

# Arbeitstitel: Eingliederung einer zusätzlichen Hardwarekomponente in ein bestehendes Hexacopter System zur Leistungssteigerung

Malte Markus Breitenbach, Autor B, Autor C

**Zusammenfassung—**Hierhin kommt eine kurze (5-6 Sätze) Zusammenfassung der Arbeit. In diesem Fall beschreibt das Dokument die  $\LaTeX$ -Vorlage für die Erstellung der Ausarbeitungen eines Projektseminars.

**Abstract—**This is the english translation of your „Zusammenfassung“.

## I. EINFÜHRUNG

Ein Multikopter stellt mit seinen zahlreichen Messgrößen und dynamischem Verhalten eine zugleich komplexe als auch herausfordernde Regelaufgabe dar.

Der digital ausgeführte Regler erfordert somit ein hohes Maß an Rechenleistung. Es ist daher erforderlich, eine ausreichend leistungsfähige Rechenhardware zur Verfügung zu haben, was jedoch nicht bei jedem Multikopter gegeben ist. Keine Ausnahme stellt der Asctec Firefly mit seinem frei programmierbaren High-Level-Prozessor dar. Sowohl der kleine Programmspeicher als auch die fehlende Rechengeschwindigkeit insbesondere bei Gleitkommaoperationen schränken den Regelungstechniker bei der Lösungsfindung stark ein. So muss beispielsweise durch den limitierten Programmspeicher auf lange Codesequenzen verzichtet werden und durch die fehlende Rechengeschwindigkeit zeitintensive Rechenoperationen nur sparsam Anwendung finden. Abhilfe schaffen soll der in jeder Hinsicht performantere Microcontroller „Nucleo STM32F767ZI“ (im Folgenden als Nucleo abgekürzt), der dem System als weiterer Rechenprozessor hinzugefügt werden soll. Um den neuen Prozessor sinnvoll in die Berechnungen miteinzubeziehen, muss jedoch zunächst eine Schnittstelle zwischen dem zu leistungsschwachen High-Level-Prozessor und dem Nucleo hergestellt werden. Dabei gelten besondere Anforderungen an die Schnittstelle, wie hohe Transferraten, ein hohes Maß an Robustheit als auch Echtzeitfähigkeit. Nur wenn diese Anforderungen auch ausreichend erfüllt werden, können später die Berechnungen praktisch auf dem Nucleo durchgeführt werden und der Hexakopter entsprechend geregelt werden.

Somit soll durch diese Arbeit eine solide Grundlage geschaffen werden, um darauf aufbauend eine Regelung für den Hexakopter entwerfen zu können.

Das vorliegende Paper wird zunächst in die notwendigen Grundlagen einführen. Daran anschließend wird (FORTSETZUNG, wenn finale Struktur feststeht)

Diese Arbeit wurde von M.Sc. Raúl Acuña Godoy, Dipl.-Ing. Dinu Mihailescu-Stoica unterstützt.

## II. GRUNDLAGEN

### A. Systembeschreibung

#### 1) UAV: Asctec Firefly

Im praktischen Mittelpunkt dieses Papers steht ein UAV (zu englisch *unmanned aerial vehicle*). Genauer wird der Hexakopter *Firefly* der Firma *Ascending Technologies* verwendet, der sich mit zahlreicher Sensorik und Erweiterbarkeit an den Forschungsmarkt richtet. Die Größe des *Firefly* beträgt 60.5 x 66.5 x 16.5 cm und durch sechs 100 W Motoren kann es bis auf 12 m/s beschleunigt werden. Die Steuerung bzw. Regelung des *Firefly* wird durch zwei Mikroprozessoren realisiert, die auf dem sog. *AscTec AutoPilot Board* platziert sind. Neben den Prozessoren sind darauf auch verschiedene Sensoren wie z.B. ein Gyroskop, Beschleunigungs- und Drucksensor integriert. Mit diesen Sensoren kann das UAV als inertielle Messeinheit (zu englisch *inertial measurement unit* oder kurz *IMU*) modelliert werden. Zahlreiche physikalische Schnittstellen stehen dem Benutzer auf dem Board zur Verfügung. Somit ist die Kommunikation zwischen UAV und verschiedenen Peripherien bzw. weiteren Sensoren möglich. Die wesentliche Steuerung des UAV basiert wie bereits erwähnt auf zwei integrierten ARM7 Mikroprozessoren. Zum einen dem sogenannten *LLP* (engl. Abkürzung für *Low Level Processor*), der Sensorik ausliefert, Motoren ansteuert sowie einen stabilen Regelalgorithmus implementiert hat, auf den im Notfall zurückgegriffen werden kann. Neben dem unveränderlichen Code des *LLP* kann der Benutzer eigene Algorithmen auf dem sogenannten *HLP* (engl. Abkürzung für *High Level Processor*) implementieren. *Ascending Technologies* stellt zu diesem Zweck mit dem *AscTec SDK* ein Softwareentwicklungswerkzeug zur Verfügung, was die Programmierung vereinfacht. Die Vorgehensweise zur Programmierung wird in [VERWEIS AUF TUTORIAL] genauer vorgestellt. Wie Bild ?? andeutet, kommunizieren *HLP* und *LLP* über eine SPI-Schnittstelle. Somit werden Sensor- und Steuerungsdaten zwischen beiden ausgetauscht und dem jeweils anderen Partner bereitgestellt.

