Bachelorarbeit

Ulrich ter Horst

Ein generischer CANaerospace-Knoten für redundante UAS-Architekturen basierend auf MicroPython

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Ulrich ter Horst

Ein generischer CANaerospace-Knoten für redundante UAS-Architekturen basierend auf MicroPython

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Bachelor of Science Technische Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Stephan Schulz
Zweitgutachter: Prof. Dr. Franz Korf

Eingereicht am: 19. Januar 2018

Ulrich ter Horst

Thema der Arbeit
Ein generischer CANaerospace-Knoten für redundante UAS-Architekturen basierend auf MicroPython

Stichworte
CAN, CANaerospace, MicroPython, Python, Feldbussysteme, Eingebettete Systeme, UAS, UAV, STM32

Kurzzusammenfassung
Diese Arbeit befasst sich mit der Entwicklung eines CAN-Bus Knotens für redundante Systemarchitekturen im Bereich unbemannter Flugsysteme. Ausgangspunkt ist eine Systemarchitektur mit zwei unabhängigen CAN-Bussen, welche alle Systemkomponenten mit dem CANaerospace-Protokoll verbinden. Der Kern dieser Arbeit besteht aus der Konzeption und der technischen Umsetzung auf Basis von STM32-Mikrocontrollern und MicroPython. Es wird die Eignung von CANaerospace und MicroPython für den Anwendungsfall validiert.

Ulrich ter Horst

Title of the paper
A generic CANaerospace node for redundant UAS architectures based on MicroPython

Keywords
CAN, CANaerospace, MicroPython, Python, Fieldbus, Embedded Systems, UAS, UAV, STM32

Abstract
This thesis is about the development of a CAN-bus node for redundant flight system architectures for unmanned aircraft systems. Origin of this work is a system architecture with two independent CAN busses, which connect all system compontens via CANaerospace protocol. The concept and technological implementation based on STM32-microcontrollers and MicroPython forms the core of this thesis. The suitability of CANaerospace and MicroPython for this use case is validated.

Inhaltsverzeichnis

Abbildungsverzeichnis

Abkürzungen

AMP Arbitration on Message Priority

API Application Programming Interface

ASCII American Standard Code for Information Interchange

BCU Bus Converter Unit

BLDC Brushless Direct Current

CAN Controller Area Network

CANas CANaerospace

CAU Control Service Allocator Unit

CD Collision Detection

COB Communication Object

COTS Commercial Off-The-Shelf

CR Collision Resolution

CRC Cyclic Redundancy Check

CSMA Carrier Sense Multiple Access

DFU Device Firmware Upgrade

DIP Dual In-Line Package

DMA Direct Memory Access

ECU Electronic Control Unit

ESC Electronic Speed Controller

FIFO First In First Out

GNSS Global Navigation Satellite System

GPIO General Purpose Input Output

GUI Graphical User Interface

HAL Hardware Abstraction Layer

HLP Higher Layer Protocol

I²C Inter-Integrated Circuit

IDS Identification Service

IMU Inertial Measurement Unit

IoT Internet of Things

LCC Logical Communication Channel

LED Light Emitting Diode

MISO Master In Slave Out

NOD Normal Operation Data

OPAV Optronik und Avionik

OSI Open Systems Interconnection

PC Personal Computer

PCB Printed Circuit Board

PDO Process Data Object

PWM Pulsweitenmodulation

RAM Random Memory Access

RC Radio Control

REC Receive Error Count

REPL Read-Eval-Print Loop

ROM Read Only Memory

rPDO receive Process Data Object

RTR Remote Transmission Request

SAFRAN Sicherheitskonforme (S)ystem(a)rchitektur von (F)lug(r)obotern und lebenszyklusgerechtes Komponentendesign für zivile (An)wendungen

SDIO Secure Digital Input Output

SLAM Simultaneous Localization And Mapping

SMD Surface Mount Device

SPI Serial Peripheral Interface

SUAV Small Unmanned Aerial Vehicle

TEC Transmit Error Count

tPDO transmit Process Data Object

UART Universal Asynchronous Receiver Transmitter

UAS Unmanned Aircraft System

UAV Unmanned Aerial Vehicle

USB Universal Serial Bus

ZIM Zentrales Innovationsprogramm Mittelstand

µC Mikrocontroller

µP MicroPython

1. Einleitung

1.1. Organisatorischer Kontext

Die vorliegende Arbeit greift Fragestellungen aus dem Forschungsprojekt Sicherheitskonforme Systemarchitektur von Flugrobotern und lebenszyklusgerechtes Komponentendesign für zivile Anwendungen (SAFRAN) auf, das innerhalb der Förderlinie Zentrales Innovationsprogramm Mittelstand (ZIM) beantragt wurde. Im Rahmen des Projekts wurde eine Systemarchitektur für SUAV entwickelt, die durch Redundanz und ein spezielles Systemdesign ein besonders hohes Maß an Zuverlässigkeit aufweist.

1.2. Aufgabenstellung

Es soll eine generische Lösung für die Anbindung von Systemkomponenten an eine redundante Flugsystemarchitektur für Small Unmanned Aerial Vehicles (SUAVs) erarbeitet werden. Sowohl für die Sensorik, als auch für die Aktorik eines SUAV kommen verschiedene Hardware-Schnittstellen in Betracht, die für eine gemeinsame Kommunikation an ein einheitliches Bussystem angeschlossen werden müssen.

Barometer, elektronische Kompasse, Gyroskope, Accelerometer, Pitotrohre (Staudrucksonden), Global Navigation Satellite Systems (GNSSs), Ulraschall-Entfernungsmesser und Electronic Speed Controllers (ESCs) können in SUAVs verbaut sein und für deren Anbindungen werden je nach Bauteil und Hersteller verschiedene Schnittstellen genutzt. Auch die Ansteuerung jeder Komponente muss gemäß der Herstellerangaben individuell implementiert werden können.

Somit sind die beiden Hauptaufgaben, der zu entwickelnden Einheit, das Ansteuern der angeschlossenen Komponenten und die Schnittstellenkonvertierung zwischen den Komponentenschnittstellen und des Bussystems. Die Flugsystemarchitektur gibt für die systemweite Kommunikation einen redundanten CAN-Bus mit dem Protokoll CANaerospace vor. Daher wird diese Einheit in der Systemarchitektur und in diesem Dokument nachfolgend als Bus Converter Unit (BCU) bezeichnet.

Beim Anschluss von Sensorik soll die BCU zusätzlich in der Lage sein, eine Rohdatenverarbeitung zur Reduktion der Buslast vorzunehmen. Dies kann beispielsweise ein Tiefpassfilter für Sensorwerte oder eine Sensorfusion sein. Auch der Betrieb mehrerer Sensoren an einer BCU soll möglich sein.

Diese Anforderungen verlangen es, dem Benutzer eine Möglichkeit bereitzustellen, das Verhalten der BCU zu definieren. Er soll in der Lage sein, mit möglichst wenig Einarbeitungsaufwand und in kurzer Zeit den Schnittstellenkonverter auf seinen Anwendungsfall zu spezialisieren. Nach Aufgabenstellung soll jedoch die Firmware dabei unverändert bleiben.

Die BCU muss daher über eine Art von Skript-Fähigkeit verfügen und ermöglicht dadurch eine Steuerung von komplexen Vorgängen mit nur wenigen Busnachrichten. In Kombination mit der Buslastreduktion aus der Rohdatenverarbeitung soll so die Anzahl möglicher Systemkomponenten an dem Bussystem maximiert werden.

