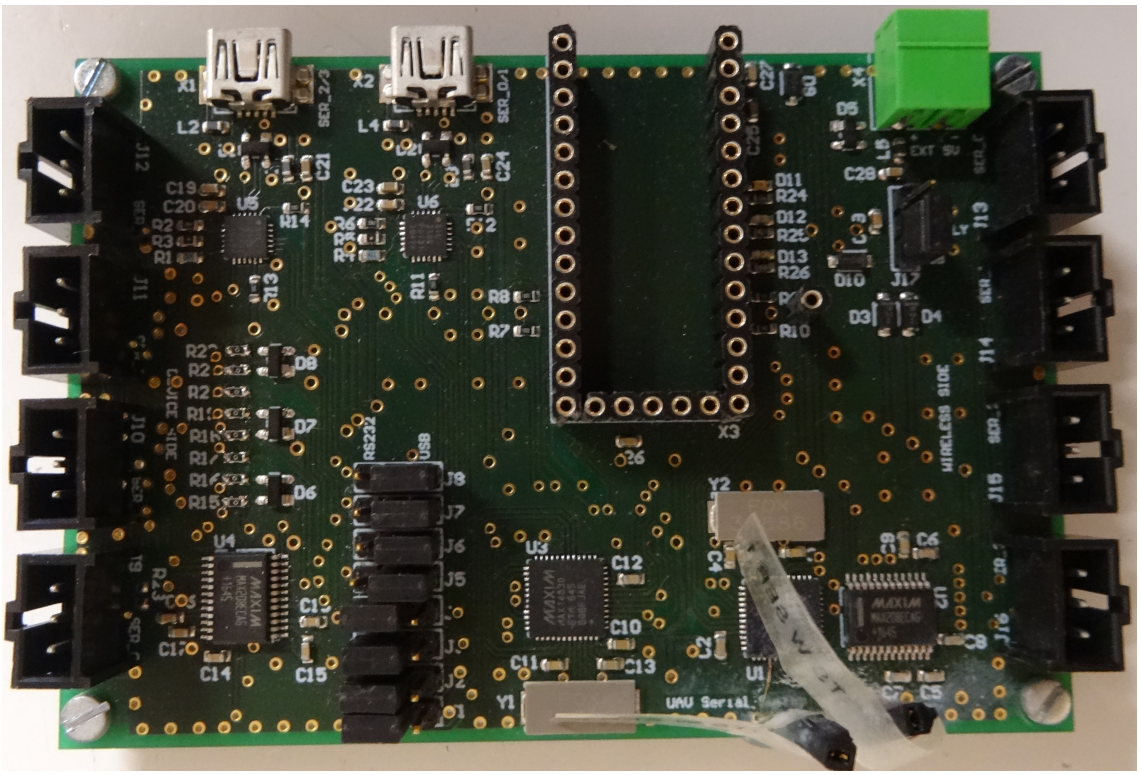

Figure 4.2: Base board with female header, Teensy 3.1 not plugged in

4.1.2 Adapter Board

The most profound hardware change is the replacement of the Teensy 3.1.
First, a new development board or micro controller has to be selected that supports hardware debugging and meets the requirements for data encryption.
After selecting a replacement for the Teensy 3.1, the fastest way to get started with software development for the new micro controller is by designing an adapter board with a Teensy 3.1 footprint.
The Teensy 3.1 ships with headers that can be soldered onto the development board. Andreas Albisser designed the interface for the Teensy with female header pins so the development board could simply be plugged into the header and exchanged if needed. This makes it easy to design an adapter print with male headers that match the footprint of the previously used Teensy 3.1.
The base board with female headers can be seen in Figure 4.2. In this picture, the Teensy 3.1 is not plugged in, the empty headers can be used to route the pins of an other microcontroller to control the base board.
This solution has been chosen in the scope of this work to ensure that the end result of this project would provide solid ground work for further development.

4.2 Component Evaluation

Before starting with the design of an adapter board, a replacement for the Teensy 3.1 development board has to be chosen.

4.2.1 Development Board Selection

The easiest option is to select a more powerful Teensy development board that meets the requirements listed in appenedix B.
Fortunately, both the Teensy 3.5 and Teensy 3.6 meet the requirements and have an on-board SD card slot. A comparison between the Teensies can be found in Table 4.1 The pins of both Teensy 3.5 and

	Teensy 3.1	**Teensy 3.5**	**Teensy 3.6**
Processor	MK20DX256 32 bit ARM Cortex-M4 72 MHz	MK64FX512VMD12 Cortex-M4F 120 MHz	MK66FX1M0VMD18 Cortex-M4F 180 MHz
Flash Memory [bytes]	262 k	512 k	1024 k
RAM Memory [bytes]	65 k	196 k	256 k
EEPROM [bytes]	2048	4096	4096
I/O	34, 3.3V, 5V tol	58, 3.3V, 5V tol	58, 3.3V, 5V tol
Analog In	21	27	25
PWM	12	17	19
UART,I2C,SPI	3	6	6
SD Card	no	yes	yes
Price	$19.80	$25.25	$29.25

Table 4.1: Teensy comparison

3.6 are backwards compatible to the pin out of Teensy 3.1 which will make it easier to develop the PCB of an adapter board.
The Teensy 3.5 and Teensy 3.6 development board have all pins needed for SWD hardware debugging available as pads on their backside.
The Teensy 3.5 was chosen for this application because there is more support available for this component and an already configured FreeRTOS. This is not the case with the Teensy 3.6.

4.2.2 Preparation for Hardware Debugging

The Teensy development boards are meant for USB programming and debugging. They are equipped with a small micro controller that acts as a boot loader. The small micro controller is in control of the hardware debugging and reset pins of the main micro controller and does the programming of the main micro controller. This way, all Teensies can be used with standard Arduino libraries and programmed with the Arduino IDE.
The schematic of the Teensy 3.5 can be seen in Figure 4.3. The MKL02Z32VFG4 acts as the boot loader and the MK64FX512 is the main micro controller.
.
The pins available to the user are shown in Figure 4.4 and Figure 4.5.

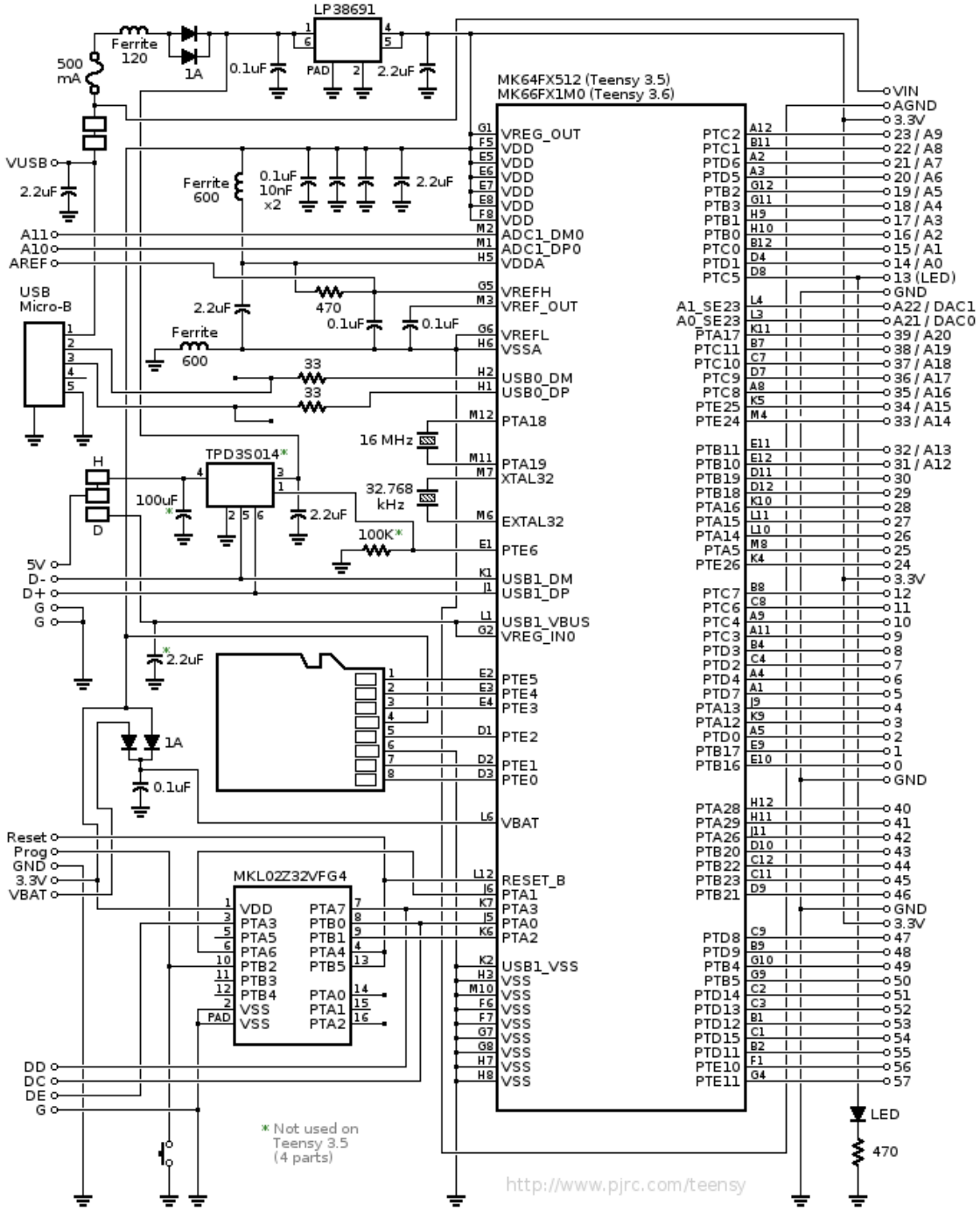

Figure 4.3: Schematic Teensy 3.5

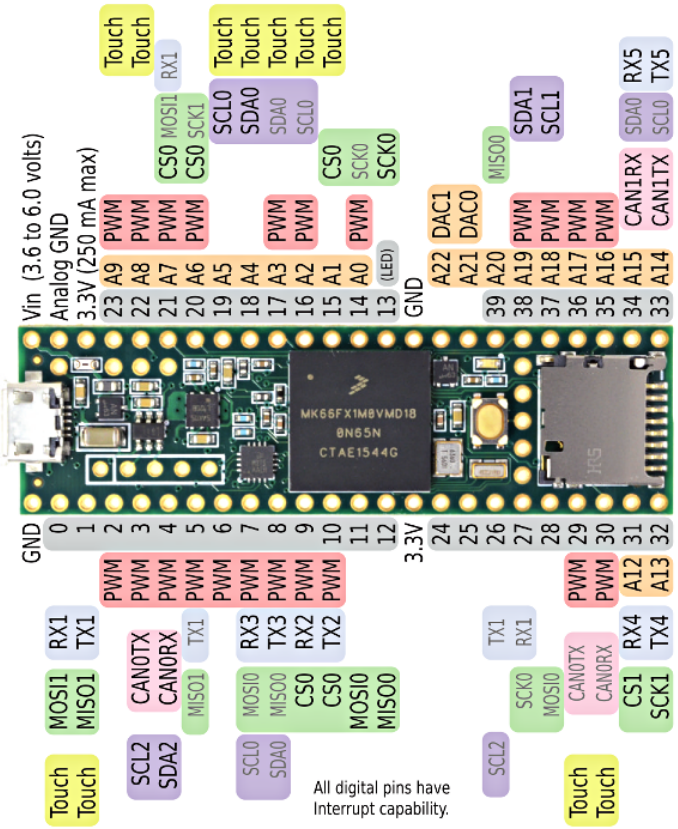

Figure 4.4: Pin assignment, front side

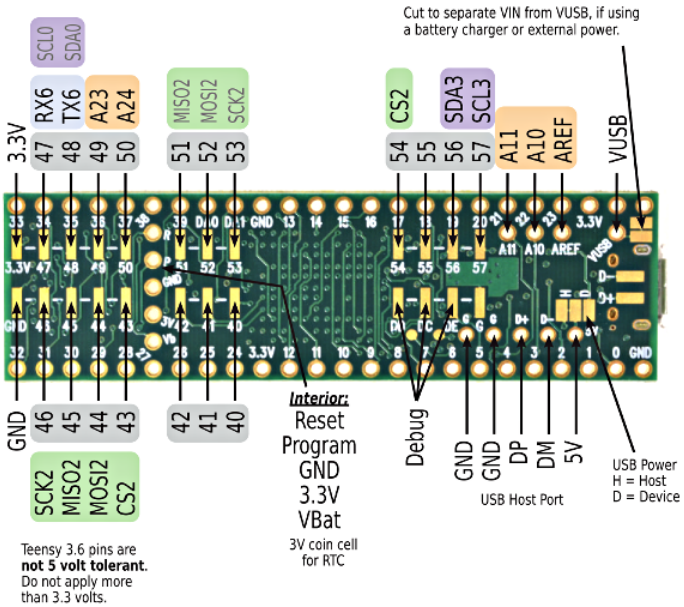

Figure 4.5: Pin assignment, back side

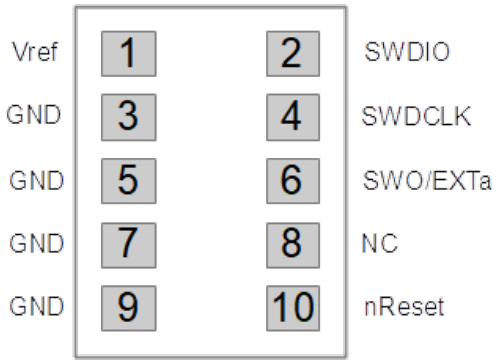

Figure 4.6: SWD pinout

Serial Wire Debug (SWD)

As can be seen in Figure 4.5, there are SWD (Serial Wire Debug) pins are available as pads on the back side of the Teensy 3.5.
The SWD interface consists of the following pins:

- Vref: Supply voltage
- GND: Ground
- SWDIO/DD: Debug Data
- SWDCLK/DC: Debug Clock
- DE: Debug Enable
- RST: Reset

The physical pinout of a SWD interface can be seen in Figure 4.6. Only the reset, data, clock and ground pins are absolutely required to be connected for the debugging interface to work correctly.
Both Teensy 3.5 and Teensy 3.6 have the required SWD pins available on their back side but the debug interface is controlled by the on-board boot loader.
There are two ways to communicate to the main micro controller directly without the boot loader interfering on the hardware debugging interface:

- Holding the boot loader in reset mode.
- Removing the boot loader completely.