#### 2) Sensorik: Ein Überblick

Sensorik spielt für die Regelung des UAV eine zentrale Rolle. Es kann prinzipiell zwischen zwei Sensortypen unterschieden werden. Zum einen den Sensoren, die im UAV integriert sind und zum anderen äußere neu hinzugefügte Sensoren. Das *Firefly* besitzt bspw. einen Gyrosensor, Kompass und auch ein GPS-Empfänger. Diese sind zunächst ausreichend für weniger anspruchsvolle Situationen, jedoch bspw. nicht gut geeignet für Indoor-Situationen. Um besser zu positionieren und die



Bild 1. Äußere Sensoren

Haltung besser zu schätzen, werden darüber hinaus noch weitere Sensoren wie eine Kamera oder auch *PX4Flow* (optischer Flusssensor) hinzugefügt. Als innere Sensoren kommen folgende zum Einsatz:

- Gyrosensor (zeigt die lineare Beschleunigung und Winkelgeschwindigkeit)
- Kompass (misst das Magnetfeld der Erde)
- GPS (misst u.A. die globale Position, Geschwindigkeit und Zeit)
- Drucksensor (zur Höhenmessung)

Als äußere Sensoren des UAV werden folgende eingesetzt:

- Optische Flussmessung (GPS kann nicht innerhalb des Gebäudes arbeiten. Daher ist die Messung des optischen Flusses eine gute Ergänzung. Konkret in dieser Arbeit angewendet wird PX4FLOW, das das GPS als Positionierungssystem ersetzt. Im Laufe dieser Arbeit wird noch auf das Auslesen und die Integration in das Protokoll eingegangen. (Ref))
- *LIDAR-Lite v3* (für Indoor-Höhen-Bestimmungen, dabei schneller als Ultraschall und sehr zuverlässig (Ref)).

Das in Form dieser Arbeit erstellte Protokoll soll zulassen, dass auch im Nachhinein noch weitere Sensoren hinzugefügt werden können, so bspw. eine Kamera.

### 3) Nucleo: STM32F767 Nucleo-144

Wie zuvor erwähnt, wird der Sensorfusion Algorithmus auf dem hinzugefügten Hochleistungsprozessor „Nucleo STM32F767ZIT6“ implementiert. Der von STMicroelectronics entwickelte Microcontroller findet aufgrund seiner hohen Leistungsfähigkeit heutzutage breite Anwendung. Microcontroller enthalten neben Prozessor, Arbeits- und Programmspeicher zugleich auch Peripheriefunktionen. Dies umfasst nicht nur simple Peripherien sondern auch komplexe Peripheriefunktionen wie z.B. PWM-Ausgänge, CAN, USB, I2C und SPI. Diese Komponenten werden komplett und kompakt auf demselben Chip integriert, was eine kostengünstige und flexible Möglichkeit für Benutzer bietet Projekte zu realisieren. Dank weit verbreiteter Entwicklungswerkzeuge (vgl. »ST WEBLINK«) wird die Entwicklungsarbeit sehr erleichtert. Durch die weite Verbreitung der STM32-Familien existieren zahlreiche Bibliotheksfunktionen und Tutorials, die vor allem den Einstieg erleichtern und auch später noch Entwicklungszeit verkürzen. Auch GUI-Werkzeuge wie CubeMX («LINK»), die als Codegeneratoren einfach und sicher Grundkonfigurationen bzw. ein Grundgerüst erstellen, sind weit verbreitet und häufig in der

Anwendung. Zur genaueren Vorgehensweise zur Codegenerierung und der übrigen Werkzeugkette wird auf (TUTORIAL) verwiesen.

Der STM32767 besitzt einen ARM Cortex-M7 Kern, der bis maximal 216 MHz arbeiten kann. Zudem erhält er 2 MByte Flash-Speicher, 512Kbytes SRAM und 144 Pins, während der Stromverbrauch mit c.a. 250mA (alle Peripheriegeräte aktiviert. TODO: BibReference reference datasheet P124) zeitgleich sehr gering bleibt. Das Debugger- und Programmierwerkzeug ST-LinkV2 ist bereits implementiert. Ausreichende Peripheriegeräte sind implementiert, darunter insgesamt vier I2C Schnittstellen sechs SPI Schnittstellen. Die später noch verwendete SPI-Peripherie kann eine maximale Transferrgeschwindigkeit von 25 Mbit/s erreichen und ist somit gut für größere Datenmengen geeignet.

### B. Serial Peripheral Interface

#### SPI

Das Serial Peripheral Interface (kurz SPI) ist ein im Jahr 1987 von Susan C. Hill Et al., damals Motorola (heute NXP Semiconductors), entwickeltes Bus-System. Es stellt einen „lockeren“ Standard für einen synchronen seriellen Datenbus dar, mit dem digitale Schaltungen nach dem Master-Slave-Prinzip miteinander verbunden werden (BibReference).

SPI verwendet im Gegensatz zu vielen anderen Bussystemen keine Adressen zur Slave Auswahl, sondern einen reservierten Pin als „Chip Select“. Das Interface ist ein voll duplexfähiger Bus und weist vergleichsweise hohe Kommunikationsgeschwindigkeiten im Vergleich zu anderen gängigen seriellen Bussen wie USART, I2C auf. Typischerweise findet die Verbindung, wie Bild 2 gezeigt, sternförmig statt. Die Masterperipherie ist mit jedem Slave über folgende vier Leitungen verbunden:

- SCLK (Serial Clock) auch SCK, wird vom Master zur Synchronisation ausgegeben
- MOSI (Master Output, Slave Input) oder SIMO (Slave Input, Master Output)
- MISO (Master Input, Slave Output) oder SOMI (Slave Output, Master Input)
- $\overline{SS}$  (Slave Select), oder  $\overline{CS}$  (Chip Select).

Dabei besitzt jeder Slave seine eigene „Slave Select“-Verbindung, die übrigen Leitungen werden geteilt. Eine Bitübertragung findet immer abhängig vom SCLK statt, man spricht daher auch von einem synchronen Datenbus.

Der Protokollablauf wird in Bild 3 veranschaulicht. Legt der Master die Leitung „Slave Select“ auf logisch Null, ist der jeweilige Slave aktiv. Dann werden wortweise in beide Richtungen Daten ausgetauscht.

Die Polarität vom Clocksignal bei der Datenübertragung ist durch Motorola nicht fest definiert. Sie werden durch „CPHA“ (Clock Phase) und „CPOL“ (Clock Polarity) definiert. Dadurch wird entschieden, ob bei einer fallenden oder einer steigenden Flanke Bits übernommen werden und auf welchem Niveau „Clock Idle“ ist.