Die Stromversorgung der angeschlossenen Komponenten ist ebenfalls Aufgabe der BCU. Des Weiteren soll es möglich sein, Komponenten via Skript von der Stromversorgung zu trennen, falls diese außer Kontrolle geraten oder um deren Energieverbrauch einzusparen.

2. Projektbeschreibung

Die Anforderungen, der in dieser Arbeit zu entwickelnden Lösungsansätze, werden durch den bestehenden Flugsystemarchitekturentwurf definiert. Die Kernaufgabe der geforderten Lösung ist es, andere Komponenten einer Flugsystemarchitektur einheitlich via CANaerospace (CANas) zu verbinden.

Dieses Kapitel zeigt im ersten Abschnitt einige Vorgaben der Systemarchitektur aus dem Forschungsprojekt SAFRAN auf und entwickelt diese weiter.

Zum besseren Verständnis wird die Architektur im darauf folgenden Abschnitt 2.1 in vereinfachter Form anhand seiner wichtigsten Systemkomponenten vorgestellt.

Der letzte Abschnitt des Kapitels zeigt die Schnittstellenauslegung der BCU auf und erläutert das breite Einsatzspektrum, welches sich aus dem erarbeiteten Konzept ergibt.

2.1. Vorgaben der Systemarchitektur

Die Eignung von CAN-Bus und Busprotokoll als Vernetzung der Komponenten innerhalb der Systemarchitektur zu bewerten, ist Teil dieser Arbeit.

Als systemweiter Kommunikationsweg zwischen den einzelnen Komponenten der Flugsystemarchitektur ist das Feldbussystem CAN vorgegeben. Auch der Redundanzgrad wurde durch zwei unabhängige CAN-Busse festgelegt.

Der Standardfall einer Umsetzung der Systemarchitektur ist in Abbildung 2.1 vereinfacht dargestellt. Die verschiedenen Systemkomponenten sind in den Armen des gezeigten Octocopters angeordnet. Jeder Arm verfügt über die gleiche Komponenten-Konfiguration. Alle Arme sind über einen doppelt ausgeführten (redundanten) CAN-Bus miteinander verbunden. Konventionelle Multirotor SUAVs verfügen über drei bis zehn Arme. So ergibt sich die nötige Redundanz in der Flugsystemarchitektur.

Die echtzeitfähige und zuverlässige Bus-Kommunikationslösung aus dieser Arbeit ist essenziell für eine Flugsystemarchitektur entsprechend dem Projekt SAFRAN.

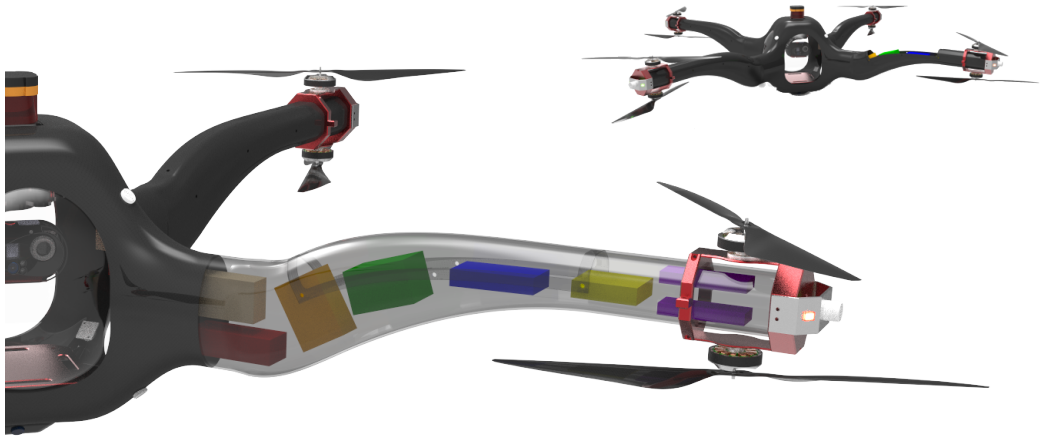


Abbildung 2.1.: CAD Grafik eines SUAV mit 4 Auslegern und 8 Motoren - Umsetzung der redundanten Flugsystemarchitektur von SAFRAN [10]

2.2. Komponenten der Systemarchitektur

Durch den Einsatz eines generischen Buskonverters zwischen Komponente und CANas-Bus wird das Spektrum einzusetzender Komponenten maßgeblich erweitert, weil die Wahl der Schnittstelle flexibel getroffen werden kann und das Übertragungsprotokoll frei wählbar ist. Auch preisgünstigere Komponenten sind möglich, weil beim Hersteller die CANas-Protokollimplementierung entfällt.

Die folgende Liste schlüsselt die Komponentenkürzel, welche auch in Abbildung 2.2 verwendet werden, auf und beschreibt die grundlegenden Aufgaben der Komponenten. Die Auflistung vermittelt einen Eindruck über die Diversität der Komponenten und die Anforderungen an die generische Kommunikationslösung. Die Flight Control Unit (FCU) unterscheidet sich beispielsweise in der Relevanz für einen sicheren Flug stark von der Data Recording Unit (DRU), welche keinen Einfluss aktiv auf den Flug ausübt.

Unabhängig von der Relevanz und der daraus abzuleitenden Nachrichtenpriorität einer Komponente gibt es starke Unterschiede in der jeweiligen Buslast. Während die FCU, CAU und RCU jeweils mindestens die Sollwerte für drei Freiheitsgrade (Roll, Pitch, Yaw) in hoher Frequenz (z. B. 50Hz) über den Bus verbreiten müssen, versendet die PMU die Spannungen der Akkuzellen nur niederfrequent (z. B. 0,1Hz):

- **FCU** - Flight Control Unit
  - Bahnplanung: Es muss für das Abfliegen von Wegpunkten immer ein Kurs zur aktuellen Position berechnet werden.
  - Fluglage regeln: Dem aktuellen Kurs entsprechend ist die Fluglage in allen drei Achsen (Roll, Pitch, Yaw) zu halten.
  - Ausgabe von Steuerbefehlen: Um den Kurs zu halten, werden Soll-Drehraten für alle drei Achsen ausgegeben.
- **SHU** - System Health Unit
  - Überwachung: Es ist die Anwesenheit aller Komponenten auf dem Bus zu überwachen.
  - Plausibilitätsprüfung von Sensordaten: Die Ausgaben redundanter Sensorik kann verglichen werden.
  - Redundanzverwaltung: Redundante Informationen auf dem Bus sind zu filtern bzw. zu priorisieren.
  - Einleiten von Notfallmaßnahmen: Die SHU ist der Entscheidungsträger bei Missionsabbrüchen und Notlandungen.
- **CAU** - Control Service Allocator Unit
  - Die CAU konvertiert abstrahierte Steuersignale der FCU zu Signalen für die UAV-Aktorik (Motordrehzahlen, Ruderflächenwinkel, etc.).
- **MCU** - Motor Control Unit
  - Ansteuerung von ESCs: Die Soll-Drehzahlen der Antriebe müssen in Signale für die Aktorik umgewandelt werden. Bei Modellbau ESCs wären das PWM-Signale.
  - Falls möglich, soll die MCU auch die Motordrehzahl und die Temperaturen von Motor und ESC überwachen.
- **PMU** - Power Management Unit
  - Überwachen der Akkuspannung: Für die Flugsicherheit empfiehlt es sich, die Restflugzeit über Integration der Stromstärke oder über die aktuelle Akkuspannung abzuschätzen.