Resetting the Boot Loader

According to the data sheet of the boot loader (see Figure 4.7), pin 15 of the boot loader can have one of three functions:

- Reset
- GPIO input
- GPIO output

As default, the pin will be configured as a reset pin, but this function can be turned off by configuring it for any of the other two functions in software.
Even though the Teensies are fully open source, the software for the boot loader is not available. The only way to find out if the reset pin is still configured as such is by pulling it low and attempting a reset.

32 QFN	24 QFN	16 QFN	Pin Name	Default	ALT0	ALT1	ALT2	ALT3
16	12	8	PTB0/ IRQ_5	ADC0_SE6	ADC0_SE6	PTB0/ IRQ_5	EXTRG_IN	SPI0_SCK
17	13	9	PTB1/ IRQ_6	ADC0_SE5/ CMP0_IN3	ADC0_SE5/ CMP0_IN3	PTB1/ IRQ_6	UART0_TX	UART0_RX
18	14	10	PTB2/ IRQ_7	ADC0_SE4	ADC0_SE4	PTB2/ IRQ_7	UART0_RX	UART0_TX
19	15	—	PTA8	ADC0_SE3	ADC0_SE3	PTA8	I2C1_SCL	
20	16	—	PTA9	ADC0_SE2	ADC0_SE2	PTA9	I2C1_SDA	
21	—	—	PTA10/ IRQ_8	DISABLED		PTA10/ IRQ_8		
22	—	—	PTA11/ IRQ_9	DISABLED		PTA11/ IRQ_9		
23	17	11	PTB3/ IRQ_10	DISABLED		PTB3/ IRQ_10	I2C0_SCL	UART0_TX
24	18	12	PTB4/ IRQ_11	DISABLED		PTB4/ IRQ_11	I2C0_SDA	UART0_RX
25	19	13	PTB5/ IRQ_12	NMI_b	ADC0_SE1/ CMP0_IN1	PTB5/ IRQ_12	TPM1_CH1	NMI_b
26	20	—	PTA12/ IRQ_13/ LPTMR0_ALT2	ADC0_SE0/ CMP0_IN0	ADC0_SE0/ CMP0_IN0	PTA12/ IRQ_13/ LPTMR0_ALT2	TPM1_CH0	TPM_CLKIN0
27	—	—	PTA13	DISABLED		PTA13		
28	—	—	PTB12	DISABLED		PTB12		
29	21	—	PTB13	ADC0_SE13	ADC0_SE13	PTB13	TPM1_CH1	
30	22	14	PTA0/ IRQ_0	SWD_CLK	ADC0_SE12/ CMP0_IN2	PTA0/ IRQ_0	TPM1_CH0	SWD_CLK
31	23	15	PTA1/ IRQ_1/ LPTMR0_ALT1	RESET_b		PTA1/ IRQ_1/ LPTMR0_ALT1	TPM_CLKIN0	RESET_b
32	24	16	PTA2	SWD_DIO		PTA2	CMP0_OUT	SWD_DIO

Figure 4.7: Pin out MKL02Z32VFG4

The location of the boot loader can be found in Figure 4.10.
Before putting the boot loader into reset mode, any other functions that this micro controller may be responsible for need to be ensured.
Because the internal pull ups of the boot loader are used for the reset line of the main micro controller, this reset line needs to be pulled up externally first.
A resistor can be soldered onto the Teensy directly for this purpose as seen in Figure 4.8. Afterwards, pin 15 of the boot loader can be pulled low.
Unfortunately, pin 15 seems to be configured as a GPIO pin because pulling the boot loaders reset line low does not prevent it from communicating to the main micro controller of the Teensy 3.5.
The state of the hardware debugging pins during idle state were checked with the scope but as can be seen from Figure 4.9, the boot loader is still in control of the debugging interface and therefore the main micro controller.
Instead of investigating further into this option, the second option was chosen.

Removing the Boot Loader

The MKL02Z32VFG4 has two functions:

– It acts as a boot loader and controls the SWD interface to the main micro controller.

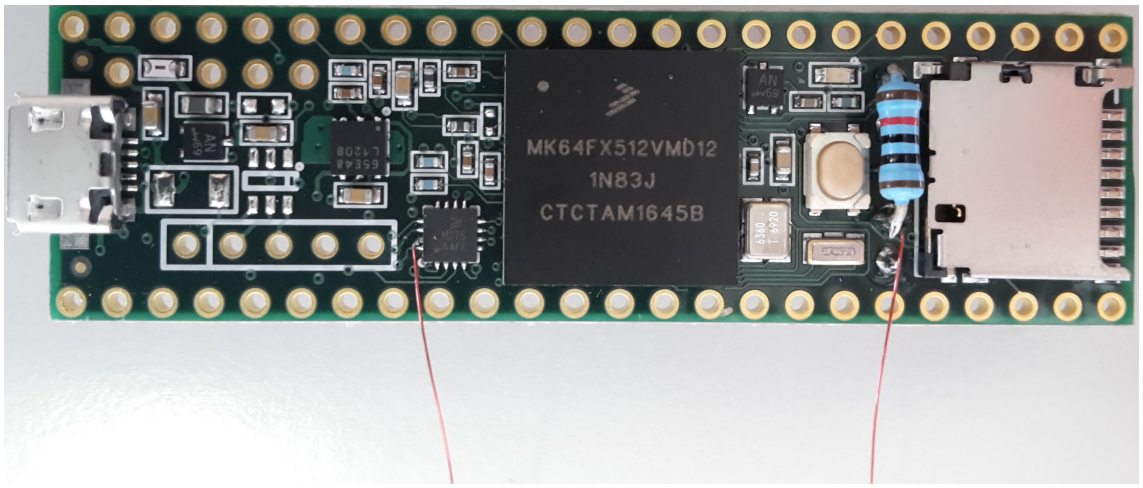

Figure 4.8: Trying to pull the boot loader into reset mode

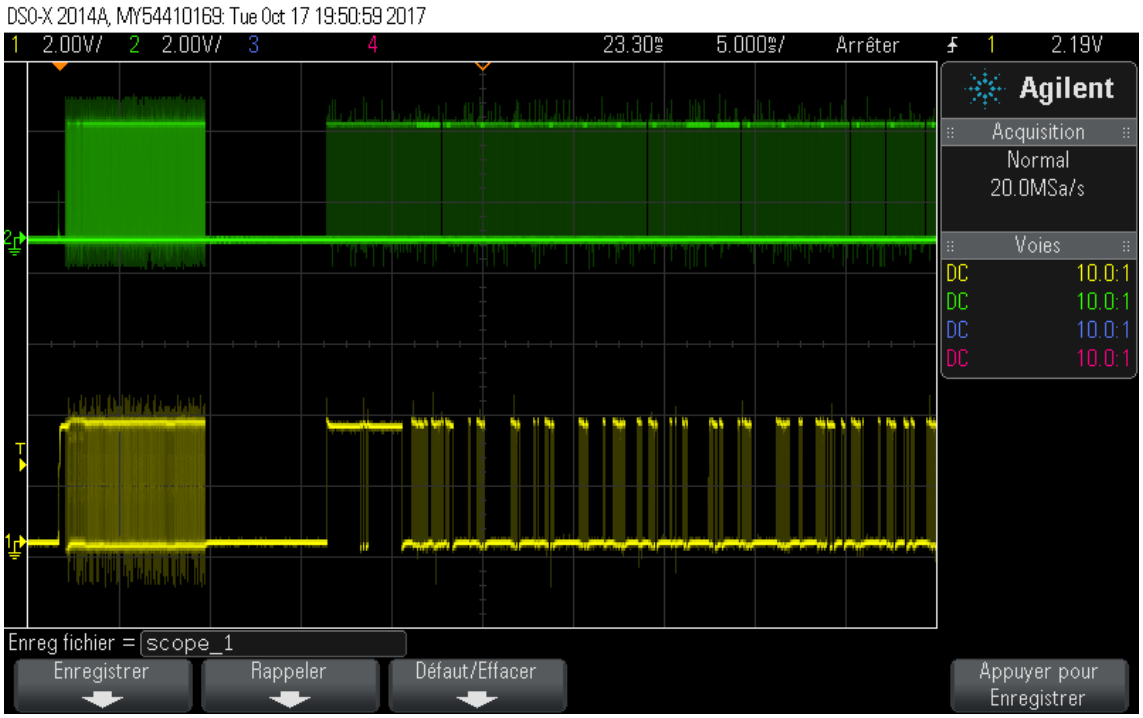

Figure 4.9: The boot loader keeps communicating to the Teensy

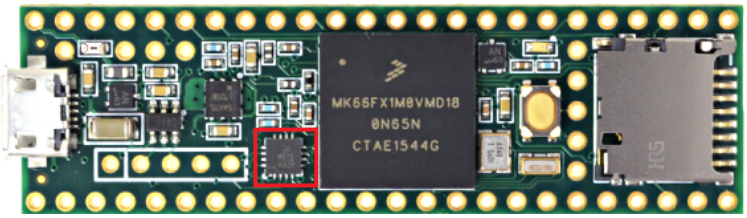

Figure 4.10: Location of the bootloader on Teensy 3.5

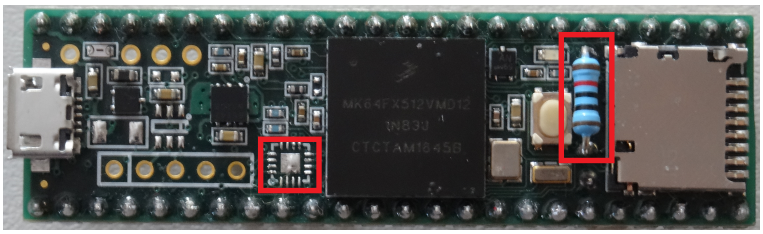

Figure 4.11: Teensy 3.5 modified for hardware debugging

– It controls the reset line of the main micro controller and its internal pull ups are used to keep the reset line in idle state.

To leave the user in full control of the SWD hardware debugging interface, the boot loader has to be removed (or silenced, as attempted in 4.2.2).
The MKL02Z32VFG4 is located on the front side of the Teensy, as indicated in Figure 4.10.
Flux gel was applied around the boot loader before heating the soldering pads up with hot air to remove the MKL02Z32VFG4. Now a pull up resister was added as seen in Figure 4.11. Afterwards, the SWD debugging interface could be used.

4.3 Teensy Adapter Board

The design of the adapter board from the footprint of the Teensy 3.1 to the footprint of the Teensy 3.5 was straight forward because of backwards compatibility of the pinout.
The Teensy 3.5 is slightly longer than the Teensy 3.1 so the extra pins of the newer version will not be routed down to the main board. The backwards compatible pins can be used like before.
Additional components for the adapter board include:

– Hardware debugging interface.
– Pull up resistor for reset line
– Ground header
– 3.3V header

To prepare the Teensy 3.5 for usage within this project, male headers have to be soldered onto the board and the boot loader has to be removed as in 4.2.2. Because the reset line will be pulled up to 3.3V on the adapter board, the on-board pull up resistor (as seen in Figure 4.11) for the reset line is optional. It is only required when working with the Teensy 3.5 stand-alone without the adapter board.
Schematic and PCB of the Teensy adapter board can be found in the appendix .
Issues encountered during development of the adapter board were the pin numbering and interlayer connections.