In jeder Taktperiode wird ein Bit übertragen. Beim üblichen Bytetransfer sind also acht Taktperioden für eine vollständige Übertragung nötig. Der Master überträgt durch MOSI ein Bit



Bild 2. SPI-Verbindung durch Kaskadierung der Slaves

Bild 3. Testresultate bei variiierenden Einflussparamtern

Richtung Slave und übernimmt durch MISO ein Bit gesendet vom Slave. Wie genau die empfangenen Bits gespeichert werden unterscheidet je nach Realisierung der Peripheriekomponente. In der Regel kommen Schieberegister und FIFOs (First In First Out) zum Einsatz, damit Daten für kurze Zeit gepuffert werden können. Eine Übertragung ist beendet, wenn das Slave-Select-Signal endgültig wieder auf logisch Eins gesetzt wird.

### C. Protokolltheorie

Um eine Kommunikation zwischen den zuvor erwähnten Mikrocontrollern zu ermöglichen, bedarf es eines gut geplanten sowie gut umgesetzten Kommunikationsprotokolls.

Ein Kommunikationsprotokoll ist nach [1] nichts anderes als eine Verhaltenskonvention zwischen zwei Kommunikationspartnern (oft auch als Instanzen bezeichnet). Durch diese Verhaltenskonvention wird einerseits der zeitliche Ablauf der Kommunikationsaktionen zwischen den Partnern definiert. Andererseits wird auch die Form der zu übermittelnden Nachrichten eindeutig festgelegt. Diese Nachrichten werden auch als PDUs (*protocol data units*) bezeichnet und ihre Struktur muss bei beiden Kommunikationspartnern bekannt sein, damit sie identisch interpretiert werden. Damit ein Protokoll erfolgreich zwischen zwei Partnern ausgeführt werden kann, müssen besagte Partner über bestimmte Protokollfunktionen verfügen. Dazu zählt bspw. die PDU-Codierung oder Fehlerkontrolle.

Einen wichtiger Begriff in der Protokolltheorie stellen die sogenannten Schichten dar, durch die sich die Funktionalität eines Protokolls veranschaulichen lässt. Eine Schicht umfasst dabei vereinfacht ausgedrückt alle Teilfunktionalitäten, die zur Erfüllung einer bestimmten Zielstellung herangezogen werden.

Während der Protokollausführung interagieren besagte Schichten miteinander, wobei jede Schicht (N) die Funktionalitäten der nächsttieferen Schicht (N-1) verwendet. Dabei ist für die höhere Schicht irrelevant, wie die benötigte Funktionalität genau realisiert wird. Als Beispiel für eine Schicht wäre die *physikalische Bitübertragungsschicht* als unterste Schicht oder die darüberliegende *Datensicherungsschicht* anzusehen. Auf beide Schichten wird später noch genauer eingegangen.

Das nachfolgendem Bild 4 veranschaulicht die Generierung einer PDU über mehrere Schichten beispielhaft. Dabei soll deutlich werden, dass ein Datentransfer ausschließlich in der untersten Schicht N=1 stattfindet. Es wird offensichtlich, dass die Schichten N=1 und N=2 während der PDU Generierung die PDUs noch durch zusätzliche Informationen ergänzen, die bspw. für den eigentlichen Bit-Transfer notwendig sind. Während dem Empfang auf der rechten Seite werden diese Zusatzinformationen wieder entfernt und die ursprüngliche PDU aus Schicht N=2 steht auf der Empfangsseite bereit.

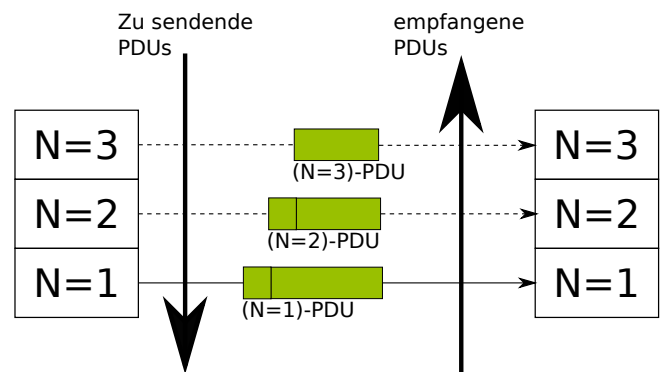


Bild 4. Bildung von PDUs in der Schichtendarstellung (angelehnt an [1])

## III. ENTWICKLUNG UND VORSTELLUNG DES PROTOKOLLS

### A. Vorgehensweise

Wie in [1] aufgeführt, kommt zur Entwicklung des Protokolls ein einfaches Wasserfall-Modell zur Anwendung. Bild 5 zeigt den groben Verlauf dieses Entwicklungsprozesses. Prinzipiell werden die einzelnen Aufgaben nacheinander abgearbeitet, es wird jedoch nicht ausgeschlossen, dass nach der Feststellung einer Designschwäche wieder zu einem vorherigen Schritt zurückgekehrt werden kann.

Zunächst wird demnach eine Anforderungsspezifikation ermittelt, die das System möglichst umfassend berücksichtigt. Aus dieser Spezifikation wird dann ein Entwurf des Protokolls erstellt, das die Anforderungen möglichst optimal erfüllt. Es folgt die eigentliche Implementierung, die diesen Entwurf auf die Zielsysteme angepasst implementiert. Nach der Implementierung werden verschiedene Testszenarien durchlaufen, um das Protokoll auf seine Leistungsfähigkeit und Robustheit zu untersuchen. Werden in diesem finalen Schritt Designfehler festgestellt, muss wieder zum vorherigen Schritt zurückgekehrt werden. Im Folgenden werden die Ergebnisse der einzelnen durchlaufenen Schritte vorgestellt.