- **RCU** - Radio Control Unit
  - Auslesen der Radio Control (RC)-Signale (Pilot-Eingaben): Zur manuellen Steuerung muss der Empfänger einer Funkverbindung ausgelesen werden. Oft werden Modellbau Empfänger in Unmanned Aerial Vehicles (UAVs) eingesetzt, unabhängig von der Telemetrie.
- **FSM** - Flight Sensors Module
  - Flugzustand ermitteln: Das Auslesen sämtlicher Flugdaten-Sensoren (IMU, GNSS, Barometer, Magnetometer) ist Aufgabe der FSM.
  - Sensorfusion: Sensordaten verschiedener physikalischer Größen und Messverfahren werden in Kombination interpretiert.
- **USU** - Ultra Sonic Unit
  - Im Nahbereich lassen sich mit Ulrtraschall-Entfernungsmessern Kollisionen vermeiden und Landungsmanöver unterstützen.
- **TU** - Telemetry Unit
  - Kommunikation: Über die TU kann zwischen einer Bodenstation und dem UAV kommuniziert werden (Echtzeitflugdaten, Missionsdaten, Systemkonfiguration).
- **DRU** - Data Recording Unit
  - Flugdatenschreiber: Bei der Absturzanalyse oder der weiteren Entwicklung des UAV wird eine DRU eingesetzt, um beliebige Daten während des Betriebes aufzuzeichnen.

Die Gesamtbuslast durch zyklische Nachrichten muss unterhalb eines bestimmten Wertes liegen, um noch genügend Buslaufzeit für nicht-zyklisch versendete Nachrichten aufbringen zu können. Die CANas-Spezifikation empfiehlt eine Buslastgrenze von 80 % einzuhalten, vgl. [20], Abschnitt 6.3.

Die Priorität ist bei der Zuordnung zwischen CAN-Identifier und Information relevant, da der Zahlenwert des Identifiers durch die bitweise Arbitrierung die Priorität, bzw. Durchsetzungsfähigkeit der Nachricht auf dem Bus vorgibt.

Einige der oben gelisteten Komponenten kann man theoretisch auf der BCU-Platine selbst als Skript umsetzen. Diese Komponenten werden auf Abbildung 2.2 als hellblaues Achteck dargestellt.

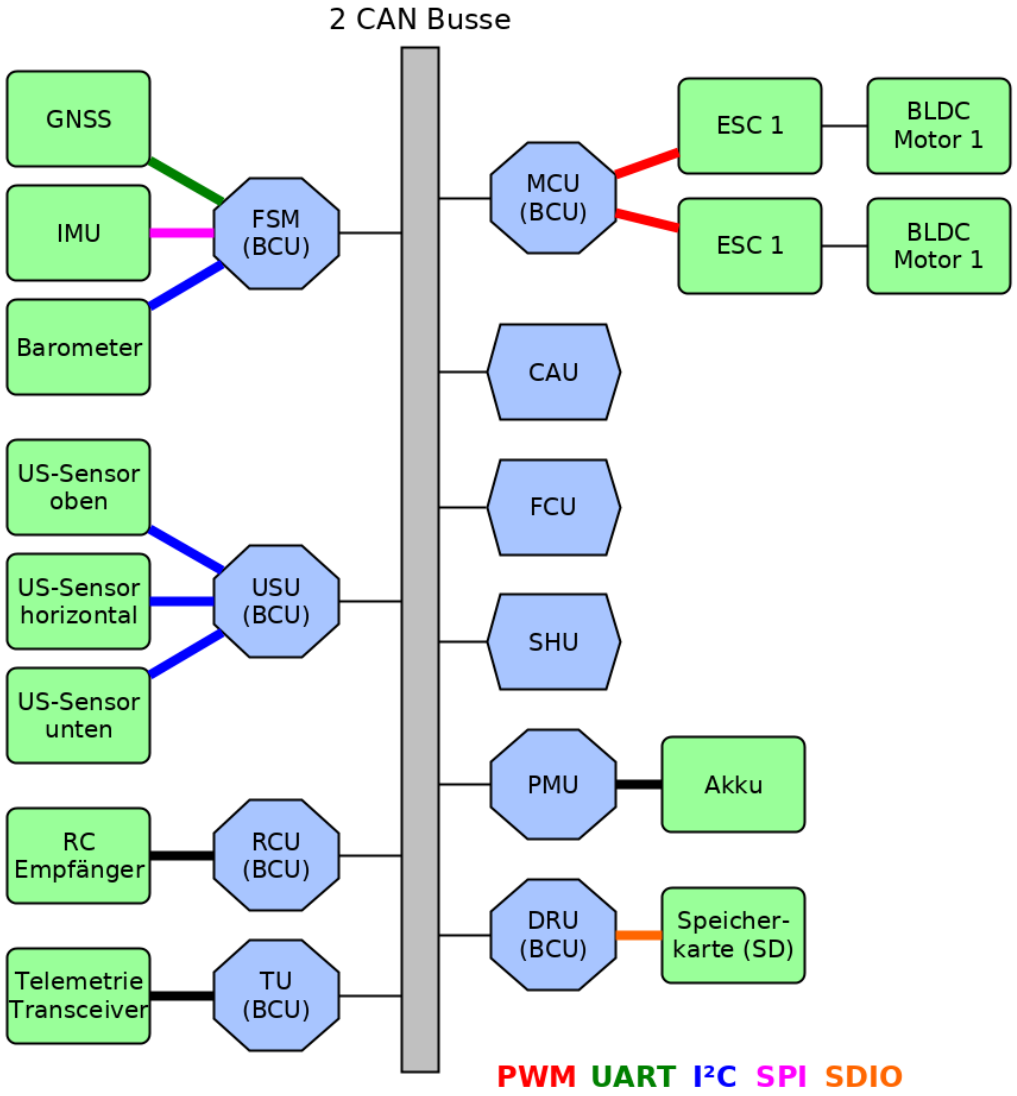


Abbildung 2.2.: Schema ([25]) eines Multicopter-Armes auf Basis einer möglichen Systemarchitektur nach SAFRAN unter Verwendung des Buskonverters (BCU)

Die sechseckig dargestellten Komponenten CAU, FCU und SHU sind zunächst als eigenständige Platine mit eigener CAN-Schnittstelle im Projekt SAFRAN geplant, jedoch könnte es möglich sein, deren Funktion auch als Skript in der BCU zu implementieren. Dieser Ansatz hängt stark von der Performanz der BCU ab und wäre im Nachgang dieser Arbeit noch auf Umsetzbarkeit hin zu evaluieren.

Die Grafik 2.2 zeigt die Umsetzung der Systemarchitektur innerhalb eines SUAV-Armes als Schema. Für ein Multicoptersystem, wie es auf Abb. 2.1 dargestellt ist, muss man sich das Schema also vierfach vorstellen, mit einem alles verbindenden CAN-Buspaar (grau).

Zweck der Abbildung 2.2 ist die Verdeutlichung des generischen Einsatzes der BCU im SUAV. Jede Komponente kann durch die BCU in das System integriert oder, wie zuvor erwähnt wurde, ggf. sogar durch die BCU-Platine allein umgesetzt werden.

Außerdem ist zu erkennen, dass die BCU auch mehrere Komponenten gleichzeitig an den Bus anbinden kann. Dies ist beispielsweise beim FSM, der MCU und der USU ersichtlich. Die Anzahl der integrierbaren Komponenten wird also nicht nur durch die Datenreduktion erhöht, sondern auch durch den Anschluss mehrerer Komponenten an eine BCU. Die maximale Anzahl von Busteilnehmern hängt von der Bustopologie, dessen elektrischen Eigenschaften und der verwendeten Bitrate ab. Das verwendete Protkokoll CANaerospace kann theoretisch bis zu 255 Teilnehmer adressieren.