Link zum Schema+PCB vom Adapter Board im Appendix

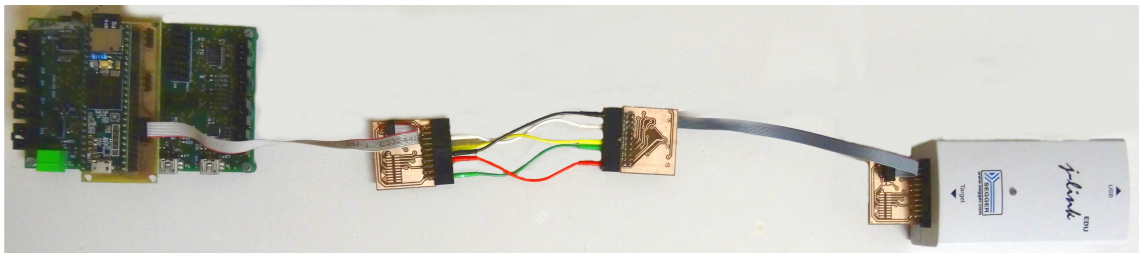

Figure 4.12: Hardware debugging with faulty SWD footprint

Pin numbering

In a first version of the Teensy adapter board, the pin numbering of the SWD debugging connector was done wrong and had to be adjusted in the footprint for a next PCB version.
But instead of waiting for the next PCB version to be produced, all debugging signals were routed manually from the faulty SWD pinout to the debugger. This way, development of the software could be started without delay. A picture of the signal routing can be seen in Figure 4.12.

Interlayer connections

The next PCB ordered for internal production at HSLU had poor interlayer connections, most vias did not connect through.
To verify the changes on the SWD pinout, the relevant debugging pins were routed manually with small wires and they seem to be correct.
But for simplicity and time reasons, not all signals were routed manually but an other order was placed at the HSLU internal production with the hope of better interlayer connections but again with insufficient result.
Only on the third HSLU internal order, the inter connection seemed to be satisfying.
But by then, there was not enough time to assemble and test the produced adapter boards.

5 Software

A complete documentation of the software written by Andreas Albisser for the Teensy 3.1 can be found in chapter Section 3.2.
Various issues found with his software can be found in Section 3.3.1.
Now there are two options on how to proceed

- Transfer the existing software to C with FreeRTOS.
- Create a new software concept and implement it.

Both options are evaluated below.

5.1 Transfer existing Software Concept

The existing software concept can be seen in Figure 3.7. As seen in Section 3.3.1, there are various issues that need to be solved. Because the software concept is rather complex and needs refactoring, it is easier to come up with a new software concept.

5.2 New Software Concept

A good approach for a software concept is to divide the responsibilities of tasks similar to ISO/OSI layers. The software concept implemented in the scope of this project can be seen in **??**.
Queues are used as an interface between any two tasks where one task will always be pushing data onto the queue and an other task will pop data from the queue.
The software runs with three main tasks while each task covers an ISO/OSI layer. The purpose of each task is explained in the following sections.

5.2.1 Physical Layer

Description

This task is named SPI Handler and covers ISO/OSI layer 1. It is the only task that accesses the SPI interface. It reads data from the SPI to UART converters and pushes those to a byte queue and it pops bytes from an other byte queue and sends those to the SPI to UART converter.
This task does not know anything about packages, it does byte handling only.

Queue Interface

There are two SPI to UART converters on the baseboard, one for the wireless side and one for the device side. Each SPI to UART converter allows for four UART connections and has an internal buffer of 128 bytes for both RX and TX side of each serial connection.
There are two queues per serial connection, one for bytes read from the SPI to UART converter and one for bytes that need to be forwarded to the SPI to UART converter. There are a total of eight serial connections available to the user which results in 16 byte queues that all interface the SPI Handler.

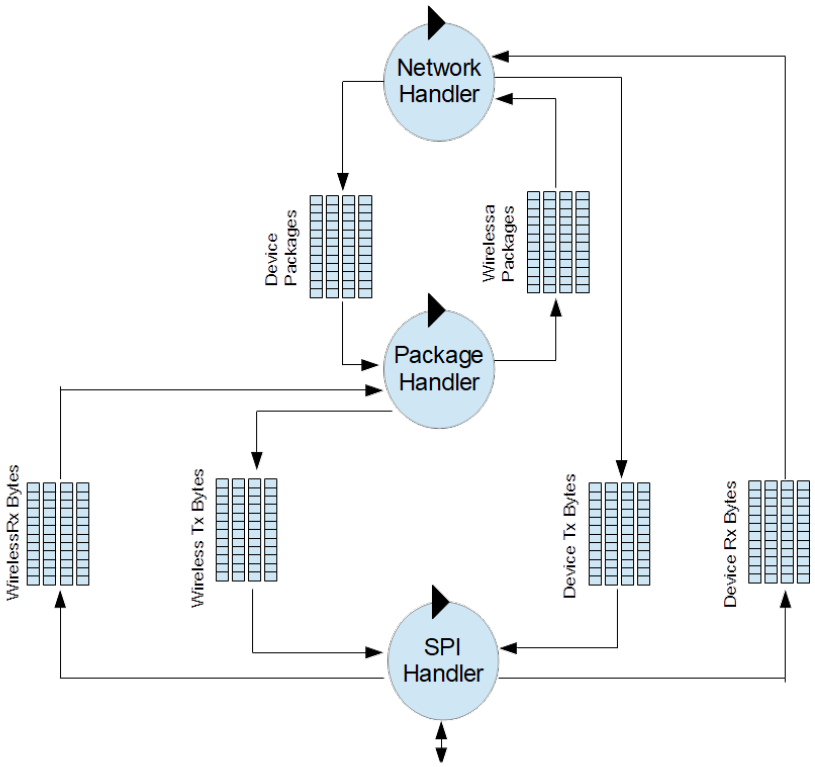

Figure 5.1: New software concept

Dynamic Memory Management

No memory management is needed within this task because it does not allocate or free memory.

Data Loss

As seen in chapter **??**, the tasks in the software developed by Andreas Albisser do not take the state of their interfacing queues into account during runtime.
This issue has been solved for the new software concept. When reading bytes from the SPI to UART converter, the state of the byte queue is not taken into account. Upon unsuccessful push to the corresponding queue because it is full, the oldest ten bytes will be popped from this queue and dropped to ensure storage of the new byte.

```
1  if (xQueueSendToBack(queue, &buffer[cnt], ( TickType_t ) pdMS_TO_TICKS(
       SPI_HANDLER_QUEUE_DELAY) ) == errQUEUE_FULL)
2  {
3      /* queue is full -> delete oldest NUM_OF_BYTES_TO_DELETE_ON_QUEUE_FULL bytes */
4      for(int i = 0; i < NUM_OF_BYTES_TO_DELETE_ON_QUEUE_FULL; i++)
5      {
6          static uint8_t data;
7          xQueueReceive(RxWirelessBytes[uartNr], &data, ( TickType_t ) pdMS_TO_TICKS(
               SPI_HANDLER_QUEUE_DELAY) );
8      }
9      numberOfDroppedBytes[uartNr] += NUM_OF_BYTES_TO_DELETE_ON_QUEUE_FULL;
10     xQueueSendToBack(queue, &buffer[cnt], ( TickType_t ) pdMS_TO_TICKS(
           SPI_HANDLER_QUEUE_DELAY) );
```

Before pushing data to the SPI to UART converter, the state of this converter is checked and the software only transmits as many bytes as the hardware buffer can hold.

```
uint8_t spaceTaken = spiSingleReadTransfer(spiSlave, uartNr, MAX_REG_TX_FIFO_LVL);
spaceLeft = HW_FIFO_SIZE - spaceTaken;
/* check if there is enough space to write the number of bytes that should be written */
if (spaceLeft < numOfBytesToWrite)
{
    /* There isn't enough space to write the desired amount of data - just write as much
        as possible */
    numOfBytesToWrite = spaceLeft;
}
```

Data is dropped unintentionally on unsuccessful queue operations. There is a parameter that specifies a wait time in ticks for a queue operation to finish successfully. This parameter is set to zero for all byte queue operations within the SPI Handler task. This results in failure if a queue operation can not be executed immediately instead of trying again within the specified amount of ticks.

Data Priority and Data Routing

There is no data priority implemented in the SPI Handler task. All task interfacing queues are FIFO queues. The queue for UART interface zero is always processed first, followed by the UART interface one, two and then three. This is done with a for-loop, as can be seen in the code below.

```
for(int uartNr = 0; uartNr < NUMBER_OF_UARTS; uartNr++)
{
    /* read data from device spi interface */
    readHwBufAndWriteToQueue(MAX_14830_DEVICE_SIDE, uartNr, RxDeviceBytes[uartNr]);
    /* write data from queue to device spi interface */
    if(config.TestHwLoopbackOnly)
        readQueueAndWriteToHwBuf(MAX_14830_DEVICE_SIDE, uartNr, RxDeviceBytes[uartNr],
            HW_FIFO_SIZE);
    else
        readQueueAndWriteToHwBuf(MAX_14830_DEVICE_SIDE, uartNr, TxDeviceBytes[uartNr],
            HW_FIFO_SIZE);
    /* read data from wireless spi interface */
    readHwBufAndWriteToQueue(MAX_14830_WIRELESS_SIDE, uartNr, RxWirelessBytes[uartNr]);
    /* write data from queue to wireless spi interface */
    if(config.TestHwLoopbackOnly)
        readQueueAndWriteToHwBuf(MAX_14830_WIRELESS_SIDE, uartNr, RxWirelessBytes[uartNr],
             HW_FIFO_SIZE);
    else
        readQueueAndWriteToHwBuf(MAX_14830_WIRELESS_SIDE, uartNr, TxWirelessBytes[uartNr],
             HW_FIFO_SIZE);
}
```

Data routing is done straight through, meaning that data from a queue with serial number three will be pushed to the hardware buffer for serial interface three.

Issues

The buffer of the SPI to UART converter is never fully taken advantage of for the bytes received. The SPI Handler always tries to read all incoming data, not looking at the state of its internal queue but rather dropping data from the internal queue if it is full just to empty the buffer of the SPI to UART converter.
Instead of always reading all incoming bytes even if the byte queue is full, the SPI Handler should only drop data in case the buffer of the SPI to UART converter is full and the internal queue is full as well.

5.2.2 Data Link Layer

Description

This task is named Package Handler and covers ISO/OSI layer 2. It pops bytes from the queue interface of the physical layer, assembles them to full data packages and pushes them to a package queue. This task also pops packages generated by an upper layer task from another package queue to sends them byte wise to the physical layers byte queue.

Queue Interface

This task interfaces a total of 16 queues: eight byte queues and eight package queues.
It pops bytes from four wireless bytes queues to assemble them to packages and push onto to the wireless package queue for that serial connection.
This task also pops internally generated packages from all four packages queues, splits them into bytes and pushes those bytes to the wireless RX byte queue of the same serial connection.
The internal wireless package structure only holds the pointer to its payload data. When generating a package and storing it in a queue, memory for its payload needs to be allocated and freed later on. Generally, freeing payload is done upon pulling a package from the queue and allocating memory is done before pushing it onto the queue.

Dynamic Memory Management

The package structure used internally can be found in chapter **??**.
The Package Handler allocates memory for saving payload when assembling an incoming package from single bytes. The assembled package is then pushed onto the Wireless Packages queue and the allocated memory will be freed by the Network Handler.
The Package Handler frees allocated memory upon pulling generated packages from the Device Packages queue.
To prevent memory leaks, memory also needs to be freed when a package is dropped but memory for its payload has already been allocated, e.g. on unsuccessful queue push operations.

Data Loss

On receiving side, data is lost when a received package could not be assembled successfully, e.g. when the checksum is not correct, an element in the header is out of range, an unexpected byte appears or any other type of faulty package is received. In this case, data is lost without the upper layer task knowing about it. The state machine for assembling packages will go back to idle state and wait for the next package start sequence.
A successfully assembled package is dropped unintentionally on unsuccessful queue operation. There is a parameter that specifies a wait time in ticks for a queue operation to finish successfully. This parameter is set to zero for all queue operations within the Package Handler task. This results in failure if a queue operation can not be executed immediately instead of trying again within the specified amount of ticks.
On sending side, a package is lost whenever any byte wise push attempt to the Wireless Tx Byte queue is unsuccessful. Again, there is a parameter that specifies the wait time for the push to finish successfully but this parameter is set to zero for all queue operations within the Package Handler task. If unsuccessful, the Package Handler task will drop the package entirely and not push any more parts of it to the byte queue (see code snippet below, line 6). However, the already sent bytes will remain in the byte queue and be pushed out to the hardware buffer by the SPI Handler task.
Before popping an internally generated package from the Device Packages queue and pushing it byte

wise to the Wireless Tx Bytes queue, the Package Handler checks if enough space is available on the byte queue (see code snippet below, line 2).

```
1  /* enough space for next package available? */
2  if(freeSpace > (sizeof(tWirelessPackage) + package.payloadSize - 4))  /*subtract 4 bytes
       because pointer to payload in tWirelessPackage is 4 bytes*/
3  {
4      if(popReadyToSendPackFromQueue(wlConn, &package) == pdTRUE) /* there is a package
           ready for sending */
5      {
6          if(sendPackageToWirelessQueue(wlConn, &package) != true) /* ToDo: handle resending
               of package */
7          {
8              /* entire package could not be pushed to queue byte wise, only fraction in
                   queue now */
9              numberOfDroppedPackages[wlConn]++;
10             FRTOS_vPortFree(package.payload); /* free memory of package before returning
                   from while loop */
11             break; /* exit while loop, no more packages are extracted for this uartNr */
12         }
13         else
14         FRTOS_vPortFree(package.payload); /* free memory of package once it is sent to
               device */
15     }
16 }
```

Data Priority and Data Routing

There is no data priority implemented in the Package Handler task. All task interfacing queues are FIFO queues. The queue for serial connection zero is always processed first, followed by the serial connection one, two and then three.
Data routing is done straight through, which means that data packages from Device Packages queue three are routed to Wireless Tx Bytes queue three.