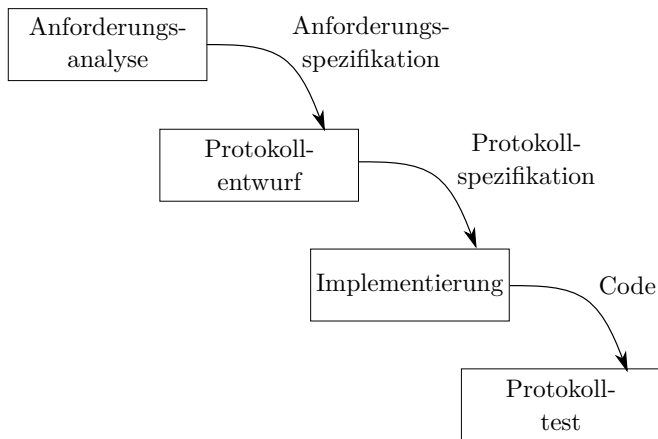


Bild 5. Wasserfall-Modell des Protokollentwicklungsprozesses (angelehnt an [1])

### B. Anforderungsanalyse

Wie bereits in der Einleitung motiviert, sind besondere Anforderungen zu erfüllen, damit das Protokoll praktisch einsetzbar ist. Zu den primären Anforderungen gehört zunächst eine ausreichend hohe Transfargeschwindigkeit, ein hohes Maß an Robustheit als auch Echtzeitfähigkeit. Die Transfargeschwindigkeit sollte groß genug sein, um eine ausreichend große Anzahl an PDUs zwischen den Microcontrollern senden zu können. Robustheit ist ebenfalls eine wichtige Anforderung, denn sowohl unerkannte Übertragungsfehler als auch ein undefinierter „festgebremster“ Status führen zum sofortigen Absturz. Diese beiden Fehler gilt es demnach unbedingt zu verhindern. Weiterhin sollte die Anzahl an (zwar) erkannten Fehlübertragungen nicht zu hoch sein. Zuletzt muss auch Echtzeitfähigkeit gegeben sein. Nach [2] wird unter Echtzeitfähigkeit verstanden, dass anfallende Daten zuverlässig innerhalb einer vorgegebenen Zeit verarbeitet werden und auch verfügbar sind. Orientiert an den Gegebenheiten des Standardprogramms auf dem High-Level-Prozessor wird zunächst eine Paketgeschwindigkeit von 1000 PDUs pro Sekunde angestrebt. Daraus lässt sich die Mindesttransfargeschwindigkeit ableiten, wenn man zunächst von einer Paketgröße von 150 Byte ausgeht. Wenn  $n$  PDUs mit jeweils  $X$  Byte Paketgröße übertragen werden sollen, muss für die Transferrate

$$T(n, X) > \frac{n * X * 8\text{bit}}{1\text{s}}$$

gelten. Für das angestrebte  $n = 1000$  und  $X = 150$  ergibt sich demnach eine Mindesttransferrate von

$$T(1000, 150) > 1,2 \frac{\text{Mbit}}{\text{s}}.$$

Diese Rechnung dient jedoch nur der groben Orientierung und beschreibt den absoluten Optimalfall. In realen Systemen muss zusätzlich Zeit berücksichtigt werden, um bspw. einen Transfer einzuleiten oder neue Daten bereitzustellen. Diese Transferrate ist weder mit UART noch mit I2C (fast mode) erreichbar, daher fällt die Wahl auf das bereits eingeführte SPI.

Als zusätzliche Anforderung ist noch ein Betrieb ohne Nucleo gewünscht. Demnach darf der High-Level-Processor auch durch eine physisch getrennte SPI-Schnittstelle nicht beeinflusst werden und nach wie vor lauffähig bleiben. Der HLP soll demnach möglichst kontinuierlich und unabhängig vom Nucleo Daten aussenden.

### C. Protokollentwurf

In [1] werden mehrere Protokollfunktionen aufgezählt, die zur Erfüllung der Anforderungsspezifikation ggf. notwendig sind, wie

- Synchronisation,
- PDU-Codierung/-Decodierung,
- Fehlerkontrolle,
- Fluss-Steuerung oder
- Anpassen der PDU-Größen.

Als *Synchronisation* wird eine Methode bezeichnet, um beide Kommunikationspartner in einen definierten Zustand zu bringen (bspw. beim Verbindungsaufbau oder -abbau). Eine einfache Methode stellt der 2-Wege-Handshake dar, wobei bei der Wunsch zum Aufbau einer Verbindung durch eine festgelegte Antwort bestätigt wird. Damit würde man jedoch gegen verschiedenste Anforderungen verstoßen. Zum einen wurde ein Handshake-Verfahren im späteren Verlauf der Entwicklung getestet und hat sich als zu langsam heraus gestellt. Zum anderen widerspricht eine Synchronisation zwischen beiden Microcontrollern der Anforderung, dass der HLP möglichst unabhängig vom Nucleo sein soll. Als Alternative kommt ein Ringspeicher (zu engl. *ringbuffer*) zum Einsatz, der empfangene Nachrichten zwischenspeichert, bis die entsprechenden Ressourcen frei sind oder Bedarf an neuen Daten besteht und die Nachricht(en) extrahiert werden. Diese Art der Speicherung erfordert jedoch eine eindeutige Form einer jeden Nachricht um Anfang und Ende zu erkennen. Diese Forderung führt zur nächsten Protokollfunktion, genauer zur *PDU-Codierung/-Decodierung*. Ein reiner Datentransfer kann theoretisch alle Byte-Werte annehmen und sieht keine zusätzlichen Informationen wie Startbytes, Kontrollsummen, etc. vor. Die Extraktion einer Nachricht aus einem Puffer ist jedoch nur schwer möglich, wenn weder Anfang noch Ende der Nachricht klar ersichtlich sind. Durch die Einführung eines sogenannten Trennzeichens (zu engl. *delimiter*) lässt sich dies jedoch ändern.

*COBS* Consistent Overhead Byte Stuffing (im Folgenden mit *COBS* abgekürzt) beschreibt einen Algorithmus, der eine beliebige Folge von Bytes encodiert. Die resultierende Byte-Folge ist dabei frei von einem fest gewählten Trennzeichen, sodass dieses Zeichen in der Folge für die Begrenzung eines jeden Pakets verwenden kann. Um zu garantieren, dass sich das Trennzeichen nicht mehr innerhalb der Byte-Folge befindet, müssen zusätzliche Bytes hinzugefügt werden (zu engl. *byte stuffing*). Der Algorithmus zeichnet sich dabei durch seine Effizienz und Zuverlässigkeit aus. Effizient ist er sowohl bezüglich des Rechenaufwands bei der En- und Decodierung als auch bei der Anzahl an zusätzlich hinzugefügten Bytes.