2.3. Anbindung der Systemkomponenten

Im Rahmen des Flugsystemarchitekturentwurfs wurde untersucht, welche Schnittstellen auf Basis existierender Komponenten am weitesten verbreitet sind. Daraus wurde eine Vorgabe für die anzubietenden Schnittstellen der BCU abgeleitet.

Zu implementieren sind Anbindungen für zwei Universal Asynchronous Receiver Transmitters (UARTs), ein Serial Peripheral Interface (SPI), ein Inter-Integrated Circuit (I^2C) und eine Möglichkeit zur digitalen Ein- und Ausgabe, sowie einer Ein- und Ausgabe von Signalen mit Pulsweitenmodulation (PWM).

Detaillierte Anforderungen bezüglich der Schnittstellen und der Hardware allgemein finden sich in Abschnitt 2.3 und in Kapitel 5.3.

Die Systemarchitektur sieht als Kompromiss zwischen Komplexität der BCU und der Flexibilität beim Anschluss von Komponenten folgende Schnittstellen an der BCU vor:

- 2 $\times$ UART

- 1 $\times$ I²C
- 1 $\times$ SPI mit 2 Chip Select Leitungen
- 8 Pins, die konfigurierbar sind für: PWM Ein-/Ausgabe oder Digital Ein-/Ausgabe
- abschaltbare Versorgungsspannungen an jeder Schnittstelle

Die Anordnung der Schnittstellen ist auf Abbildung 2.3 zu erkennen. Die Aufnahme zeigt das noch unbestückte Printed Circuit Board (PCB) der BCU. An den äußeren Rändern der Platine befinden sich die Schnittstellen, dort sind Buchsen für Wannenstecker aufzulöten.

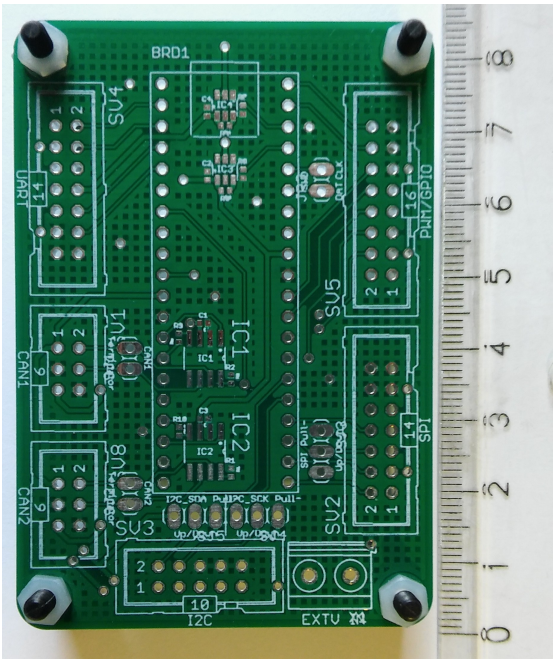


Abbildung 2.3.: Adapterplatine für Mini-M4 unbestückt [22], zu sehen ist die Anordnung der Schnittstellen in Form von Wannensteckern und einer Schraubklemme

Unten links befinden sich die beiden CAN-Schnittstellen und die Eingänge für die Versorgungsspannungen (3 VDC u. 5 VDC). Über Load-Switches werden die beiden Spannungen und eine variable dritte Spannungsquelle (max. 24 VDC) an die anderen Schnittstellen weitergeleitet.

Durch die Fähigkeit mehrere Komponenten parallel anzubinden, erweitert sich auch das Einsatzfeld der BCU. Als rekonfigurierbarer und teilautonomer Buskonverter weist sie ebenfalls Eigenschaften eines Sensor-Aktor-Knotens auf.

Die MCU auf Abb. 2.2 bindet beispielsweise zwei Motoren als Aktorik an. Wenn sie zusätzlich Sensorik zur Erfassung der Motortemperaturen oder der Ist-Drehzahl erhält, ergeben sich erweiterte Use-Cases. Wird ein Motor zu heiß oder verliert er an Drehzahl, kann die MCU diesen selbstständig entlasten und den Schubverlust beim anderen Motor durch erhöhte Drehzahl kompensieren.

Warnungen über solche Ereignisse können auch selbstständig via CAN versendet werden. Beim Beispiel der MCU in einem Multicoptersystem, wie es auf Abb. 2.1 dargestellt ist, ist eine Mitteilung an die Flugsteuerung sinnvoll, da die Motoren von den Sollwerten abweichen würden und somit ein verändertes Drehmoment auf die Z-Achse (Yaw) des SUAV ausüben würden.

3. Grundlagen

Dieses Kapitel fasst die theoretischen Grundlagen und Eigenschaften der verwendeten Technologien zusammen. Nach einer kurzen Einführung des Feldbussystems Controller Area Network (CAN) wird die Gruppe der darauf basierenden Higher-Layer-Protokolle vorgestellt.

Um das Protokoll CANaerospace am Ende dieser Arbeit bewerten zu können, beleuchtet dieses Kapitel einige Eigenschaften aus der Busarbitrierung und den Umgang mit Übertragungsfehlern in CAN-Bussystemen. Danach wird CANaerospace vorgestellt. Zur besseren Einordnung werden Parallelen und Unterschiede zum Protokoll CANopen aufgezeigt.

Der Kern der BCU-Firmware besteht aus einer speziell für eingebettete Systeme zugeschnittenen Laufzeitumgebung für die Skriptsprache Python, welche am Ende des Kapitels einführend beschrieben ist.

3.1. Controller Area Network (CAN)

Das Feldbussystem Controller Area Network (CAN) ist eine etablierte Lösung zur zuverlässigen Vernetzung verschiedenartiger technischer Einzelsysteme (CAN-Knoten).

Das Hauptanwendungsgebiet des CAN-Bus ist der Fahrzeugbau. Dort werden durch CAN die einzelnen Fahrzeugkomponenten, wie das Motorsteuergerät, das Infotainment-System, die Verstellmechaniken von Außenspiegeln oder die Pedalerie miteinander verbunden. Bei der Vernetzung technischer Industrieanlagen wird das Bussystem jedoch auch verstärkt eingesetzt. Durch seine hohe Zuverlässigkeit kommt es auch im Flugzeugbau zum Einsatz.

Der robuste und zugleich simple Aufbau als Zweidraht-Bussystem erlangt durch die Verwendung von verdrillten und geschirmten Leitungspaaren und der differenziellen Signalübertragung ein hohes Maß an Zuverlässigkeit zu einem kostengünstigen Preis.

Der Schaltungsaufwand bei den einzelnen Knoten ist im Vergleich zu Technologien wie Ethernet geringer. Es werden nur ein CAN-Controller für die OSI[1]-Sicherungsschicht und CAN-Transceiver zur Konvertierung zwischen Massenpotenzial bezogenen Signalen und differenziellen Signalen (OSI-Bitübertragungsschicht) benötigt. Oft ist die nötige Hardware für CAN-Controller in Mikrocontrollersystemen integriert.

Die Spezifikation wurde von der Robert Bosch GmbH erarbeitet. Die aktuelle Version 2.0 stammt aus dem Jahr 1991, siehe [2]. Weil darin nur Eigenschaften aus den beiden untersten OSI-Schichten definiert sind, entstanden im Laufe der Zeit einige Higher-Layer-Protokolle (HLP). Diese spezifizieren weitere Protokolleigenschaften der darüber liegenden OSI-Schichten und spezialisieren das Feldbussystem weiter für den jeweiligen Anwendungsfall.