Issues

Verification of the checksum in both header and payload of incoming data package is still commented out for development reasons. All tests have been carried out with a Teensy 3.1 as counter part because only one functional Teensy adapter board was available. Because the Teensy 3.1 uses a different polynomial for checksum calculation and no further time has been invested in selecting the matching polynomial for the new Teensy 3.5 software implementation, it was easiest to just comment checksum verification out.

5.2.3 Network Layer

Description

This task is named Network Handler and covers all upper layers in the ISO/OSI model. It reads data from device side of the physical layers byte queue and puts them into data packages to send out on wireless side by pushing them onto the package queue of the data link layer. It keeps track of acknowledges received and handles the resending of packages on the correct wireless connection.
This task also extracts data from incoming data packages popped from the data link layer queue, generates acknowledges and pushes the payload to the byte queue of the physical layer task.

Queue Interface

The Network Handler interfaces a total of 16 queues, eight package queues and eight byte queues. It pops packages from the four Wireless Package queues that hold successfully assembled packages per serial connection. Those queues hold both acknowledges and data packages, in the same order as they were received.
This task also generates data packages and pushes them to the correct Device Package queue for the Package Handler to push down byte wise.
The payload from received packages is extracted and sent to the correct Device Tx Bytes queue.
Packages are generated with payload popped from the Device Rx Bytes queue.

Dynamic Memory Management

The package structure used internally can be found in chapter **??**.
The Network Handler allocates memory when generating an new package from device bytes. The assembled package is then pushed onto the Device Packages queue and the allocated memory will be freed by the Package Handler.
In case an acknowledge is expected on a serial connection, the package also needs to be stored internally to allow for resend attempts. The Package Handler will always free memory when popping a package from the Device Packages queue. The Network Handler therefore has to allocate memory for saving a copy of the package internally in case of a pending acknowledge. The allocated memory is only freed upon reception of an acknowledge or when no acknowledge has been received after the last send attempt.
The Network Handler frees allocated memory upon pulling assembled packages from the Wireless Packages queue.
To prevent memory leaks, memory also needs to be freed when a package is dropped but memory for its payload has already been allocated, e.g. on unsuccessful queue push operations.

Data Loss

When a package is received, the payload is extracted and pushed to the Tx Device Byte queue. The state of the Tx Device Bytes queue is checked before popping a package from the Wireless Packages queue and extracting the payload. In case of an unsuccessful push operation on the Device Tx Bytes queue, all data will be lost.

```
1  /* send data out at correct device side */
2  for(uint16_t cnt=0; cnt<package.payloadSize; cnt++)
3  {
4      if(pushToByteQueue(MAX_14830_DEVICE_SIDE, package.devNum, &package.payload[cnt]) ==
           pdFAIL)
5      {
6          XF1_xsprintf(infoBuf, "Warning: Push to device byte array for UART %u failed",
               package.devNum);
7          pushMsgToShellQueue(infoBuf);
8          numberOfDroppedBytes[package.devNum]++;
9      }
10 }
```

The state of the acknowledge queue is not checked before popping a package from the Wireless Packages queue. It is therefore possible for an acknowledge to be generated but not pushed down successfully because of a full queue.
As with the other tasks, there is a parameter that specifies the wait time for the push to finish successfully but this parameter is set to zero for all queue operations within the Network Handler. If unsuccessful, the Network Handler task will drop the package entirely and not push any more parts of it to the byte queue.
This results in package loss for generated acknowledges and data packages on unsuccessful push to

the Device Packages queue as well as in data loss on unsuccessful push of extracted payload to the Device Tx Bytes queue.

Data Priority and Data Routing

All package routing is done within this task. When generating a package and sending it out for the first time, this task checks the configuration file to find out, to which of the four Wireless Package queues it needs to be pushed. It will go through the configured priorities and attempt to send the package to the corresponding wireless serial connections until a queue push attempt is successful (see code below).

```
1 for(int prio=1; prio <= NUMBER_OF_UARTS; prio++)
2 {
3     wlConn = getWlConnectionToUse(rawDataUartNr, prio);
4     if(wlConn >= NUMBER_OF_UARTS) /* maximum priority in config file reached */
5         return false;
6     /* send generated WL package to correct package queue */
7     if(xQueueSendToBack(queuePackagesToSend[wlConn], pPackage, ( TickType_t )
          pdMS_TO_TICKS(MAX_DELAY_NETW_HANDLER_MS) ) != pdTRUE)
8         continue; /* try next priority -> go though for-loop again */
```

If an acknowledge is configured on the wireless connection where the package was sent out, the package will be stored in an internal array of the Network Handler. Because the allocated memory for the package is freed once the Package Handler pulls the package from the queue, the package needs to be duplicated and memory needs to be allocated again for internal storage.
An array with space for 100 unacknowledged packages acts as internal storage. It holds the all unacknowledged packages and information about send attempts and time stamp of the last send attempt (see the storage structure in chapter **??**). This information is required by the Network Handler to calculate the time of the next send attempt in case of no received acknowledge. This task also keeps track of the number of send attempts per wireless connection and will cease all send attempts once the maximum retry timeout has been reached or all send attempts for all wireless connections have been carried out.

Issues

Currently, there is no way for the receiver of a package to know if incoming data is still in the correct order or if a package is missing. The header of a package contains a time stamp but this time stamp is not monotonically increasing but might skip over several numbers in case of a long delay between the generation of two packages.
When the receiver gets a valid package, it will extract its payload immediately and push it to out on the correct device side. The time stamp is not checked for correct order. Wireless packages may therefore arrive in wrong order and device bytes will then be pushed out in wrong order.
Possible solutions for this problem will be presented later in this chapter.

5.2.4 Package Structure

Internally, packages are always saved with the same structure. Apart from header, payload and CRC, the structure also holds information about resend attempts and time stamp of first and last resend. The later information is only needed by the Network Handler but for simplicity reasons, all packages pushed to and popped from queues are of this structure.

```
1 /*! \enum ePackType
2 *  \brief There are two types of packages: data packages and acknowledges.
3 */
```

```
4  typedef enum ePackType
5  {
6      PACK_TYPE_DATA_PACKAGE    = 0x01,
7      PACK_TYPE_REC_ACKNOWLEDGE   = 0x02
8  } tPackType;
9
10
11
12
13 /*! \struct sWirelessPackage
14 *  \brief Structure that holds all the required information of a wireless package.
15 *  Acknowledge has the same sysTime & devNum in header as the package it is sent for.
16 *  The individual sysTime for ACK package is in payload.
17 */
18 typedef struct sWirelessPackage
19 {
20     /* --- payload of package --- */
21     /* header */
22     tPackType packType;
23     uint8_t devNum;
24     uint8_t sessionNr;
25     uint32_t sysTime;
26     uint16_t payloadSize;
27     uint8_t crc8Header;
28     /* data */
29     uint8_t* payload; /* pointer to payload, memory needs to be allocated */
30     uint16_t crc16payload;
31
32     /* --- internal information, needed for (re)sending package --- */
33     uint8_t currentPrioConnection;
34     uint8_t sendAttemptsLeftPerWirelessConnection[NUMBER_OF_UARTS];
35     uint32_t timestampFirstSendAttempt;
36     uint32_t timestampLastSendAttempt[NUMBER_OF_UARTS]; /* holds the timestamp when the
           packet was sent the last time for every wireless connection */
37     uint16_t totalNumberOfSendAttemptsPerWirelessConnection[NUMBER_OF_UARTS]; /* to save
           the total number of send attempts that were needed for this package */
38
39 } tWirelessPackage;
```

5.2.5 Other Software Items

Upon startup, all components are initialized and the scheduler is started with a single task. Within that single task, the configuration file is read from the SD card and saved in a global variable. Only then will all other tasks be started and the init task that read the SD card kills itself.
The three main tasks are the Network Handler, the Package Handler and the SPI Handler. While they make up the main function of this software, there are other tasks responsible for logging, shell, printing of debug information and LED blinking. These tasks are not specified in **??** but will be listed below.

Configuration File

The Teensy 3.5 has an on-board SD card slot for a micro SD card.
The software configuration is stored in a ini file on the SD card and is read once upon startup of the software.
The file serialSwitch_Config.ini has the same parameters as the software developed by Andreas Albisser. Those parameters can be found in **??**. Additional software parameters can be found in **??** :

Configuration parameter	Possible values	Description
TEST_HW_LOOPBACK_ONLY	0, 1	This parameter enables local echo. Any data received over any serial connection will be returned over the same connection immediately.
GENERATE_DEBUG_OUTPUT	0, 1	Information about the data throughput and other warnings will be printed out on the serial terminal.
X_TASK_INTERVAL	1 ... 65535	Task interval in milliseconds for each task. Replace X with the name of the task. Use a number greater than 0 to ensure that other tasks will get called as well.

Table 5.1: Configuration parameters of new software

A sample configuration file can be found in appendix A.
The ini file is read with the miniINI tool (part of the processor expert library provided by Erich Styger) and then parsed. It is read once, on start up of the software. When modifying the configuration file, the device has to be restarted for the changes to take effect.
The miniIni tool supports sections and will find the desired variable and return its value as an array of chars. A parser to save the return values as an array of integers, booleans or any other type had to be implemented manually.

Shell Task

All debug information is printed in an RTT shell provided by the Segger debugging tool.
In order to start the shell, the RTT client or the RTT viewer have to be started during an ongoing debugging session.
The shell runs with its own task. It also has a queue as an interface where other tasks can push strings to for the shell to display.
Both RTT Client and RTT Viewer can be found in the folder where the Segger has been installed (e.g. C:\\Program\Segger\).
The output of the shell looks as in Figure 6.3. All periodic information printed is due to the Trhoughput Printout task. The shell also provides an command line interface for RTOS operations and RTOS status information, SD card operations and others that can be enabled in software. Type "help"to get an overview of the provided shell commands.

Throughput Printout Task

This task is responsible for calculating average throughputs and pushing all this information onto the shell queue for printing.
The output of the printout task looks as in Figure 6.3.

Logging Task

File logging is part of the task description as can be seen in **??**. Due to lack of time, it has not been implemented yet.
Debug information about throughput and other warnings are printed on the shell in the interval configured by the parameter THROUGHPUT_PRINTOUT_TASK_INTERVAL in the configuration file on the SD card.

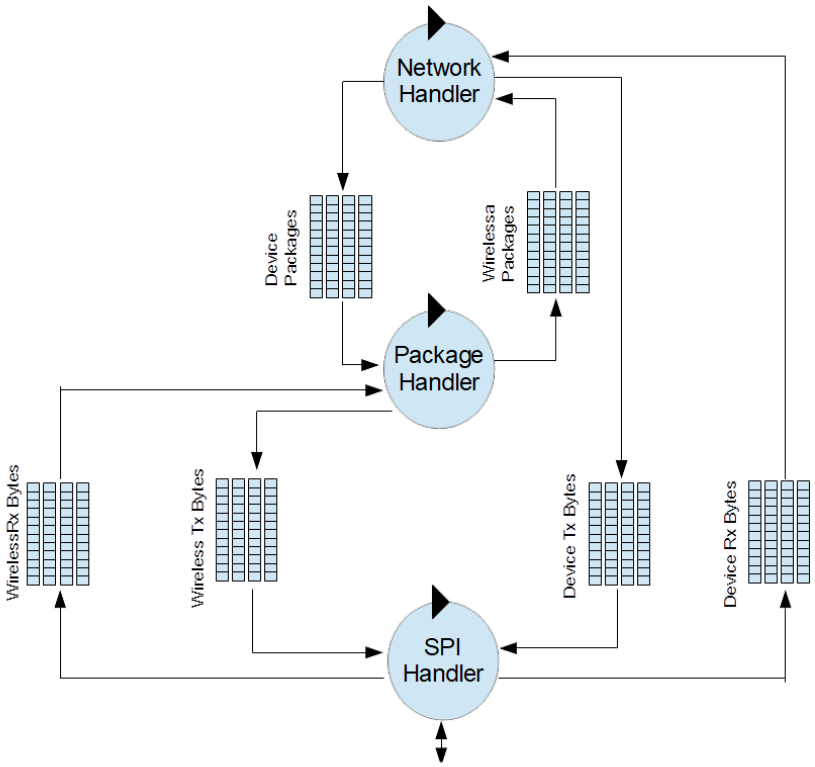

Figure 5.2: New software concept

5.2.6 Next Steps in Software Development

There various issues encoutered with the three main tasks can be seen in their relative section above. Possible solutions are given below.

