Arbeitstitel: Eingliederung einer zusätzlichen Hardwarekomponetene 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 Hexacopter 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 Hexakopter gegeben ist. Ein typisches Beispiel 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 Rechenoperatoren 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 Hexacopter entsprechend geregelt werden.

Somit soll durch diese Arbeit eine solide Grundlage geschaffen werden, um darauf aufbauend eine Regelung für den Hexacopter 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. Systembeschreigung

Subsection text.

B. Serial Peripheral Interface (SPI)

Subsubsection text.

C. Protokolltheorie

Um eine Kommunikation zwischen den zuvor erwähnten Microcontrollern zu ermöglichen, bedarf es eines gut geplaneten sowie gut umgesetzen Kommunikationsprotokolls.

Ein Kommunikationsprotokol ist nach [1] nichts anderes als eine Verhaltenskonvention zwischen zwei Kommunikationspartern (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 1 veranschaulicht die Generierung einer PDU über mehrere Schichten beispielhaft. Dabei soll deutlich werden, dass ein Datentransfer ausßlichlich 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.

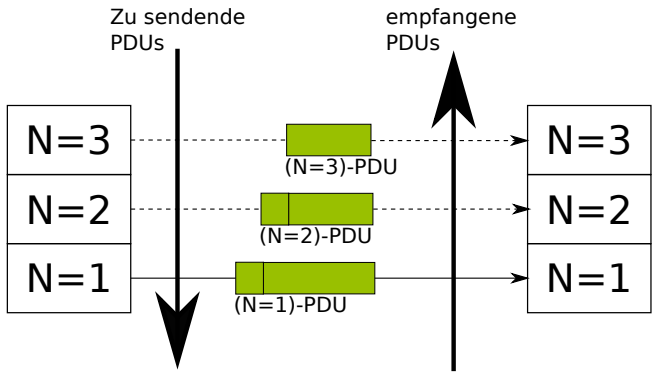

Bild 1. Bildung von PDUs in der Schichtendarstellung (angelehnt an [1])

III. Entwicklung und Vorstellung des Protokolls

A. Vorgehensweise

Wie in [1] aufgeführt kommt zur Entwicklung des Protokolls ein einfaches Wasserfall-Modell zur Anwendung. Bild 2 zeigt den groben Verlauf dieses Entwicklungsprozesses. Prinzipell 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.

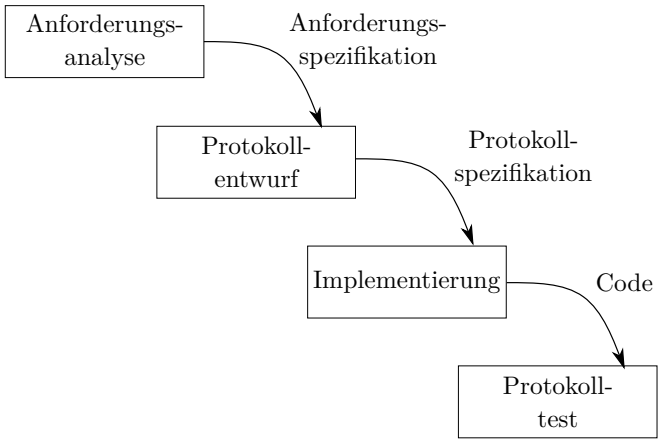

Bild 2. Wasserfall-Modell des Protokollentwicklungsprozesses (angelehnt an [1])

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 verschiednene 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.

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 Transfergeschwindkeit, ein hohes Maß an Robustheit als auch Echtzeitfähigkeit. Die Transfergeschwindigkeit 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 unnedingt zu verhindern. Weiterhin sollte auch 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 verabeitet werden und auch verfügbar sind. Orientiert an den Gegenheiten des Standardprogramms auf dem High-Level-Prozessor wird zunächst eine Paketgeschwindigkeit von 1000 PDUs pro Sekunde angestrebt. Daraus lässt sich die Mindestransfergeschwindkeit 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

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

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

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

Als zusätzliche Anforderung ist noch ein möglicher 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. Diese Transferrate ist weder mit UART noch mit I2C (fast mode) erreichbar, daher fällt die Wahl auf das bereits eingeführte SPI.

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 Kommunikationsparter in einen definierten Zustand zu bringen (bspw. beim Verbindungsaufbau oder -abbau). Eine einfache Methode stellt der 2-Wege-Handshake dar, wobei

der 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 Ressourchen 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 Exktraktion einer Nachricht aus einem Puffer ist jedoch nur schwer möglich, wenn weder Anfang noch Ende der Nachticht klar ersichtlich sind. Durch die Einführung eines sogenannten Trennzeichens (zu engl. delimiter) lässt sich diese 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 (zu englisch: delimiter), 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 (vlg. [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 aufweist, 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 ein solche Situation zu reagieren. Diese sind jedoch sehr komplex und sehen in der Regel eine dynamische Anpassung der Übertragungsrate des Sendeparters 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 garnicht 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 unterstüzt, da nur eine einzlene PDU-Definition existiert. Jede gesendete PDU enthält somit die gleiche Art von Informationen. Im Bild XY ist der prinzipielle Aufbau einer PDU veranschaulicht.

BlaBla über den Aufbau den PDU.

Bisher unbeachtet war die unterste Schicht, also die physischen Schnittstelle zwischen beiden Mikrocontrollern. Aus der minimalen Transfergeschwindigkeit 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 3 gezeigt. In dieser sind nun alle bisher beschriebenen Protokollfunktionen vereint.

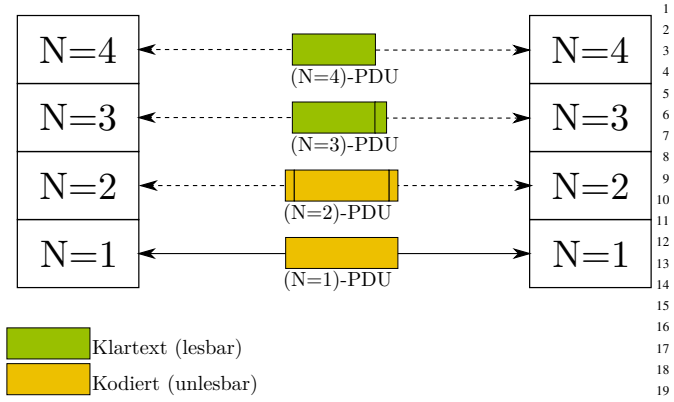

Bild 3. Schichtendarstellung des fertigen Protokolls (angelehnt an [1])

Bei der obersten Schicht N=4 handelt es sich um die *Anwendungsschicht*. In dieser Schicht liegen Nachrichten im Klartext vor und die Inhalte der Nachticht 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 während 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 konfortabel 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 Inkostistenz 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 pyhsischen 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 Asctec bereitsgestellten Programmstruktur wird folgende Funktion in der angetrebten Frequenz von 1kHz aufgerufen:

```
1
2  void mainloopSDK()
3  {
4    //Reads messages from receive buffer (control data)
5    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. Eine wichtige Vorraussetzung 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 ganz regulär fortgesetzt wird.

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

E. Benchmarks

LiveLook, etc...

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.