Für das HLP CANopen entstanden zusätzlich noch zahlreiche Geräteprofile, die das allgmeine CANopen-Protokoll weiter auf einzelne Anwendungsgruppen, wie beispielsweise Motorsteuerungen (DSP-402[2]) oder Eingabe-/Ausgabe-Karten spezialisieren.

Allgemein kann man alle CAN-Bussysteme als nachrichtenorientierte und demokratische Netzwerke mit gleichberechtigten Teilnehmern (Knoten) ohne die Notwendigkeit eines Busmasters zusammenfassen.

Die Grundlage hierfür ist die Regelung des Zugriffs auf das Shared-Medium, dabei handelt es sich um die beiden Busleitungen CAN-High und CAN-Low. Der Zugriff wird über bitweise Arbitrierung organisiert und ist im Abschnitt 3.3.1 näher erläutert.

3.2. Higher-Layer-Protokolle

Dieser Abschnitt erläutert die Aufgaben und Eigenschaften von Higher-Layer-Protokollen (HLPs). Sie stellen bei CAN-Bussystemen die OSI[3]-Schichten oberhalb von Schicht 1 und 2 dar, vgl. Abbildung 3.1.

HLPs definieren, wie die Nutzdaten in den Nachrichten angeordnet sind und organisieren die Kommunikation. Das heißt, sie legen die Byte-Order fest und beschreiben eventuelle Hand-Shake-Verfahren zwischen den Knoten. Oft werden in einem Protokoll verschiedene Nachrichtenarten für Dienste zur Knoten-Konfiguration oder zyklische und azyklische Kommunikation definiert.

[1] Open Systems Interconnection Referenzmodell, siehe [1]
[2] https://www.can-cia.org/can-knowledge/canopen/cia402/
[3] Open Systems Interconnection Referenzmodell, siehe [1]

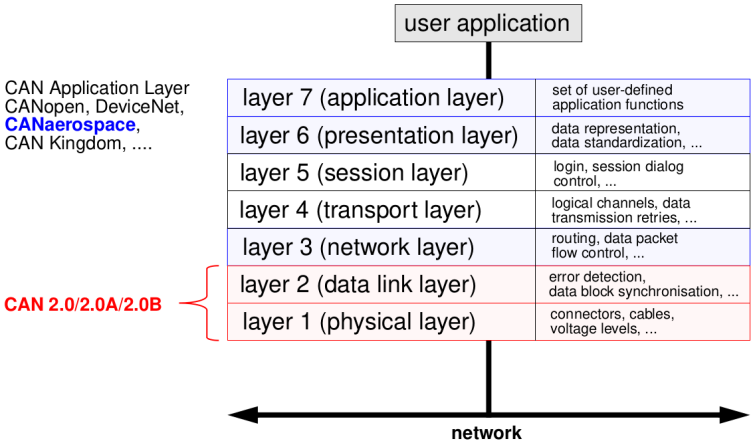


Abbildung 3.1.: OSI-Referenzmodell bezogen auf CAN [19] – Die unteren beiden Schichten werden durch die CAN-Spezifikation definiert, alle darüberliegenden Schichten hängen von der HLP-Spezifikation ab.

Die Zuordnung zwischen Information und dem CAN-Identifier einer Nachricht findet ebenfalls in HLPs statt. Anhand des Identifiers kann jeder Knoten, der das HLP implementiert hat, erkennen, wie er die Nachricht zu interpretieren hat. Außerdem ist mit ihm die Nachrichtenpriorität bei der Arbitrierung festgelegt.

Diese wird in Abschnitt 3.3.1 erläutert.

Beispiele für Higher-Layer-Protokolle sind CANaerospace[4] und ARINC825[5] aus dem Simulations- und Avionikbereich, CANopen[6] aus dem Bereich der Automatisierungstechnik und ISOBUS, vgl. [13], aus der Landmaschinentechnik.

3.3. Arbitrierung und Fehlerbehandlung beim CAN-Bus

Die CAN Spezifikation 2.0 definiert einige Fehlertypen innerhalb der Datensicherungsschicht des OSI-Modells, vgl. [2], Kap. 6.1 Error Detection, Seite 23. Weiter beschreibt die Spezifikation die Erkennung, Signalisierung und Behandlung von diesen Fehlertypen.

In diesem Abschnitt wird näher auf die bitweise Arbitrierung und auf die Fehlertypen eingegangen. So können hinterher die daraus entstehenden Konsequenzen für Higher-

[4] www.canaerospace.com, siehe auch [20]
[5] www.arinc-825.com
[6] www.can-cia.org/canopen, siehe auch [12]

Layer-Protokolle beleuchtet werden, um anschließend CANas vor diesem Hintergrund zu betrachten und mit anderen Protokollen in Relation setzen zu können.

3.3.1. Bitweise Arbitrierung und Übertragungsfehler

Jedes Bussystem muss den Schreibzugriff der Busteilnehmer auf das gemeinsame Übertragungsmedium organisieren. Beim Feldbussystem CAN erfolgt diese Organisation über bitweise Arbitrierung, welche in diesem Abschnitt genauer erläutert wird. Außerdem wird aufgezeigt, welche Konsequenz die Arbitrierung für den Umgang mit CAN-Identifiern hat.

Die in diesem Abschnitt erklärten Zusammenhänge bilden die Grundlage für eine spätere Bewertung des CANaerospace-Protokolls.

CAN-Bus-Teilnehmer, nachfolgend als Knoten bezeichnet, sind auf dem Bus als Wired-And verschaltet, um gleichzeitige Zugriffe auf das Shared Medium (Busleitungen: CAN-High u. CAN-Low) nach dem CSMA/CR-Verfahren auflösen zu können.

Jeder Knoten ist in der Lage die Busleitungen zu überwachen, während er auf den Bus schreibend zugreift. So können Unterschiede zwischen dem gewünschten logischen Pegel des sendenden Knotens und dem tatsächlichen Pegel auf dem Bus detektiert werden. Eine Bedingung dafür ist die Dominanz eines logischen Pegels gegenüber dem anderen Pegel. Beim CAN-Bus ist die logische Null dominant gegenüber der logischen Eins.

Über die dominanten und rezessiven Pegel wird die Arbitrierung organisiert, sodass sich immer die Nachricht mit der höchsten Priorität auf dem Bus unverändert durchsetzen kann und Kollisionen sofort aufgelöst werden. Daher wird das Zugriffsverfahren vom CAN-Feldbussystem manchmal auch CSMA/CD+AMP (Arbitration on Message Priority) genannt, vgl. [4], Abschnitt 3. Diese Eigenschaft ist eine Voraussetzung für die Echtzeitfähigkeit der Kommunikation und somit für die Wahl des CAN-Bussystems als Lösung zur Komponentenintegration für die SAFRAN Flugsystemarchitektur zwingend erforderlich.

Würden Kollisionen sich nicht auflösen lassen, sondern Buslaufzeit beanspruchen, so könnte es bei ausreichender Kollisionshäufigkeit zu einer Verzögerung des Informationsflusses kommen und die Sicherheit der Flugsystemarchitektur wäre nicht mehr gewährleistet.

Treten in einem CAN-Bus im Verhältnis zur Anzahl erfolgreicher Übertragungen zu viele Kollisionen auf, so kann dies zur Einstellung der Buskommunikation bei den Knoten führen. Dieser Effekt ist im folgenden Abschnitt 3.3.2 näher erläutert.