Package Numbering

All packages should be numbered consecutively instead of a time stamp so the receiver knows if a package is missing and can hold off from sending the received payload out on device side.
The Network Handler could have a buffer that can hold a number of packages to wait for a missing package in the middle. A maximum delay would specify for how long the Network Handler would wait until the received payload is pushed out on device side anyway even though a package in the middle is still missing.
This would require careful thought about reassembling shuffled packages.
An other approach would be to only send out the next package if the previous package has been acknowledged. This would guarantee correct order of the packages received.

Limit Throughput

The parameter MAX_THROUGHPUT_PER_CONNECTION is does not have any effect within the new software. The SPI Handler should limit the throughput for each serial connection to as many bytes per second as specified in the configuration file.

Redundant Package Sending

In Andreas Albissers software, it was possible to configure multiple wireless connections with the same priority. This resulted in the same package being sent out over multiple wireless connections. On receiving side, the software would only take the package that arrived first and discard the second one. This increased reliability of data transmission.
A similar concept should be implemented in this software. It is currently not possible, partly also because the timestamp of a package is always ignored which results in the software not realizing that the same package was received twice over different wireless connections.

Use Hardware Buffer for received Bytes

The SPI Handler always tries to read all received bytes from the hardware buffer, no matter the state of its internal byte queues. When its internal byte queue is full but data is read from the SPI to UART converter anyway, the task will pop the ten oldest bytes from the queue to be able to push new bytes onto the queue.
This should be handled better in a future version of this software.
A possible solution is to only pull data from the hardware buffer after a certain timeout with no free queue space and then dismiss old bytes in the queue.

Data Priority

The user should be able to specify which data is most important when transmission becomes unreliable. When working with QGroundControl, an open source autopilot software, some data exchanged is more important such as exact location of the drone. Other information such as battery level is not of vital importance and can be sent when a stable connection is reestablished. There are two approaches on how solve this problem:

- Implement the communication protocol of QGroundControl inside this software so the Teensy can filter out any information that is not absolutely required when data transmission becomes unreliable.
- Prioritizing one serial device connection over an other so when data transmission becomes unreliable, only packages of the priority device connection are sent out and no others.

Currently, the serial switch does not have any information about the data transmitted and can be used as a flexible platform. If it should stay that way, the second solution is to be preferred.

Unintentional Data Loss

As can be seen in chapter 5.2.4, the internal data structure for packages only holds the pointer to its payload. Every time a package is generated, memory has to be allocated, which could possibly fail.
In order to minimize the risk of data loss because memory allocation failed, the program should only pop data from a queue after memory for the payload has been allocated. Or more generally: all possibly unsuccessful queue or memory operations should be done before popping data from the queue to prevent unintentional data loss.

Session Number and Time Stamp Management

If either the on-board or off-board serial switch software is reset, the session number for packages will change to an other random number and the time stamp will start over.
This scenario has not been tested or considered for the software implementation.

Logging

All logging information should be saved in a file on the SD card. Because a write access to the SD card can take several hundreds of milliseconds, it should be done in a separate task.

ALOHA

Resending of a package should not be done periodically because it might have gotten lost because of a periodical noise. To decrease the possibility of a package always being sent out during the noise occurrence, resend attempts should be done after a random time interval within a certain time slot. This is called the ALOHA principle.

Data Encryption

Data Encryption is part of the task description as seen in appendix B.
For time reasons, it has not been implemented yet. The Teensy 3.6 would support hardware encryption, but the Teensy 3.5 has been chosen for reasons mentioned in Section 4.2.1. More details about encryption and how to implement it in this project can be found in Section 5.2.7

5.2.7 Encryption

To provide security against data manipulation and eavesdropping, data encryption should be implemented on wireless side of the Serial Switch.
There are three parts to data encryption:

- 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.

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.
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.

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 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.

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.

Encryption for Serial Switch

Integrity is assured with the 8 bit CRC in a package header and a 32 bit CRC for the package payload. Confidentiality is not yet assured. Because symmetric encryption requires the least CPU power and is easier to implement, this method would preferably be chosen. Two identical keys could be generated before software startup and saved on the SD card.

6 Testing

Testing can be devided into multiple sections:

- Testing of hardware
- Testing of software

More details about the validation of the system can be found below.

6.1 Hardware Tests

Testing of the base board was not done in the scope of this project because several fully assembled and tested pieces were provided by Andreas Albisser.
Only the Teensy adapter board had to be tested.
As seen in Section 4.3, the first version of the adapter board featured a faulty SWD footprint.
The numbering of this footprint was corrected and a new PCB was ordered at the HSLU internal production. The newly produced version had poor interlayer connections and some wires were soldered onto the PCB to verify correctness of the SWD pinout. Because it would have taken up too much time to wire all faulty connections manually, an other order was placed at the HSLU internal production with the hope of better interlayer connections.
This time, the vias seemed to be of better quality but there was not enough time to assemble an adapter board to test it out.
During the entire project phase, only the first version of the adapter board was available for development and only one board was fully assembled. All software development was therefore done with the setup as in Figure 4.12.

6.2 Software Tests

All software was tested against the Teensy 3.1 with the software implementation Andreas Albisser provided. The reason for this being that only one Teensy adapter board has been assembled and the next version that would have been of use was received too late.
During and after software development, the functionalities had to be tested. The tests conducted were carried out in the following order:

- Echo / loopback
- Connecting two UAV Switches directly on wireless side, two serial terminal with USB used on device side
- Connecting two UAV Switches with modems on wireless side, two serial terminal with USB used on device side
- Connecting two UAV Switches directly on wireless side, autopilot on device side
- Connecting two UAV Switches with modems on wireless side, autopilot on device side

Not only correct functionality had to be ensured but also system performance:

- CPU time of each task
- Memory leaks

Details about the tests carried out can be found in this chapter.

6.2.1 Echo / Loopback

The loopback functionality can be enabled with the parameter TEST_HW_LOOPBACK_ONLY in the configuration file located on the SD card.
When enabled, all bytes received on each serial connection will be returned on the same serial connection. The main functionality is handled inside the SPI Handler task, both Package Handler and Network Handler tasks will not process any data (but will be running with their configured task interval anyway).

```
1  vTaskDelayUntil( &lastWakeTime, taskInterval ); /* Wait for the next cycle */
2  /* read all data and write it to queue */
3  for(int uartNr = 0; uartNr < NUMBER_OF_UARTS; uartNr++)
4  {
5      /* read data from device spi interface */
6      readHwBufAndWriteToQueue(MAX_14830_DEVICE_SIDE, uartNr, RxDeviceBytes[uartNr]);
7      /* write data from queue to device spi interface */
8      if(config.TestHwLoopbackOnly)
9      {
10         readQueueAndWriteToHwBuf(MAX_14830_DEVICE_SIDE, uartNr, RxDeviceBytes[uartNr],
               HW_FIFO_SIZE);
11     }
12     else
13     {
14         readQueueAndWriteToHwBuf(MAX_14830_DEVICE_SIDE, uartNr, TxDeviceBytes[uartNr],
               HW_FIFO_SIZE);
15     }
16     /* read data from wireless spi interface */
17     readHwBufAndWriteToQueue(MAX_14830_WIRELESS_SIDE, uartNr, RxWirelessBytes[uartNr]);
18     /* write data from queue to wireless spi interface */
19     if(config.TestHwLoopbackOnly)
20     {
21         readQueueAndWriteToHwBuf(MAX_14830_WIRELESS_SIDE, uartNr, RxWirelessBytes[uartNr],
               HW_FIFO_SIZE);
22     }
23     else
24     {
25         readQueueAndWriteToHwBuf(MAX_14830_WIRELESS_SIDE, uartNr, TxWirelessBytes[uartNr],
               HW_FIFO_SIZE);
26     }
27 }
```

This can be tested easily by connecting the base board with the following configuration to a computer:

- TEST_HW_LOOPBACK_ONLY = 1 in configuration file
- Set jumper on device side of base board to USB
- Connect computer with base board with an USB cable
- Open a serial terminal (e.g. Tera Term or PuTTy), select one of the two COM ports that appeared (reminder: the base board has two dual USB to UART converter, see section Section 3.1.3)
- Select the baud rate that was set in the configuration file with the parameter BAUD_RATES_DEVICE_CONN
- Turn off local echo on the serial terminal

When running the software in Loopback configuration, connecting the device side as an USB COM port, it will look like an echo on the serial terminal in case the local echo is turned off. If local echo is turned on, each character sent will appear twice on the terminal.

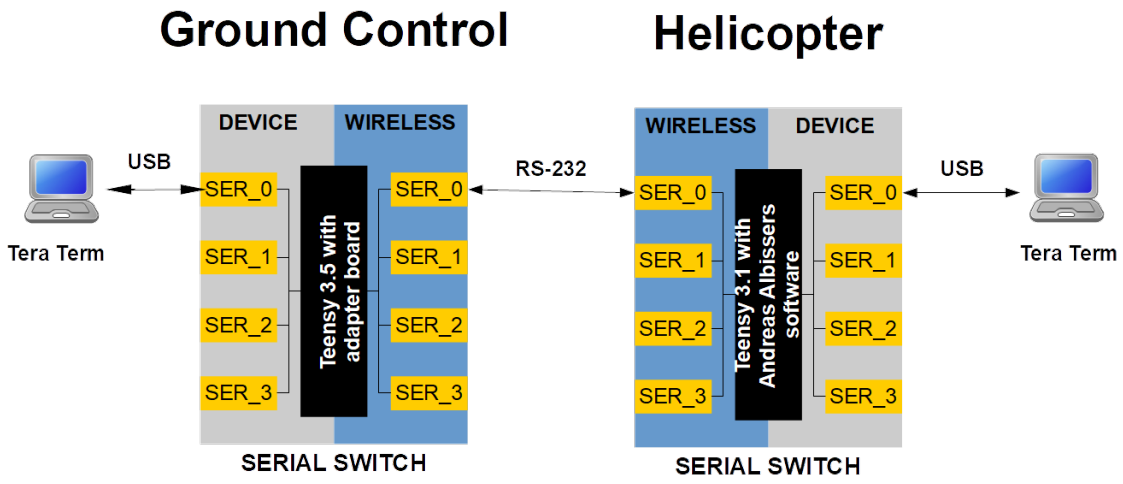

Figure 6.1: Setup with direct connection on wireless side, Tera Term on device side

6.2.2 Direct Connection of Switches, Tera Term on Device Side

Successful package transmission and reception was tested by connecting two UAV Serial Switches directly.
Because only one Teensy adapter board was assembled and tested (see Section 6.1), the second base board was used with the Teensy 3.1 running with the software developed by Andreas Albisser.
The software configuration was as follows:

- Set BAUD_RATES_WIRELESS_CONN to the same value on both UAV Serial Switches for the serial interface used
- Set jumper on device side of base board to USB
- Open a serial terminal (e.g. Tera Term or PuTTy), select one of the two COM ports that appeared (reminder: the base board has two dual USB to UART converter, see section 3.1.3)
- Turn off local echo on the serial terminal
- Select the baud rate that was set in the configuration file with the parameter BAUD_RATES_DEVICE_CONN
- Route the used COM port to wireless serial connection 0 with PRIO_WIRELESS_CONN_DEV_X = 1, 0, 0, 0
- Either set a single byte payload mode with USUAL_PACKET_SIZE_DEVICE_CONN = 1, 0, 0, 0 or set PACKAGE_GEN_MAX_TIMEOUT to a low value (e.g. 5ms) so a package will be generated soon after a single character has been received, no matter how many bytes of payload the package currently holds.

No Acknowledge

First, the software was tested without an acknowledge configured by setting SEND_ACK_PER_WIRELESS_CONN = 0, 0, 0, 0
The software worked as expected. When typing a character on the serial terminal for one UAV Serial Switch, the character appeared at the other terminal and vice versa.

With Acknowledge

The software was also tested with an acknowledge configured by setting SEND_ACK_PER_WIRELESS_CONN = 1, 0, 0, 0

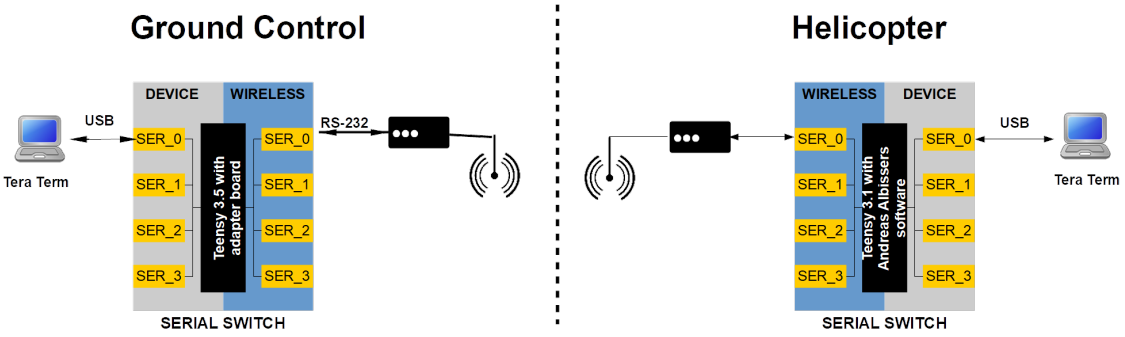

Figure 6.2: Setup with modem connection on wireless side, Tera Term on device side

The software worked as expected. When typing a character on the serial terminal for one UAV Serial Switch, the character appeared at the other terminal and vice versa.