—ToDo: Erklärung des Algorithmus — —ToDo: Alternativen —



Um der Anforderung der Robustheit gerecht zu werden, muss das Protokoll eine *Fehlerkontrolle* besitzen. Diese *Fehlerkontrolle* prüft alle eingehenden PDUs hinsichtlich ihrer Integrität, wobei verschiedenste Verfahren zur Verfügung stehen (vgl. [1]). Bereits COBS stellt ein robustes Element dar, denn eine valide Nachricht darf kein Trennzeichen innerhalb der Nachricht enthalten und die dekodierte Nachricht muss genau der festgelegten Nachrichtenlänge entsprechen, um valide zu sein. Da COBS jedoch primär nicht der Fehlererkennung dient, wird zusätzlich noch ein Checksum-Verfahren Anwendung finden. Zahlreiche Verfahren sind erprobt und heute noch in Anwendung. Sie unterscheiden sich in der Regel in zwei Attributen. Zum einen sind sie unterschiedlich komplex in der Ausführung, zum anderen unterscheiden sie sich in der Erkennungsrate. Während simple Verfahren wie *Two's Complement Addition Checksum* zwar sehr effizient auch in eingebetteten Systemen einsetzbar sind, ist ihre Fehlererkennungsrate nicht hinreichend. Auf der anderen Seite existieren Verfahren wie der *cyclic redundancy check (CRC)*, die zwar eine hohe Fehlererkennungsrate aufweisen, jedoch ohne Hardwarebeschleunigung auf langsamen Prozessoren nur eingeschränkt nutzbar sind. Ein Vergleich dieser und weiterer Checksum Verfahren wurde in [3] angestellt. Wenn man von 16Bit-Checksums und Paketgrößen von 150 Byte ausgeht, unterscheiden sich die getesteten Verfahren bei zufälligen Bit-Errors wie in Tabelle I dargestellt.

TABELLE I  
VERGLEICH VERSCHIEDENER VERFAHREN ZUR BILDUNG EINER  
CHECKSUM (NACH [3])

Verfahren	Wahrscheinlichkeit eines unentdeckten Fehlers	Rechenaufwand im Vergleich
2's Complement Add-16	$10^{-6}$	100%
Fletcher-16	$10^{-12}$	200%
CRC-16 bound	$10^{-14}$	>400%

Es zeigt sich, dass die *Fletcher Checksum* einen guten Kompromiss aus der simplen *Add* und der komplexen *CRC* Checksum darstellt. Sie wird daher für das Protokoll vorgesehen. Die genaue Funktionsweise kann XYZ entnommen werden.

Auch *Fluss-Steuerung* stellt eine Protokollaufgabe dar, insbesondere wenn Puffer zur Anwendung kommen. Sowohl ein Empfangspuffer als auch ein Sendepuffer kann an Kapazitätsgrenzen stoßen, sodass ohne den Verlust von gespeicherten Daten keine weiteren Daten mehr aufgenommen werden können. Es existieren nach [1] zahlreiche Verfahren wie das Kredit-Verfahren, um auf eine solche Situation zu reagieren. Diese sind jedoch sehr komplex und sehen in der Regel eine dynamische Anpassung der Übertragungsrate des Sendepartners vor. Da jede PDU jedoch identische Informationen (zu unterschiedlichen Messpunkten) enthält, kann man davon ausgehen, dass das zuerst empfangene Paket die ältesten Daten enthält. Dieses wird daher verworfen und durch das aktuelle Paket ersetzt. In einem gut ausgelegten System sollte diese Situation jedoch gar nicht erst auftreten, denn die Übertragungsrate der PDUs

sollte immer im Verhältnis zur Verarbeitungsgeschwindigkeit stehen.

Die Aufgabe *Anpassen der PDU-Größen* wird im entwickelten Protokoll nicht unterstützt, da nur eine einzelne PDU-Definition existiert. Jede gesendete PDU enthält somit die gleiche Art von Informationen. Im Bild XY ist der prinzipielle Aufbau einer PDU veranschaulicht.

Text über den Aufbau den PDU. [GRAFIK MIT AUFBAU DER NACHRICHT]

Bisher unbeachtet war die unterste Schicht, also die physischen Schnittstelle zwischen beiden Mikrocontrollern. Aus der minimalen Transferrate von  $1,2 \frac{\text{Mbit}}{\text{s}}$  in der Anforderungsspezifikation lässt sich schnell ableiten, dass weder UART (Universal asynchronous receiver-transmitter) noch I2C (Inter-Integrated Circuit) schnell genug für eine solche Verbindung sind. Die zur Verfügung stehende SPI-Peripherie auf dem High-Level-Prozessor bietet jedoch theoretische Transferraten bis zu  $10 \frac{\text{Mbit}}{\text{s}}$ , was den Anforderungen entspricht. Sie kommt demnach zum Einsatz und wird in der folgenden Schichtendarstellung zusammengefasst mit den softwareseitigen Puffern.

Zum Abschluss wird nun die entstandene Schichtendarstellung in Bild 6 gezeigt. In dieser sind nun alle bisher beschriebenen Protokollfunktionen vereint.