Das Arbitrierungsfeld einer CAN-Nachricht umfasst den Identifier und das Remote Transmission Request (RTR) Bit. Gestattet sind parallele Schreibzugriffe nur im Arbitrierungsfeld, im End of Frame und im Acknowledge Field. Die beiden zuletzt genannten Möglichkeiten sind für die Quittierung und Fehlermarkierung der Nachricht notwendig. Parallele Buszugriffe außerhalb der drei genannten Möglichkeiten können zu nichtvaliden Nachrichten führen.

Beginnen also beispielsweise zwei oder mehr Knoten zur gleichen Zeit den Versand einer Nachricht mit gleichem Identifier und unterschiedlichen Nutzdaten-Bytes im Datenfeld, so wird die Kollision nicht aufgelöst, die Nachrichten werden verworfen und es geht Buszeit verloren.

Die Wahrscheinlichkeit, für das Auftreten solcher unlösbaren Kollisionen ist so klein wie möglich zu halten. Ein Higher Layer Protocol (HLP) muss die Verwendung von CAN-Identifiern entsprechend gestalten. Im Idealfall wird jeder Identifier von maximal einem Knoten schreibend genutzt. Dann ist jede mögliche Kollision auf dem Bus auflösbar.

Die unten stehende Legende schlüsselt die Abkürzungen von Tabelle 3.1 auf, welche den schematischen Aufbau einer CAN-Nachricht verdeutlicht. Die Zusammensetzung einer Nachricht (Frame), variiert mit ihrem Typ. Es gibt Data Frames und Remote Frames. Bei Übertragungsfehlern durch Kollisionen oder anderer Art verändert sich ein Frame noch beim Versenden zu einem Error Frame. Dies geschieht durch das Setzen von Error Flags, welche die regulären Bits im End of Frame Feld ersetzen. Außerdem sind noch Overload Frames zur gewollten Verzögerung nachfolgender zusammenhängender Nachrichten definiert.

Diese Nachrichtentypen existieren seit der Einführung von 29-Bit Identifiern in jeweils zwei Formaten. Die ältere Variante ist das Standard-Identifier-Format mit 11-Bit-Identifern. Die erweiterte Variante ist das Extended-Identifier Format mit 29-Bit-Identifiern, dessen Verwendung innerhalb der Nachricht mit dem bisher reservierten r1-Bit gekennzeichnet wird. Dieses Bit heißt jetzt Identifier Extension Bit (IDE).

Die Tabelle 3.1 stellt einen Data Frame mit Standard-Identifier dar. In der untersten Zeile ist der Umfang des jeweiligen Feldes in Bit angegeben. Bei Ausnutzung aller acht Datenbytes im Data Field ergibt sich eine Länge von 108 Bit. Error Frames können in ihrer Länge davon leicht abweichen, je nach Error Flag Signalisierung im End of Frame. Außerdem variieren alle CAN-Nachrichten abhängig von ihrem binären Inhalt aufgrund der Verwendung von Stopfbits, welche die Synchronisation der Bitzeiten im Bussystem gewährleisten.

Für eine spätere Buslastbetrachtung werden wegen den Stopfbits und einem nötigen Freiraum zwischen den Frames (Inter Frame Space) 125 Bits als durchschnittlicher Wert für eine Nachricht mit 8 Nutzdatenbytes angenommen, vgl. [20], Abschnitt 6.3. Bei Extended-Identifiern können 144 Bits angenommen werden.

- **SoF** - Start of Frame
- **Arb** - Arbitration Field, bestehend aus:
  - ID - Identifier
  - RTR - Remote Transmission Request
- **Ctrl** - Control Field, bestehend aus:
  - IDE/r1 - Identifier Extension / Reserved Bit 1
  - r0 - Reserved Bit 0
- **Data** - Data Field, bestehend aus 8 Bytes für Nutzdaten
- **CRC** - Cyclic Redundancy Check Field, bestehend aus:
  - Seq - CRC Sequenz der Länge 15 Bits
  - Del - CRC Delimiter
- **Ack** - Acknowledge Field, bestehend aus:
  - Slot - ACK Slot
  - Del - ACK Delimiter
- **EoF** - End of Frame

SoF	Arb		Ctrl			Data	CRC		Ack		EoF
	ID	RTR	IDE/r1	r0	DLC		Seq	Del	Slot	Del	
1	11	1	1	1	4	64	15	1	1	1	7

Tabelle 3.1.: Der Aufbau eines Standard 11 Bit ID Datenframes mit einer Länge 108 Bits, exklusive gelegentlich auftretender Stopfbits

3.3.2. Fehlertypen und Verhalten im Fehlerfall

Zusammen mit den im vorherigen Abschnitt 3.3.1 erläuterten Eigenschaften der Arbitrierung bildet das von der CAN-Spezifikation vorgeschriebene Verhalten im Fehlerfall die Bewertungsgrundlage für das verwendete CANas-Protokoll. Daher beleuchtet dieser Abschnitt die Konsequenzen von Übertragungsfehlern für die Knoten.

Das Verhalten eines Knotens bei invaliden Nachrichten wird durch zwei Fehlerzähler und den im Zustandsdiagramm 3.2 gezeigten Zustandsautomaten vorgegeben. Die Zählerstände von Receive Error Count (REC) und Transmit Error Count (TEC) bestimmen den aktuellen Zustand. Je nach Fehlerfall wird der entsprechende Zähler um einen bestimmten Wert inkrementiert. Bei erfolgreich übertragenen Nachrichten werden die Zähler entsprechend dekrementiert.

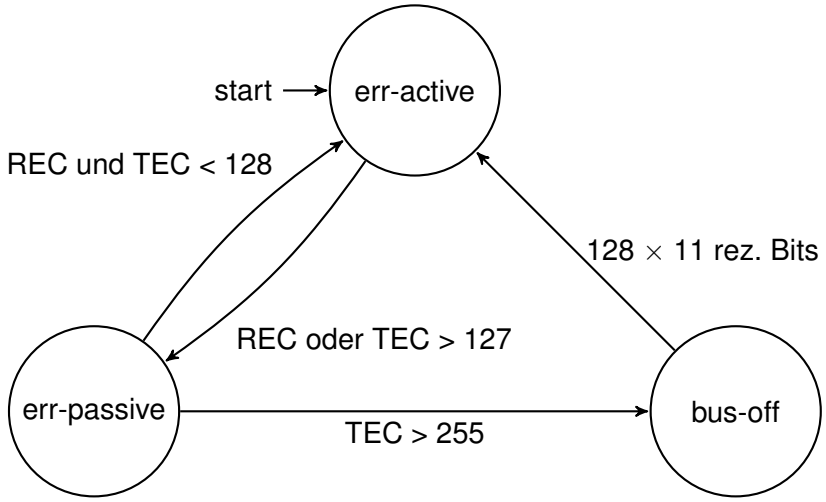


Abbildung 3.2.: Zustandsdiagramm der CAN-Fehlerbehandlung – Gezeigt wird die Abhängigkeit der Fehlerzustände von den Eingangs- und Ausgangsfehlerzählern REC und TEC

Der Zustandsautomat befindet sich im Normalfall im Zustand „Error-active“. In diesem Zustand gilt er als vollwertiger Busteilnehmer und hat das Recht, selbst Nachrichten zu versenden und die Nachrichten anderer Knoten als ungültig zu markieren, wenn er einen Fehler festgestellt hat.

Erreicht einer der beiden Fehlerzähler (REC oder TEC) einen Wert größer als 127, wechselt der Automat in den Zustand „Error-passive“. Damit ist es ihm nicht mehr erlaubt, ein „Error Active Flag“ in das End of Field (vgl. Tabelle 3.1) von Nachrichten anderer Knoten zu setzen. Er kann übertragene Nachrichten damit nicht mehr für alle

Teilnehmer als ungültig markieren. So ist sichergestellt, dass ein defekter Knoten die gesamte Buskommunikation nicht beliebig lang als Empfänger blockieren kann.