6.2.3 Modem Connection of Switches, Tera Term on Device Side

Package handling was also tested by connecting two UAV Serial Switches with modems. The setup was as can be seen in Figure 6.2.
Because only one Teensy adapter board was assembled and tested (see Section 6.1), the second base board was used with the Teensy 3.1 running with the software developed by Andreas Albisser.
Device configurations were the same as in Section 6.2.2, except for the baud rate on wireless side. The parameter BAUD_RATES_WIRELESS_CONN cannot be chosen freely but has to be set to a value supported by the modem used.
The setup was tested with two different modems, one with RS232 interface and the other (RFD900) with a TTL interface where a level converter had to be used to connect modem and wireless side.
All modems and hardware used was provided by Aeroscout GmbH. There seemed to be issues with the level converters as they were over heating regularly. This problem requires further investigation which is outside the scope of this project.

No Acknowledge

First, the software was tested without an acknowledge configured by setting SEND_ACK_PER_WIRELESS_CONN = 0, 0, 0, 0
The software worked as expected. When typing a character on the serial terminal for one UAV Serial Switch, the character appeared at the other terminal and vice versa.

With Acknowledge

The software was also tested with an acknowledge configured by setting SEND_ACK_PER_WIRELESS_CONN = 1, 0, 0, 0
The software worked as expected. When typing a character on the serial terminal for one UAV Serial Switch, the character appeared at the other terminal and vice versa.

6.2.4 Direct Connection of Switches, Autopilot on Device Side

To test the system under stress, more data needs to be exchanged between the on-board and the off-board Serial Switch. This is the case when using the system with an autopilot. The hardware used for this test case (QGroundControl and PX4) has been introduced in Section 3.3.1.

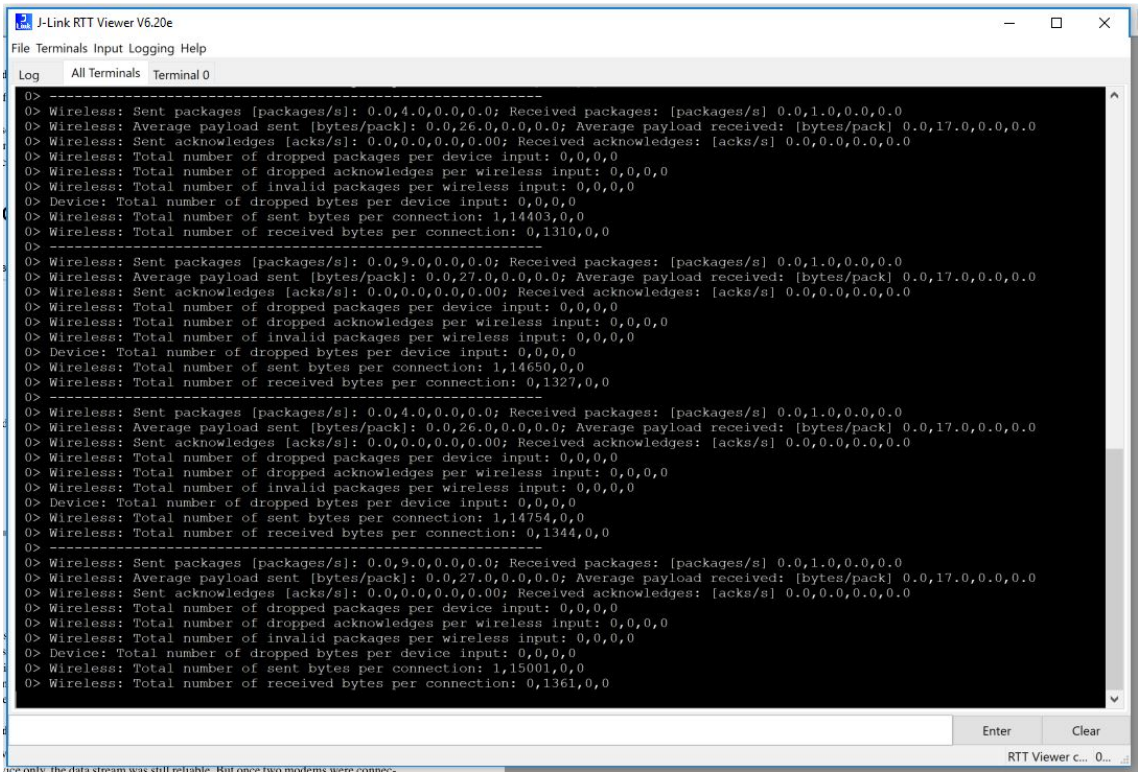

Figure 6.3: QGroundControl in idle mode, no acknowledge configured

The setup is the same as in Figure 3.8 but this time with the new software implementation and on the Teensy3.5.
Because of a lack of Teensy Adapter Boards, two new software implementation could not be tested against each other. The off-board side always consisted of the base board with a Teensy 3.1.
The Serial Switches were configured as follows:

- Set jumpers on device side of both base boards to USB
- Connect computer with one base board by USB cable and open QGroundControl
- Select the baud rate for both Serial Switches that was set in the configuration file with the parameter BAUD_RATES_DEVICE_CONN as configured on pixhawk and QGroundControl (57600 in this case)
- Set BAUD_RATES_WIRELESS_CONN to the same value on both UAV Serial Switches for the serial interface used
- Set USUAL_PACKET_SIZE_DEVICE_CONN = 25, 0, 0, 0 and set PACKAGE_GEN_MAX_TIMEOUT to a low value (e.g. 2ms)
- Set GENERATE_DEBUG_OUTPUT = 1 to enable debug output and the printing interval of the shell to one second by setting THROUGHPUT_PRINTOUT_TASK_INTERVAL = 1

No Acknowledge

The software was tested without an acknowledge configured by setting SEND_ACK_PER_WIRELESS_CONN = 0, 0, 0, 0
First, the PX4 will extablish a connection with QGroundControl. Once the devices have been linked successfully, they will stay in idle mode and less data is exchanged periodically.
The connection was established successfully within about 1 second. The amount of data exchanged during connection establishment and during idle mode could be estimated by looking at the debug

output on the serial temrinal. The output on the serial terminal during idle mode was as can be seen in Figure 6.3.
During idle mode, about 150 bytes/second will be transmitted from PX4 to QGroundControl and about 20 bytes/second will be transmitted from QGroundControl to PX4.
From the debug output on the serial terminal during connection establishment, it can be seen that the PX4 transmits around 2500 bytes/second to QGroundControl and receives around 100 bytes/second from QGroundControl.

With Acknowledge

The software was also tested with acknowledges configured by setting SEND_ACK_PER_WIRELESS_CONN = 1, 0, 0, 0
The software worked as expected. Again, a connection was established successfully within about 1 second and no problems were experienced during idle mode.

6.2.5 Modem Connection of Switches, Autopilot on Device Side

The system was tested under stress conditions as in Section 6.2.4 but this time a modem connection instead of directly connected UAV Serial Switches. The setup can be seen in Figure 3.9.
Software configurations were the same as in Section 6.2.4, except for the baud rate on wireless side. The parameter BAUD_RATES_WIRELESS_CONN cannot be chosen freely but has to be set to a value supported by the modem used.
The setup was tested with the RS232 modem provided by Aeroscout GmbH.
Because of a lack of Teensy Adapter Boards, two new software implementation could not be tested against each other. The off-board side always consisted of the base board with a Teensy 3.1.
Tests were conducted with and without acknowledges configured by modifying the configuration parameter SEND_ACK_PER_WIRELESS_CONN.
Due to a lack of time, this test has only been conducted briefly and further time needs to be invested.
This test was successful when using the RS232 modem with and without acknowledge configured.
The tests were repeated using the RFD900 modem. When no acknowledge was configured, the link establishment successful and took about 2 seconds. When acknowledges were configured, a link could not always be established.

6.2.6 System Analysis

System View is a free tool for system analysis provided by Segger. It reveals runtime behaviour of an application that cannot easily be seen by using a debugger. It provides insight into multi-threading, queues and resource conflicts.
There is a processor expert component for System View in Erich Stygers library that makes the usage of this tool very easy. When including this component, it will take care of the communication between the System View software and the developed user application.
System View records all queue operations and will display them as events. Because each task performs multiple queue events when called, the processor expert component produces lots of data traffic between System View software and user application when the System View software is active and analyzing. This will result in performance loss of the user application.
System View is also limited to capturing one million events for one recording. Because there are many queue operations (which are listed as events), this limit is reached rather quickly.

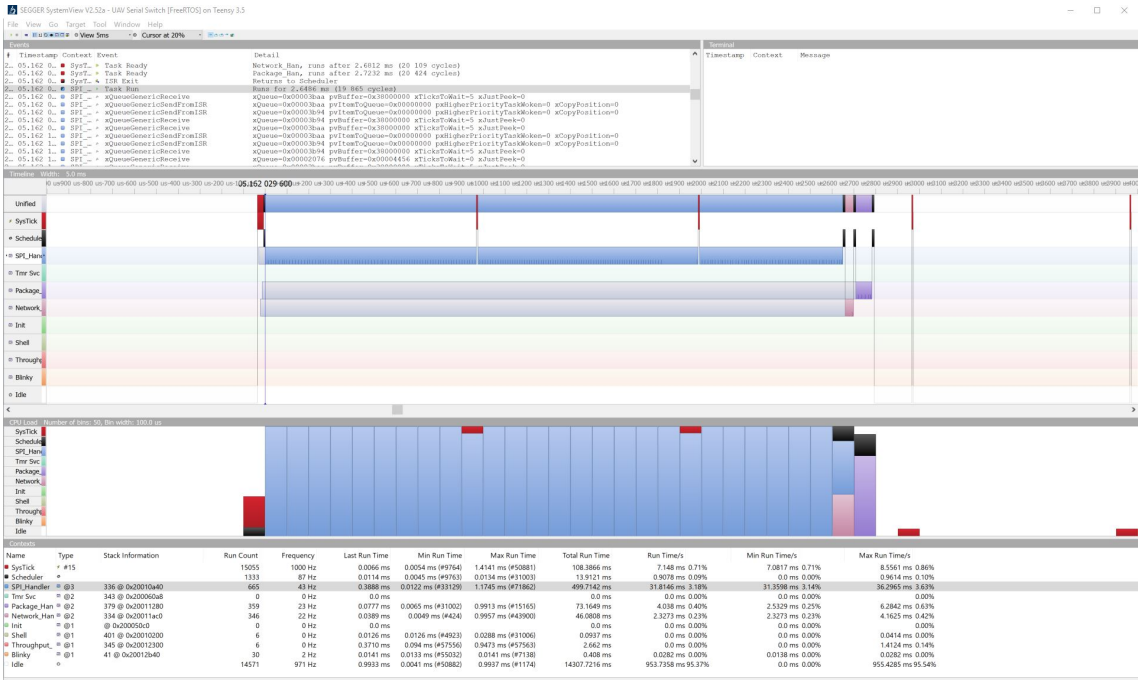

Figure 6.4: System View output when queue events captured

First, the system analysis was done with the processor expert component in its initial state. The software was configured as follows:

- Wireless sides were connected directly by wires
- Acknowledges were configured
- Autopilot running in idle mode, link has already been established
- No debug output configured
- Shell task interval to a very high and debug output deactivated

The System View output for this case can be seen in Figure 6.4. As can be seen in the figure, the SPI Handler takes up most of the CPU time. The SPI Handler task runs for about 2.6 milliseconds, while the Package Handler and the Network Handler task only run for about 0.1 millisecond.
But the system performance is affected by the events logged by the System View processor expert component. Deactivating logging of all queue events results in a much better overall system performance. Deactivate logging of queue events can be done by commenting the respective defines out in SEGGER_SYSTEM_VIEWER_FreeRTOS.h:

```
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)
```