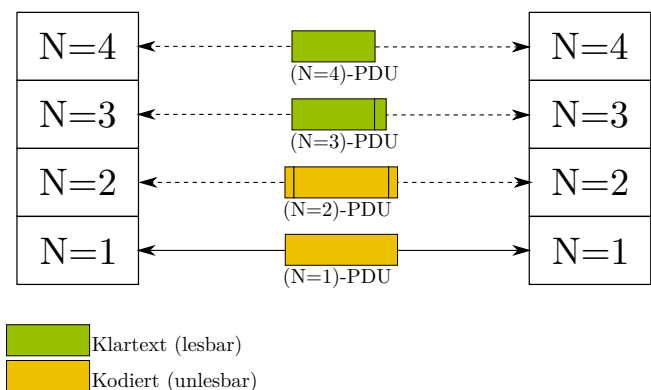


Bild 6. Schichtendarstellung des fertigen Protokolls (angelehnt an [1])

Bei der obersten Schicht N=4 handelt es sich um die *Anwendungsschicht*. In dieser Schicht liegen Nachrichten im Klartext vor und die Inhalte der Nachricht können nach Belieben durch den Anwender ausgelesen bzw. abgeändert werden. Bei Schicht N=3 handelt es sich um die *Datensicherungsschicht*. Wenn das Paket inhaltlich fertiggestellt ist, was nach Anforderungsspezifikation im 1kHz Takt geschieht, wird von der Anwendungsschicht in diese Schicht übergegangen. In ihr wird die Checksum generiert und an die (N=4)-PDU angehängt. Beim Empfang einer PDU wird an dieser Stelle die empfangene Checksum auf ihre Integrität untersucht. Nach dem Anhängen der Checksum wird die (N=3)-PDU durch den COBS Algorithmus in der Schicht N=2 (*De-/Encodierungsschicht*) encodiert. Die dadurch entstandene (N=2)-PDU besitzt nun das gültige Sendeformat. Im Falle eines Empfangs wird hier die kodierte PDU decodiert und liegt danach in Schicht N=3 als Klartext vor. Die unterste *Übertragungsschicht* N=1 ist nun

für Empfang und Absendung der Nachrichten über die SPI Schnittstelle zuständig.

#### D. Implementierung

Für die Implementierung muss nun der entstandene Protokollentwurf auf die Zielsysteme angepasst werden. Beide Microcontroller unterscheiden sich hauptsächlich in dem Aufbau ihrer SPI Peripherie. Der erste Schritt besteht daher aus der Implementierung von Treiberfunktionen. Es wurde sowohl eine Treibervariante auf Blocking-Basis als auch auf Interrupt-Basis und Ringspeicher entwickelt. Während die erste Variante den regulären Programmablauf blockiert währenddessen Daten gesendet oder empfangen werden, ist bei der zweiten Variante der Programmablauf weiterhin möglich und wird nur zu notwendigen Zeitpunkten kurz unterbrochen. Aus Gründen von Performance und der gewünschten Unabhängigkeit zwischen beiden Prozessoren, kommt die zweite Variante zur Anwendung. Diese ist Ringspeicher-basiert, daher interagiert das Hauptprogramm nur indirekt durch Auslesen und Beschreiben der Puffer mit der Peripherie.

Für den Anwender des Protokolls ist zunächst wichtig, wie man neue Pakete (PDUs) generiert und fertigstellt. Um die Schnittstelle zum Anwender möglichst komfortabel zu gestalten, kommt ein *c-struct* bzw. ein *c-union* zur Anwendung. In Code-Darstellung sieht die Umsetzung folgendermaßen aus:

```
1 typedef union {
2     struct{
3         /*** --- Protocol Header --- ***/
4         uint8_t startByte; //Status Flags
5         uint64_t timeStamp_IMU_Sensory;
6
7         /*** --- Sensory Data --- ***/
8
9         int angle_pitch;
10        int angle_roll;
11        int angle_yaw;
12
13        //[...]
14
15        /*** --- Protocol Footer --- ***/
16        uint16_t checksum;
17    }__attribute__((packed)) protocol_s;
18
19    uint8_t bytestream[num_sum];
20 }protocol_u;
```

Der Anwender greift in diesem *union* über den *struct* *protocol\_u* auf die einzelnen Datenfelder zu und kann das nächste zu sendende Paket somit manipulieren. Die Protokollfunktionen greifen über das Bytearray *bytestream* auf die PDU zu, interpretieren die Sensordaten also als Bytefolge. Ein Beispiel für eine solche Protokollfunktion ist die *generate\_Checksum*-Funktion, die auf alle Bytes zugreift und schließlich die errechnete Checksum in *checksum* (siehe Z. 16) abspeichert. Das Union sorgt dafür, dass *struct* und Bytearray auf den gleichen Speicherbereich zugreifen. Dabei ist unbedingt zu berücksichtigen, dass Compileroptimierungen zu einer Inkonsistenz zwischen *struct* und Bytearray führen können. Um die physischen Speicherzugriffe zu beschleunigen, fügt der Compiler Leerräume in das *struct* ein. Damit wird versucht, die Variablen so im Speicher auszurichten, dass die Ausrichtung mit der natürlichen Ausrichtung der physischen Speicherzugriffe übereinstimmt und der Zugriff möglichst schnell ablaufen kann. Diese

Optimierung wird als *padding* bezeichnet und ist in [5] noch wesentlich genauer ausgeführt. Um dies zu verhindern existieren verschiedene Techniken, die den Compiler an besagter Optimierung hindern. Im vorangegangenen Code sorgt bspw. *\_\_attribute\_\_((packed))* in Z. 17 für die Verhinderung der Optimierung. Eine Alternative stellt die Benutzung von *#pragma push()* vor der *struct*-Definition und *#pragma pop()* nach der Definition dar.

Nach der Fertigstellung der Nachricht (zu sendene Daten in *struct* geschrieben) kann nun über den Aufruf von *pack\_message()* dafür gesorgt werden, dass die Nachricht codiert und abgeschickt wird. Alle Änderungen am vorgestellten *struct*, die nach dem Aufruf dieser Methode getätigt werden, können erst für die nächste Nachricht berücksichtigt werden. Zum Lesen der Nachricht existiert eine ähnliche Funktion namens *unpack\_message()*.

In der von Astec bereitgestellten Programmstruktur wird folgende Funktion in der angestrebten Frequenz von 1KHz aufgerufen:

```
void mainloopSDK()
{
    //Reads messages from receive buffer (control data)
    unpack_message();

    //Forwards the currently read control data to the LLP
    apply_message_to_UAV();

    //Received sensory from the IMU (like angle_pitch)
    // is written into the next message
    update_message_with_IMU_Sensory();

    //Current version of message-struct gets packed
    // and sent to the Nucleo
    pack_message();
}
```

Die ersten zwei Funktionen dienen dem Auslesen des Empfangspuffers und des Weiterleitens der Steuerungsdaten an den *LLP*. Dafür wird der Puffer ausgelesen, der zuvor durch den Nucleo mit Steuerungsdaten gefüllt wurde. Eine wichtige Voraussetzung an die Methode *unpack\_message()*; lautet, dass im Falle eines Empfangsfehlers (bspw. wenn der Nucleo nicht verbunden ist) die Ausführung nicht in einen Deadlock-Zustand gerät und fortgesetzt wird.