Zusätzlich muss ein Knoten im „Error-passive“ Zustand einen Zeitabschnitt nach dem Versand einer Nachricht verstreichen lassen, bevor er erneut senden darf. So ist ausgeschlossen, dass ein defekter Knoten eine hochpriorisierte Nachricht hintereinander zu versenden versucht.

Bei jedem fehlerhaften Übertragungsversuch wird der TEC um 8 inkrementiert, sodass ein Knoten nach dem 16. Sendeversuch mit den vorgeschriebenen Sendeverzögerungen beginnen muss, wenn er vorher den Wert 0 im TEC gespeichert hatte.

Überschreitet der TEC einen Wert von 255, so wird der Zustandsautomat in den „Bus-off“ Zustand versetzt und ist vorerst vollständig von der Kommunikation ausgeschlossen.

Der Zustandsautomat dient also dem Schutz des restlichen Bussystems vor dem jeweiligen Knoten. Er ist damit ein entscheidendes Regulierungselement für die Garantie einer sicheren Kommunikation.

Unterschieden werden fünf Fehlertypen, vgl. [2], Kap. 6.1 Error Detection, Seite 23. Jeder führt zu einer Inkremenierung von REC oder TEC:

Bit Error

Er liegt vor, wenn der Sender einen rezessiven Pegel auf den Bus zu legen versucht, er jedoch einen dominanten Pegel liest. Tritt diese Diskrepanz im Arbitrierungsfeld oder im ACK-Slot auf, so liegt kein Fehler vor.

Stuff Error

Ab dem Start of Frame bis zur CRC Sequenz exklusive CRC Delimiter werden nach dem 5. Bit gleichen Pegels in Folge Stopfbits des jeweils anderen Pegels in den Frame eingefügt. Wird auf dem Bus innerhalb dieser Felder eine Folge von 6 Bits gleichen Pegels detektiert, so liegt ein Stuff Error vor.

CRC Error

Stimmt die versendete Cyclic Redundancy Check (CRC) Sequenz nicht mit der vom Empfänger berechneten Sequenz überein, so liegt ein CRC Error vor.

Form Error

Dieser Fehler tritt auf, wenn in Fixed-Form Feldern fehlerhafte Bits erkannt werden, das sind die Felder, bei denen keine Stopfbits eingefügt werden.

ACK Error

Der Sender einer Nachricht erwartet im ACK Slot ein dominantes Bit als Quittierung von den Empfängern. Detektiert er dort ein rezessives Bit, so liegt ein ACK Error vor.

Die In-/Dekrementierungsregeln der beiden Fehlerzähler sind im Detail der CAN-Spezifikation zu entnehmen, siehe [2]. Vereinfacht lässt sich das Regelwerk wie folgt zusammenfassen:

- Receive Error Count
  - Er wird nur bei empfangenden Knoten verändert.
  - Wenn der Empfänger einen Fehler feststellt, außer es handelt sich um einen Bit Error in einem Active Error Flag oder Overload Flag, wird um 1 inkrementiert.
  - Wenn der Empfänger ein dominantes Bit direkt nach seinem gesendeten Error Flag detektiert, wird um 8 inkrementiert.
  - Wenn der Empfänger eine Nachricht bis zum ACK Slot als valide erkennt und erfolgreich sein ACK Bit versendet, wird um eins dekrementiert.
- Transmit Error Count
  - Wenn der Sender ein Error Flag sendet, wird um 8 inkrementiert.
  - Wenn beim Versenden eines Active Error Flags oder Overload Flags ein Bit Error auftritt, wird um 8 inkrementiert.
  - Wenn eine Nachricht erfolgreich versendet wurde, wird um eins dekrementiert. Liegt der TEC jedoch über 127, so wird der Zähler auf einen Wert zwischen 119 und 127 gesetzt.

3.4. CANaerospace

Dieser Abschnitt stellt das verwendete Higher-Layer-Protokoll CANaerospace (CANas) vor. Zur besseren Einordnung gegenüber anderen Protokollen wird das Prinzip der selbst identifizierenden Nachrichten und das Time Triggered Bus-Scheduling vorgestellt.

CANaerospace (CANas) ist eine lizenzfreie Schnittstellen-Spezifikation für CAN basierte Machine to Machine (M2M) Kommunikation in Luftfahrzeugen. Sie definiert nicht nur ein Higher-Layer-Protokoll, sondern legt auch Standards für die Steckverbindungen des CAN-Bussystems fest, siehe [20], S. 55 ff, Abschnitt 8 Physical connector definition.

Entwickelt wird die Spezifikation von Michael Stock, der das Protokoll auch in eigenen Avionik-Produkten seines Unternehmens Stock Flight Systems[7] nutzt. Darüber hinaus

[7] http://www.stockflightsystems.com

findet es in Projekten mit Forschungsflugzeugen wie SOFIA[8] (siehe Abb. 3.3), sowie auch im Simulatorbau beim Systemtechnikanbieter Reiser (siehe Abb. 3.4) Anwendung und wird in Electronic Control Units (ECUs) von Flugzeugmotoren des Herstellers ROTAX eingesetzt.

Abbildung 3.3.: SOFIA Forschungsflugzeug m. eingebautem Teleskop unter Verwendung v. CANas, [11]

Abbildung 3.4.: Reiser Simulatorkomponenten, die Ansteuerung im Simulationssystem erfolgt via CANas, [19]

Eine erste Version von CANas wurde am 10.10.1998 veröffentlicht, vgl. [20], S. 1. Für die BCU wird die aktuelle Version 1.7 vom 12.01.2006 genutzt. Eine Version 2.0 stehe laut M. Stock kurz vor der Veröffentlichung ([18]).

Die Spezifikation ist für bemannte Luftfahrzeuge ausgelegt. Dementsprechend sind auch die Steckverbindungen auf hohe Zuverlässigkeit hin ausgewählt worden und für UAVs zu groß und zu schwer. Daher weicht die CANas-Umsetzung der BCU innerhalb des SAFRAN-Projekts vom Standard ab und verwendet statt Twisted-Pair Leitungen ein Flachbandkabel mit Wannensteckern, wie sie auf Abbildung 5.3 zu sehen sind. Das Flachbandkabel ist für Störeinflüsse von außen anfälliger als verdrillte Leitungspaare. Für Testaufbauten bei der Entwicklung ist es aufgrund seiner einfachen Verwendbarkeit durch schnell montierbare Stecker mit Stechleitungen jedoch besser geeignet.

Die Kommunikation eines CANas-Bussystems lässt sich vereinfacht in 16 Dienste (Node Services) zur Knotenkonfiguration, ereignisbasierten Notfallnachrichten (Emergency Event Data EED) und Nachrichten zur Prozessdatenübermittlung (Normal Operation Data NOD) unterteilen.

Die Dienste kommen hauptsächlich nur bei der Inbetriebnahme des Bussystems zum Einsatz. Sie wenden ein Handshake-Verfahren an, um von einem Knoten aus genau

[8] Stratospheric Observatory for Infrared Astronomy (SOFIA), [21]

einen anderen Knoten zu adressieren und nutzen somit eine verbindungsorientierte Form der Kommunikation.