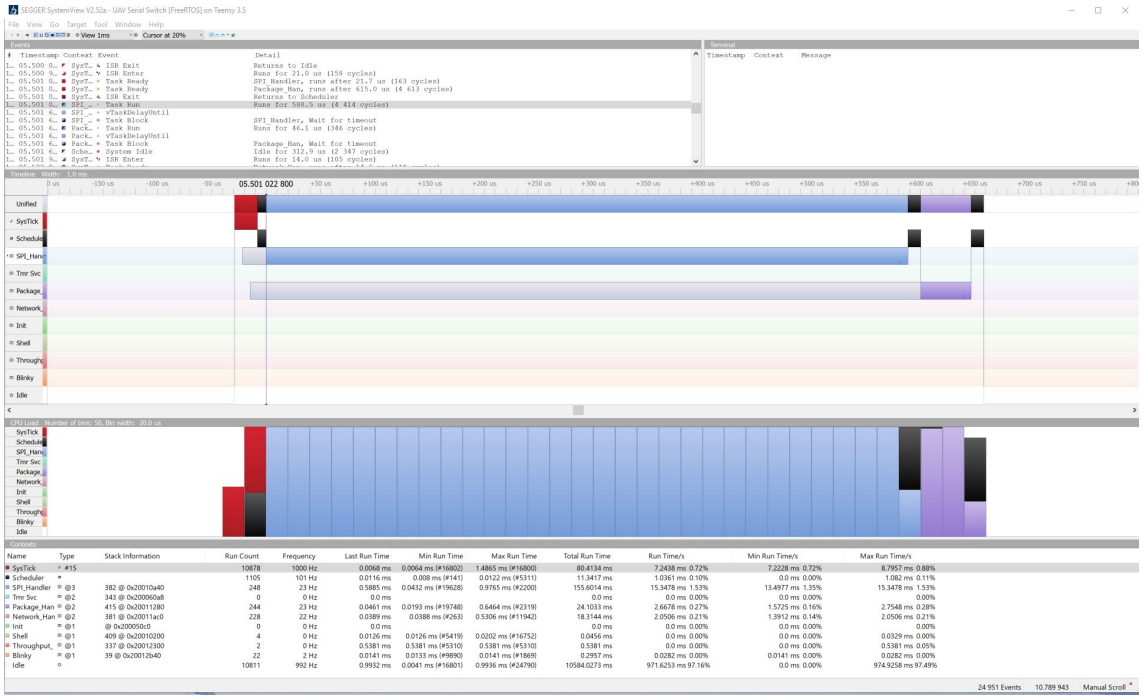

Figure 6.5: System View output when queue events not captured, best case

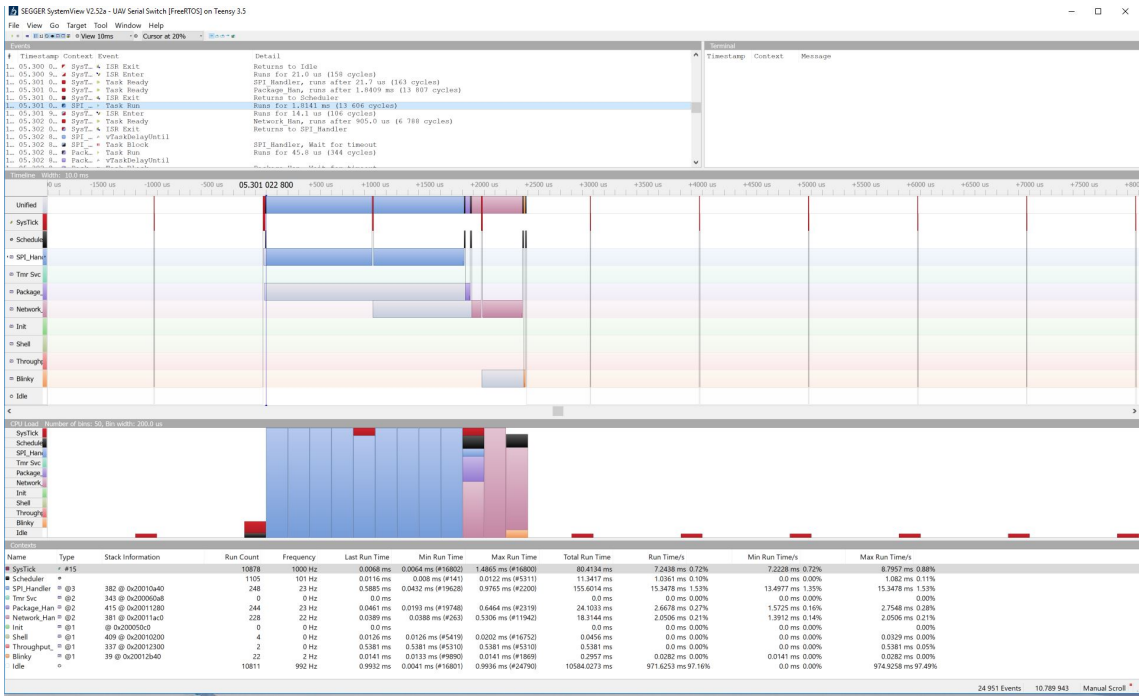

Figure 6.6: System View output when queue events not captured, worst case

```
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)
```

Remember that these lines need to be commented out again every time code is generated by the processor expert component.
Also, the processor expert component in the library is compatible with the System View version 2.42 while the System View version used is 2.52a. For this reason, some of the events logged inside the System Viewer do not match with the name of the actual event that took place. This does not affect the overall outcome of the system analysis though.
Without logging of queue operations, the System View component will generate less traffic. The overall application performance is much better because the tasks take up less CPU time.
When the application is running, the CPU time of the SPI Handler fluctuates depending on the amount of data it needs to read from or transmit to the hardware buffer. This can be seen from the System Viewer output. A best case scenario where only little to no data is exchanged between the application and the hardware buffer can be seen in Figure 6.5. The longest CPU time found during one runtime example can be found in Figure 6.6.
In best case conditions during idle mode of the autopilot, the SPI Handler task runs for 0.6 milliseconds, the Package Handler and Network Handler tasks both run for 0.05 milliseconds.
In worst case conditions during idle mode of the autopilot, the SPI Handler task runs for 1.9 milliseconds, the Package Handler runs for 0.1 milliseconds and the Network Handler task runs for 0.5 milliseconds.

6.2.7 Memory Leak

There are tools to analyze dynamic memory usage such as the Percepio Tracealyzer. Tracelyzer is a real-time visualization tool for RTOS software. There is even a free licence for students in case of non-commercial use.
For time reasons, the Tracealyzer has not been used yet but memory management has been verified by

looking at the free heap available during runtime. The shell offers a command interface for the Free RTOS. There is a command that will display the free heap available to the RTOS. During runtime, this command was run frequently and the heap usage did not change over time, even during stress tests as mentioned in Section 6.2.5 and Section 6.2.4.
Many memory leaks have been found at first as there were numerous cases how the software can allocate memory for data but not free it later on. For example memory can be allocated for a package but never freed in case the package cannot be pushed onto the queue successfully.

6.3 Verification of Task Description

Due to time reasons, not all requirements have been fulfilled. More details can be found in Table 6.1.

Requirement	Fulfilled	Comment
Hardware optimization of weight/size	x	No base board redesign done
Hardware suitable for outdoor use	x	No base board redesign done
More powerful processor used	✓	Teensy 3.5
Hardware debugging interface	✓	SWD debugging
UART Hardware Flow Control	x	No base board redesign done
SD Card	✓	SD card slow on Teensy 3.5 development board
Free RTOS used	✓	
Low Power Mode	x	Low power mode not implemented
Shell/command interface	✓	
Verification with FreeRTOS + Trace	x	Verification with FreeRTOS command interface and System View
Logging on SD card	x	
Configuration on SD card	✓	.ini file
Handling of data loss	✓	All tasks take the state of their queues into account
Encryption	x	
Interleaving	x	

Table 6.1: Verification of Task Description

7 Conclusion

The aim of this project was to come up with a flexible application that routes data of connected devices to modems.
There was no need to start from scratch for this project as some ground work has been done by Andreas Albisser. He developed a hardware with four RS232 interfaces to connect data generating and processing devices and four RS232 interfaces to connect modems for data transmission. The base board has a header to plug in a Teensy 3.2. The Teensy is a small and powerful USB development board that works with Arduino libraries and acts a main micro controller for the base board.
The software Andreas Albisser developed for the Teensy 3.2 is complex and not well thought out. Because requirements were added during development, it is hard to maintain and expand.
The task description of this project features the use of a more powerful micro controller with Free RTOS as an operating system. Using a different micro controller requires hardware changes. There were two options on how to proceed: either redesign the base board for the new micro controller or design an adapter board that routes the used signals from the new micro controller down to the header of the Teensy 3.2. Because of time reasons, the second option was chosen and an adapter board was designed for the new micro controller used.
The Teensy 3.5 was chosen to replace the Teensy 3.2. The Teensy 3.5 features an SD card slot which is also part of the requirements, has more memory and is generally more powerful. Its identical pins are backwards compatible to the pinout of the Teensy 3.2, except for the extra pins because it is slightly longer and has more pins available to the user.
The Teensy 3.5 does have hardware debugging pins available on its backside but in order to use them, the on-board bootloader has to be removed as it is in control of the debug pins and cannot be silenced.
The hardware debugging pins are routed out on a SWD debug header but the footprint of the header was faulty in the first adapter board version ordered. This mistake was corrected and a second version ordered but the second version had poor inter layer connections. The faulty footprint was tested for correctness and seemed A third version was ordered which is probably better from what can be seen under the microscope. For time reasons, it could not be assembled and tested.
Software development was started after the first Teensy Adapter Board had been assembled and the hardware debugging pinout had been corrected manually. First, the software written by Andreas Albisser was analized. Because there was almost no documentation available about the software concept or test results, his software concept had to reverse engineered. Because it is complex and hard to expand, a new software concept was drawn up and implemented. The new software concept is according to the ISO/OSI layers and features three main tasks: one for the physical layer (ISO/OSI layer 1), one for the data link layer (ISO/OSI layer 2) and one for the network layer (ISO/OSI layer 3).
Layer 1 deals with bytes only and communicates with the hardware components directly. Layer 2 does the assembling of data packages and splits generated data packages into bytes. Layer 3 deals with packages only, their resending in case no acknowledge was received, generates acknowledges for received packages and extracts the payload to push out to the devices connected.
Inter task communication is realized with queues, where received data and data to transmit is passed up or down the ISO/OSI layers. All task take the state of all their queues into account during runtime so they will never generate data packages when the queue they should push it to is full. This way, data is never lost intentionally. Currently, each task may lose data unintentionally if a queue operation fails without it being full or empty but for apparently no reason. The only task that will drop data on purpose is the physical layer as it will flush a certain amount of bytes from its queue when full if new data is received and old data has not yet been processed.

Generally, the implementation provides about the same functionality as the software developed by Andreas Albisser. The only parameter missing that was part of Andreas' configuration is the limitation of throughput per wireless connection.
The base provided by this software is well documented and suitable for further expansion which was not the case with the previous software written for the Teensy 3.2.
The next step would be to implement package numbering instead of a system time stamp in the package header. Currently, the receiver does not know about missing packages when looking at the time stamp in the package header because the time stamp is not monotonically increasing. To ensure that packages are always received in the right order, either the sender waits for the acknowledge for a package before sending out the next one or the receiver implements a queue to reassemble packages in their right order before extracting the payload and sending it out to the correct device.
Aeroscout GmbH would also like to have data priority configurable. Currently, there is no configuration parameter that allows the user to prioritize one connected device over an other in case of unreliable wireless connection between on-board and off-board Serial Switch.
Also, logging has not been implemented yet because of time reasons. The Serial Switch currently prints out debug information on the shell but does not save it in a file.
Data encryption and interleaving were part of the task description but have not been implemented yet either.
Generally, the aim was of this project was to provide a software that works at least as good as the one provided by Andreas Albisser.
When testing Andreas Albissers application with an autopilot software as in a real use-case of Aeroscout GmbH, a connection could not be established successfully when acknowledges were configured for the application and on-board and off-board Serial Switches were communicating wirelessly.
When testing the new application with the same autopilot software, a connection was established successfully, even with acknowledges configured and wireless communication between the off-board and on-board devices.
Therefore, the goal of this project has been reached because the outcome seems to be of better quality than the state of the project upon start.
More time should have been invested into the testing phase to get more detailed results of the project outcome.

7.1 Lessons Learned

This project has been educational in many ways. It was my first time using the operating system Free RTOS and I was struggling during the starting phase because of too many new tools.
Also because the start of the project was rather slow and it felt like there was no real progress to show, I was neglecting the documentation up until the very end of the semester. This results in missing pictures in the documentation because they were not taken at that time and missing information because not all the configurations could be remembered when documenting a test case.
During the semester, I spent most time working on the project, implementing features and improving performance. This left me with little time for testing at the end where I should have stopped developent earlier to invest more time in the testing phase so the person to follow up with this project would know exactly where he/she is at.
I do not have a lot of experience with schematics and layouts and did not know about the very high possibility of poor inter layer connections with internally ordered PCBs. Altium Designer together with finding the mistake in the poor inter layer connections cost me a lot of time during this project.
Also, I am a first time LaTex user and there were more than one occation where I remembered Erich Styger's advice: "LaTex without version control is suicide". I can only agree with this statement and

am forever glad he mentioned it so early.
Last but not least, it will be my goal to always keep the requirements in mind at all times during the next project. It is easy to get lost in coding and adding features. One should always take a step back, look at the requirements and the task description to make sure no element is forgotten.