Die Implementierung beim Kommunikationspartner (Nucleo) sieht dazu äquivalent aus:

```
while(1)
{
    //Reads messages from receive buffer (sensory data)
    unpack_message();

    /* --- Here the sensor-fusion is taking place --- */
    // Estimate control signals and store them inside
    // the PDU-struct

    // Current version of Control-PDU gets packed
    // and sent to the HLP
    pack_message();
}
```

(ToDo: Verweis auf GiRepo oder Code in Anhang?)

#### E. Protokolltest

Nach der erfolgreichen Entwicklung eines Protokolls muss dieses auf Leistungsfähigkeit und Fehler getestet werden. Bei

der Fehlererkennung ergibt sich dabei nach [?] das Problem, dass Fehler zwar nachgewiesen werden können, jedoch nicht das vollständige Nichtvorhanden sein von Fehlern. Der Nachweis eines restlos fehlerfreien Protokolls kann demnach nicht geführt werden, das Vertrauen in die Funktionalität und Verlässlichkeit jedoch gesteigert werden. Die zuvor genannte Literatur nennt prinzipiell vier Typen von Protokolltests, den

- *entwicklungsbegleitenden Test*,
- *Konformitätstest*,
- *Interoperabilitätstest* und
- *ergänzende Tests*.

Der *entwicklungsbegleitende Test* wird oft bereits während der Implementierung gestartet. Darin werden einzelne Teile der Implementierung auf ihre Funktionstüchtigkeit untersucht. Diese Tests erfordern eine detaillierte Kenntnis über die jeweilige Implementierung und werden daher oft vom Implementierer selbst durchgeführt. Ein Beispiel einer solchen Teilfunktion stellt die Generierung einer Checksum dar. Diese Funktion kann unter verschiedenen Eingabefolgen getestet werden, von denen das korrekte Ergebnis bekannt ist, und dann die entsprechenden Ausgaben überprüft werden. Von dieser Art Test wurde während der Implementierung des Protokolls ausgiebig gebraucht gemacht. Fehler wurden somit bereits frühzeitig erkannt und konnten direkt behoben werden. Im *Konformitätstest* wird überprüft, ob das äußere Verhalten des Protokolls der Entwurfsspezifikation entspricht. So wurde beispielsweise festgelegt, wie groß jede PDU zu sein hat bzw. wie diese codiert werden. Beides lässt sich durch ein Oszilloskop oder Logic Analyser überprüfen, obwohl dies stellenweise sehr aufwändig ist. Nach entsprechender Konfiguration kann damit an den Ausgängen der seriellen Schnittstelle (im vorliegenden Fall SPI) die gesendete Nachricht auf ihre Korrektheit überprüft werden. Auf der anderen Seite wird ebenfalls überprüft, ob das System korrekt auf definierte eingehende Nachrichten reagiert. Aufgrund des hohen Aufwands wurde dieser Test nur sehr eingeschränkt durchgeführt.

Häufig ist die jeweilige Implementierung des Protokolls vom jeweiligen Zielsystem abhängig. So müssen bspw. Treiberfunktionen abhängig von den verbauten Peripheriekomponenten programmiert werden. Durch den *Interoperabilitätstest* wird überprüft ob unterschiedliche Implementierungen auch miteinander kompatibel sind. Dieser Test kann im vorliegenden Fall schnell durchgeführt werden, da es nur zwei unterschiedliche Implementierungen (für beide Microcontroller) gibt.

Zuletzt erfolgen noch *ergänzende Tests*, die hier etwas ausführlicher ausgeführt werden. Genannte Literatur unterscheidet in *Leistungstests* und *Robustheitstests*. Ersterer Test dient der Überprüfung der Anforderungen an Durchsatz. In letzterem Test wird die Robustheit des Protokolls gegenüber fehlerhaften Übertragungen untersucht. Für die Durchführung dieser Tests ist von großer Bedeutung, dass Rahmenbedingungen gewählt und eingehalten werden. Nur so wird Realitätsnähe und Vergleichbarkeit sichergestellt. Für den vorliegenden Fall bedeutet dies in erster Linie den Hexakopter in einen definierten Zustand zu versetzen. Dazu gehört zunächst, dass die gesamte Sensorik korrekt kalibriert ist. Nur so kann verhindert

werden, dass die Testergebnisse aufgrund einer Rekalibrierung während der Testlaufzeit verfälscht werden. Wichtig ist ebenso, dass der *HLP* während der Testlaufzeit vom Debugging-Werkzeug getrennt ist. Nur so arbeitet der Prozessor in seiner vorgesehenen Geschwindigkeit. Von enormer Wichtigkeit ist ebenfalls, dass jederzeit eine Verbindung zur Fernsteuerung besteht. Ist dies nicht gegeben, verfällt der *HLP* in regelmäßigen Abständen in einen wartenden Zustand, der die reguläre Programmausführung verzögert. Die Einschränkung der nicht verwendbaren Debugging Schnittstelle am *HLP* stellt eine besondere Herausforderung dar, denn es muss eine alternative Möglichkeit gefunden werden, die Testergebnisse im *HLP* an den Tester zu übermitteln. Dieses Problem wird dadurch gelöst, dass die Testergebnisse neben einigen Testwerten der PDU angehängt werden, die dem Nucleo gesendet wird. Somit sind die Testergebnisse beider Prozessoren über den Debugger des Nucleos entnehmbar.

Einige Protokoll- und Übertragungsparameter sind zum Testzeitpunkt noch nicht final festgelegt und tauchen somit als variable Größe während der Tests auf. Dazu zählen

- SPI-Übertragungsgeschwindigkeit (in Bit/Sekunde),
- Nachrichtenrate (in Anzahl zu sendender Nachrichten pro Sekunde) und
- Paketgröße (in Bytes).