References

Abbrevations

ALOHA	System for coordinating access to a shared communication channel
COM port	Simulated serial interface on computer
ISO/OSI	7 Layers Model
RS232	Serial interface with +-12V
SPI	Serial Peripheral Interface, synchronous communication standard
UART	Universal Asynchronous Receiver Transmitter
UAV	Unmanned Aeral Vehicle

Anhang A Configuration File

A sample configuration file saved on the SD card looks as can be seen below

```
1
2  ;=============================================================================================
3  [BaudRateConfiguration]
4  ;
5  ;
6  ; BAUD_RATES_WIRELESS_CONN
7  ; Configuration of baud rates on wireless side from 0 to 3.
8  ; Regarding the supported baud rates see implementation of hwBufIfConfigureBaudRate in
       hwBufferInterface.cpp
9  BAUD_RATES_WIRELESS_CONN = 57600, 38400, 57600, 57600
10 ;
11 ;
12 ; BAUD_RATES_DEVICE_CONN
13 ; Configuration of baud rates on wireless side from 0 to 3.
14 ; Regarding the supported baud rates see implementation of hwBufIfConfigureBaudRate in
       hwBufferInterface.cpp
15 BAUD_RATES_DEVICE_CONN = 57600, 57600, 38400, 38400
16 ;
17 ;
18 ;=============================================================================================
19 [ConnectionConfiguration]
20 ;
21 ;
22 ; PRIO_WIRELESS_CONN_DEV_X
23 ; Priority of the different wireless connections from the viewpoint of a single device.
24 ; 0: Wireless connection is not used; 1: Highes priority; 2: Second priority, ..
25 PRIO_WIRELESS_CONN_DEV_0 = 1, 0, 0, 0
26 PRIO_WIRELESS_CONN_DEV_1 = 0, 1, 0, 0
27 PRIO_WIRELESS_CONN_DEV_2 = 0, 0, 1, 0
28 PRIO_WIRELESS_CONN_DEV_3 = 0, 0, 0, 1
29 ;
30 ;
31 ; SEND_CNT_WIRELESS_CONN_DEV_X
32 ; Number of times a package should be tried to be sent over a single wireless connection.
33 SEND_CNT_WIRELESS_CONN_DEV_0 = 1, 0, 0, 0
34 SEND_CNT_WIRELESS_CONN_DEV_1 = 0, 1, 0, 0
35 SEND_CNT_WIRELESS_CONN_DEV_2 = 0, 0, 1, 0
36 SEND_CNT_WIRELESS_CONN_DEV_3 = 0, 0, 0, 1
37 ;
38 ;
39 ;=============================================================================================
40 [TransmissionConfiguration]
41 ;
42 ;
43 ; RESEND_DELAY_WIRELESS_CONN_DEV_X
44 ; Time in ms that should be waited until a package is sent again when no acknowledge is
45 ; received per device and wireless connection.
46 RESEND_DELAY_WIRELESS_CONN_DEV_0 = 3, 3, 3, 3
47 RESEND_DELAY_WIRELESS_CONN_DEV_1 = 3, 3, 3, 3
48 RESEND_DELAY_WIRELESS_CONN_DEV_2 = 255, 255, 255, 255
49 RESEND_DELAY_WIRELESS_CONN_DEV_3 = 255, 255, 255, 255
50 ;
51 ;
52 ; MAX_THROUGHPUT_WIRELESS_CONN
53 ; Maximal throughput per wireless connection (0 to 3) in bytes/s.
54 MAX_THROUGHPUT_WIRELESS_CONN   = 10000, 10000, 10000, 10000
55 ;
56 ;
57 ; USUAL_PACKET_SIZE_DEVICE_CONN
58 ; Usual packet size per device in bytes if known or 0 if unknown.
59 USUAL_PACKET_SIZE_DEVICE_CONN = 25, 25, 1, 1
60 ;
```

```
;
; PACKAGE_GEN_MAX_TIMEOUT
; Maximal time in ms that is waited until packet size is reached. If timeout is reached,
; the packet will be sent anyway, independent of the amount of the available data.
PACKAGE_GEN_MAX_TIMEOUT = 2, 2, 20, 20
;
;
; DELAY_DISMISS_OLD_PACK_PER_DEV
DELAY_DISMISS_OLD_PACK_PER_DEV  = 10000, 10000, 10000, 10000
;
;
; SEND_ACK_PER_WIRELESS_CONN
; To be able to configure on which wireless connections acknowledges should be sent if a
; data package has been received. Set to 0 if no acknowledge should be sent, 1 if yes.
SEND_ACK_PER_WIRELESS_CONN  = 0, 1, 0, 0
;
;
; USE_CTS_PER_WIRELESS_CONN
; To be able to configure on which wireless connections CTS for hardware flow control
; should be used. Set to 0 if it shouldn't be used, 1 if yes.
; If enabled, data transmission is stopped CTS input is high and continued if low.
USE_CTS_PER_WIRELESS_CONN = 0, 0, 0, 0
;
;
;=============================================================================================
[SoftwareConfiguration]
;
;
; TEST_HW_LOOPBACK_ONLY
; Set to 0 for normal operation, 1 in order to enable loopback on all serial interfaces
; in order to test the hardware.
TEST_HW_LOOPBACK_ONLY = 0
;
; GENERATE_DEBUG_OUTPUT
; Set to 0 for normal operation, 1 in order to print out debug infos
; (might be less performant).
GENERATE_DEBUG_OUTPUT = 1;
;
; SPI_HANDLER_TASK_INTERVAL
; Interval in [ms] of corresponding task which he will be called. 0 would be no delay -
; so to run as fast as possible.
SPI_HANDLER_TASK_INTERVAL = 5;
;
; PACKAGE_GENERATOR_TASK_INTERVAL
; Interval in [ms] of corresponding task which he will be called. 0 would be no delay -
; so to run as fast as possible.
PACKAGE_GENERATOR_TASK_INTERVAL = 5;
;
; NETWORK_HANDLER_TASK_INTERVAL
; Interval in [ms] of corresponding task which he will be called. 0 would be no delay -
; so to run as fast as possible.
NETWORK_HANDLER_TASK_INTERVAL = 5;
;
; TOGGLE_GREEN_LED_INTERVAL
; Interval in [ms] in which the LED will be turned off or on -> frequency = 2x interval
TOGGLE_GREEN_LED_INTERVAL = 500
;
; THROUGHPUT_PRINTOUT_TASK_INTERVAL
; Interval in [s] in which the throughput information will be printed out
THROUGHPUT_PRINTOUT_TASK_INTERVAL = 5
;
; SHELL_TASK_INTERVAL
; Interval in [ms] in which the shell task is called to refresh the shell
; (which prints debug information and reads user inputs)
SHELL_TASK_INTERVAL   = 10
```

A.1 Unterkapitel im Anhang

Dies hier ist ein Blindtext zum Testen von Textausgaben. Wer diesen Text liest, ist selbst schuld. Der Text gibt lediglich den Grauwert der Schrift an. Ist das wirklich so? Ist es gleichgültig, ob ich schreibe: „Dies ist ein Blindtext“ oder „Huardest gefburn“? Kjift – mitnichten! Ein Blindtext bietet mir wichtige Informationen. An ihm messe ich die Lesbarkeit einer Schrift, ihre Anmutung, wie harmonisch die Figuren zueinander stehen und prüfe, wie breit oder schmal sie läuft. Ein Blindtext sollte möglichst viele verschiedene Buchstaben enthalten und in der Originalsprache gesetzt sein. Er muss keinen Sinn ergeben, sollte aber lesbar sein. Fremdsprachige Texte wie „Lorem ipsum“ dienen nicht dem eigentlichen Zweck, da sie eine falsche Anmutung vermitteln.

A.1.1 Tieferes Kapitel

Noch tieferes Kapitel

Dies hier ist ein Blindtext zum Testen von Textausgaben. Wer diesen Text liest, ist selbst schuld. Der Text gibt lediglich den Grauwert der Schrift an. Ist das wirklich so? Ist es gleichgültig, ob ich schreibe: „Dies ist ein Blindtext“ oder „Huardest gefburn“? Kjift – mitnichten! Ein Blindtext bietet mir wichtige Informationen. An ihm messe ich die Lesbarkeit einer Schrift, ihre Anmutung, wie harmonisch die Figuren zueinander stehen und prüfe, wie breit oder schmal sie läuft. Ein Blindtext sollte möglichst viele verschiedene Buchstaben enthalten und in der Originalsprache gesetzt sein. Er muss keinen Sinn ergeben, sollte aber lesbar sein. Fremdsprachige Texte wie „Lorem ipsum“ dienen nicht dem eigentlichen Zweck, da sie eine falsche Anmutung vermitteln.

Anhang B Task Description

The full task description for this project can be found below.

MSE – Vertiefungsmodul 1

Horw, 17.Sept.2017

Aufgabenstellung für:

Stefanie Schmidiger_____ (Masterstudierende/r)

Embedded Systems und Mikroelektronik_____ (Fachgebiet)

von Prof. Erich Styger_____ (Advisor)

Dr. Christian Vetterli_____ (Experte/Expertin)

1. Arbeitstitel

UAV Serial Switch

2. Fremdmittelfinanziertes Forschungs-/Entwicklungsprojekt

KTI Projekt LINDA „UAV Power Line Inspektion“

3. Industrie-/Wirtschaftspartner

Aeroscout GmbH, ewz

4. Fachliche Schwerpunkte

Schwerpunkt A: Mikrokontroller
Schwerpunkt B: Kommunikation und Bus-Schnittstellen
Schwerpunkt C: Sensoren und Sensorik

5. Einleitung

In autonomen Flugsystemen besteht der Bedarf an einer universellen und sicheren seriellen Verbindung. Diese wird sowohl für interne Board-Systeme als auch zur Kommunikation nach aussen benötigt. Es existiert eine Vorarbeit von Andreas Albisser. Diese soll in dieser Arbeit verbessert und feldtauglich gemacht werden.

6. Aufgabenstellung

Definieren und Verfeinern Sie in Zusammenarbeit mit dem Industriepartner die Anforderung. Die Basisanforderung sind die folgenden:

- Hardware
  - Optimierung Grösse und Gewicht
  - Feldtauglichkeit (Stecker/Anschlüsse/Gehäuse)

  - Leistungsfähigeren Prozessor mit mehr Speicher und RNG/Encryption Support (z.B. K64 oder K66).
  - SWD/JTAG Debugging
  - UART Hardware Flow Control
  - SD Karte (Normal oder Micro)
- Software
  - FreeRTOS als Betriebssystem
  - Low Power: Tickless IDLE Mode mit einfachem IDLE Sleep Mode
  - Shell/Command Line Interface
  - Verifikation mit FreeRTOS+Trace
  - SD-Karte/File System für Logging und Konfiguration (Schlüssel)
  - Behandlung verlorener Datenpakete: Kodierung, Überlagerung/Kodierung

Die Hard- und Software soll zuverlässig in einer Feldumgebung funktionieren. Überlegen Sie sich auch mögliche Lösungen für eine sichere Verbindung (Verschlüsselung). Erstellen Sie einen Projektplan mit den nötigen Meilensteinen. Verifizieren und Testen Sie Ihre Lösung und dokumentieren Sie sowohl mit einer Projektdokumentation (Bericht) als auch mit einem Benutzerhandbuch.

---

7. Durchführung der Arbeit

Termine

Start der Arbeit:	Montag, 17.Sept.2017
Zwischenpräsentation:	Nach Absprache mit Advisor/Experten, Nov. 2017
Schlusspräsentation:	Nach Absprache mit Advisor/Experten, Januar 2018
Abgabe Bericht:	bis Fr. 22.12.2017 – 16.30 Uhr (D311, Prof. Erich Styger)

Organisatorisches

Advisor und Masterstudierende vereinbaren ein wöchentliche Besprechung.

Die Termine für die Präsentationen (Zwischen- und Schlusspräsentation) werden frühzeitig vereinbart.

---

8. Dokumentation

Die wissenschaftliche Dokumentation ist in 3-facher Ausführung zu erstellen.

- die folgende Selbstständigkeitserklärung auf der Rückseite des Titelblattes:
  „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, Datum, eigenhändige Unterschrift"
- Inhaltsverzeichnis.
- eine Zusammenfassung maximal 1 A4.
- einen englischen Abstract maximal 1 A4.
- Kurzlebenslauf maximal 1 A4 (tabelarisch).

Zusätzlich muss dem Advisor eine CD mit dem Bericht (inkl. Anhänge), mit den Präsentationen, Messdaten, Programmen, Auswertungen, usw. abgeben werden.

9. Fachliteratur/Web-Links/Hilfsmittel

Vorarbeiten (Software, Layout, PCB's) von Andreas Albisser.

10. Zusätzliche Bemerkungen

- keine

11. Beilagen

- Bewertungsraster
- Hinweise zu Projektarbeiten

Horw, 18.Sept.2017

Advisor	Experte/Expertin	Studierende
_____	_____	_____

→ eine Kopie der Aufgabenstellung ist vor Semesterbeginn an den Studiengangleiter abzugeben!

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

Todo list