Die Testmethode sieht zunächst folgendermaßen aus: Sowohl *HLP* als auch *Nucleo* senden fest definierte Pakete. Die Testmethode wurde im späteren Verlauf noch insofern weiterentwickelt, dass sich die Antwortpakete aus den Daten der eingehenden Pakete errechnen. Jeder Empfang einer Nachricht wird auf beiden Seiten dokumentiert. Dabei können vier verschiedene Situationen eintreten:

- 1) Protokoll akzeptiert Daten und diese entsprechen dem abgesendeten Paket
- 2) Protokoll erkennt einen Übertragungsfehler, der auch wirklich vorliegt
- 3) Protokoll akzeptiert Daten, diese entsprechen allerdings nicht dem abgesendeten Paket
- 4) Protokoll erkennt irrtümlich einen Übertragungsfehler, obwohl die empfangende Nachricht korrekt war

Zur Bestimmung des Durchsatzes ist primär die erste Situation von Interesse. Sie entspricht dem erfolgreichen Empfang eines Pakets und tritt im Idealfall immer auf. Für die Bestimmung der Robustheit sind primär die übrigen drei Situation relevant. Als besonders kritisch wird der dritte Fall angesehen. Eine unentdeckte Fehlübertragung kann zum sofortigen Absturz führen, da fehlerhafte Daten weiterverarbeitet werden. Diese Situation tritt dann auf, wenn das Checksum-Verfahren versagt hat.

Die erzielten Testresultate können Bild 7 entnommen werden.

ToDo: INTERPRETATION -> PARAMETERWAHL

Ein weiterer Test wurde durchgeführt, um speziell die Robustheit des Protokolls zu testen. Ob korrekt auf Fehlübertragungen reagiert wird, wird bereits im vorherigen Test untersucht. Darüber hinaus muss nach [?] jedoch noch auf sogenannte Deadlock- und Livelock-Freiheit getestet werden. Ein Deadlock beschreibt dabei einen Zustand, der sich nie

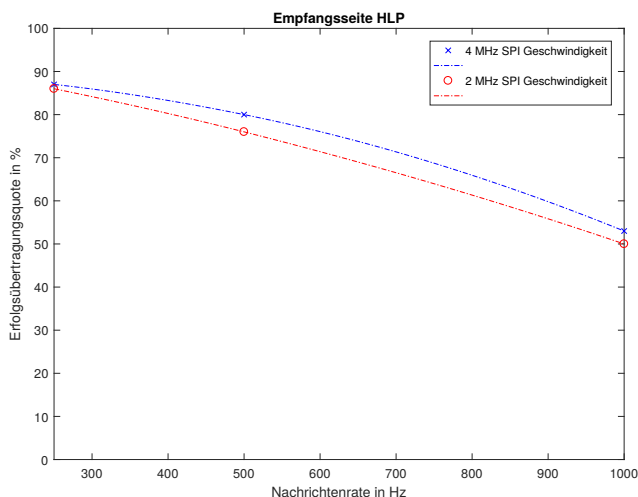
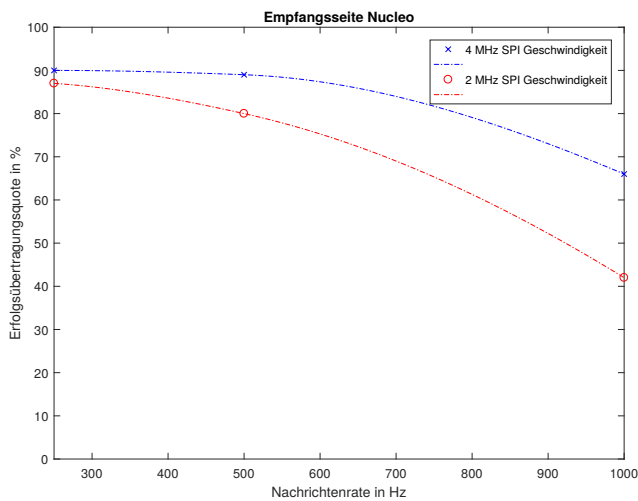


Bild 7.

wieder ändern wird. Das Programm wird demnach nicht mehr korrekt ausgeführt, was unbedingt vermieden werden muss. Als Livelock wird ein Zustand beschrieben, in dem ein endloser Zyklus ausgeführt wird, der keine produktiven Aktionen beinhaltet. Um auf diese Freiheit zu testen, wird ein mehrstündiger regulärer Datenaustausch initiiert. Am Ende des Tests wird geprüft, ob nach wie vor ein funktionierender Datentransfer vorliegt, oder ob sich einer der beiden genannten Zustände eingestellt hat. Ein 5 stündiger Test lief dabei erfolgreich ab.

#### IV. ZUSAMMENFASSUNG

Hier die wichtigsten Ergebnisse der Arbeit in 5-10 Sätzen zusammenfassen. Dies sollte keine Wiederholung des Abstracts oder der Einführung sein. Insbesondere kann hier ein

Ausblick auf zukünftige Arbeiten gegeben werden. -Ausblick: Sensor Fusion, Protokoll weiterentwickeln, Successrate steigern.

#### ANHANG I OPTIONALER TITEL

Anhang eins.

#### ANHANG II

Anhang zwei.

#### DANKSAGUNG

Wenn ihr jemanden danken wollt, der Euch bei der Arbeit besonders unterstützt hat (Korrekturlesen, fachliche Hinweise,...), dann ist hier der dafür vorgesehene Platz.

#### LITERATURVERZEICHNIS

- [1] H. König, *Protocol Engineering*, 1st ed. Wiesbaden, Deutschland: Teubner, 2003.
- [2] Echtzeitsysteme
- [3] The Effectiveness of Checksums for Embedded Control Networks
- [4] COBS Paper
- [5] padding Theroie



**Malte Markus Breitenbach** Malte Markus Breitenbach (B.Sc.) wurde am am 23.07.95 in Gelnhausen geboren. Seinen Bachelortitel erlangte er 2018 an der Technischen Universität Darmstadt im Studiengang Mechatronik.

**Autor B** Biographie Autor B.



**Autor C** Biographie Autor C.