Über Dienste lassen sich Menge und Art der angeschlossenen Knoten auslesen oder auch die Bitrate des Bussystems setzen. Auch die Node-ID, also die Adresse der einzelnen Knoten lässt sich neu zuweisen, so wie auch die Identifier-Zuordnung zu den einzelnen Informationen innerhalb eines Knotens. Auch ein Dienst zur Übertragung größerer zusammenhängender Datenblöcke, beispielsweise für Firmware-Updates, ist von der Spezifikation definiert. Durch diese Menge von Diensten ermöglicht das CANas-Protokoll ein hohes Maß an Komfort und Automatisierbarkeit bei der Inbetriebnahme und Rekonfiguration von Bussystemen und dem Austausch von einzelnen Knoten.

Zusätzlich bietet CANas die Freiheit, eigene Dienste zu definieren und ist somit sehr flexibel an verschiedenartigste Knoten anpassbar. Dieser Vorteil wurde bei der BCU für die Übertragung von MicroPython Bytecode-Skripten genutzt.

Normal Operation Data (NOD) kann azyklisch oder zyklisch versendet werden. Hierbei handelt es sich immer um einzelne, nicht zusammenhängende Nachrichten und somit um eine verbindungslose Kommunikation. Alle regulären Flugdaten, wie Position, Geschwindigkeiten, Rollraten, Motordrehzahlen oder Piloteneingaben werden während des Fluges echtzeitfähig via NOD über das Bussystem zwischen den Knoten ausgetauscht. Um die Echtzeitfähigkeit garantieren zu können, muss bei der Busauslegung einmalig eine numerische Buslastbetrachtung vorgenommen werden, um sicher zu stellen, dass trotz azyklisch auftretender Nachrichten genügend Buslaufzeit innerhalb der erwarteten Zeitintervalle für die Übertragung aller Nachrichten zur Verfügung steht. Dadurch ergibt sich die geforderte Echtzeitfähigkeit für alle Nachrichten, auch für jene mit niedriger Priorität.

Ein Vorteil von CANas ist die Selbstidentifizierung der Nachrichten. Das Nachrichtenformat von CANas teilt die acht Byte Nutzdaten des CAN-Frames in vier Byte Headerdaten und vier Byte Nutzdaten ein. Im Header werden Node-ID, Datentypkennung und zwei weitere Dienst-spezifische Codes versendet, siehe Tabelle 3.2. Durch dieses Format können Knoten verschiedener Hersteller die eingehenden Nachrichten der jeweils anderen Knoten interpretieren, ohne vorher bei der Systemintegration aufeinander abgestimmt worden zu sein.

Auch wenn einem Knoten nichts über sein Umfeld bekannt ist, kann er eingehende Nachrichten über den Service Code einem Dienst zuordnen, sofern er diesen unterstützt oder beim Typ NOD den Datentyp erkennen und weiß somit, wie er die Nutzdaten interpretieren muss. Wenn alle Knoten des Bussystems die Standard Identifier Dis-

Byte 0	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7
Header				Data			
Node ID	Data Type	Service Code	Message Code				

Tabelle 3.2.: CANaerospace Nachrichtenstruktur der acht Nutzdaten-Bytes – Der Header verleiht jeder Nachricht die Eigenschaft der Selbstidentifizierung.

tribution zwischen Information und CAN-Identifier von CANas unterstützen oder alle eine identische eigene Zuordnung nutzen, können sie auch die Nutzdaten semantisch interpretieren. Das heißt ohne Kenntnis der Zuordnung, kann ein Knoten eine NOD Nachricht empfangen und beispielsweise einen vorzeichenbehafteten Fließkommawert daraus auslesen. Mit Kenntnis über die Zuordnung kann er dann zusätzlich feststellen, dass es sich bei dem Wert beispielsweise um die Ist-Rollrate des Flugsystems handelt.

Für den Anwender von CANas reduziert sich die notwendige Konfiguration bei der Integration eines Knotens in einen Bus daher auf ein Minimum. Der Knoten benötigt eine Node-ID, die Bitrate des Busses, die korrekten CAN-Identifier für die Nachrichten, die für ihn relevant sind, und wenn er NOD versendet, benötigt er ggf. noch die Versandtfrequenz für die zyklische Wiederholung der Nachrichten.

Sobald er diese Informationen erhalten hat, welche sich auch alle via diverser Dienste setzen lassen, sofern diese unterstützt sind, kann er an der Kommunikation teilnehmen.

Damit ein Knoten als CANas kompatibel bezeichnet werden kann, muss er mindestens den Dienst IDS (Identification Service) unterstützen, alle anderen Dienste sind optional. Es empfiehlt sich aber, einige weitere Dienste in den Knoten zu implementieren, da sie den Komfort bei der Inbetriebnahme des Bussystems erhöhen, wenn sich alle benötigten Parameter von einem beliebigen Knoten aus via CAN-Bus bei allen Busteilnehmern setzen lassen.

Außerdem kann ein Knoten vollständig von der in der Spezifikation vorgestellten Standardzuordnung zwischen Information und Identifier abweichen und muss dies nur in seiner Antwort auf eine IDS-Anfrage kenntlich machen. Dieser Sachverhalt ist beispielhaft für die Freiheitsgrade und die damit verbundenen Anforderungen an den Anwender, die CANaerospace festlegt. Hierauf wird genauer in Abschnitt 5.5 eingegangen.

Die Standardzuordnung weist den gebräuchlichsten Luftfahrzeugdaten CAN-Identifier zu. Dazu wird auch eine beispielhafte Buslastbetrachtung für den zyklischen Versand dieser Daten vorgestellt. Hieran kann sich der Anwender bei der Auslegung seines CANaerospace Bussystems und der darin zu übertragenden Datenmenge orientieren.

Eine Buslastbetrachtung ist im Zusammenhang mit dem “Time-triggered bus scheduling“ notwendig, da es anders als beim TTCAN[9]-Verfahren keine busweit synchronisierten Uhren in den Knoten geben muss, vgl. [20], Abschnitt 6.

Jeder CANas-Knoten überwacht nur die Buslast, die er selbst mit zyklischer Kommunikation - Normal Operation Data (NOD) erzeugt. Es ist die Aufgabe des Anwenders, alle Knoten so zu konfigurieren, dass die Buslaufzeit der gesamten NOD des Bussystems nicht mehr als 80 % der zur Verfügung stehenden Buslaufzeit einnimmt. So ist sichergestellt, dass auch niederpriorisierte Nachrichten Zeit zur Übertragung erhalten.

Auch ein Redundanzkonzept bietet die Spezifikation an. Dazu müssen alle Knoten den CAN2.0B Standard unterstützen, weil redundante Informationen mit einem Abstand von 10.000_{16} im CAN-Identifier versendet werden und daher 29 Bit Identifier verwendet werden müssen, siehe [12], S. 32 ff. Dieses Konzept bringt einen Nachteil bezüglich der Priorisierung mit sich, welcher im nachfolgenden Abschnitt 3.4.1 genauer erläutert und bewertet wird.

Allgemein lässt sich über CANas sagen, dass sich ein grundlegendes Verständnis der Spezifikation zügig aufbauen lässt und auch die Implementierungsaufwände bei der Entwicklung eines CANas-Knotens gering sind.

3.4.1. Vergleich von CANaerospace und CANopen

Aus den in diesem Kapitel aufgezeigten Eigenschaften der CAN-Arbitrierung lassen sich Konsequenzen für das Design von Higher-Layer-Protokollen (HLP) ziehen. Vor diesem Hintergrund wird in diesem Abschnitt CANaerospace und CANopen vergleichend betrachtet. Stärken und Schwächen der Protokolle werden hierbei deutlich und zur besseren Einordnung gegenüber gestellt.

Wegen der in Abschnitt 3.3.1 erläuterten Einschränkung der bitweisen Arbitrierung, empfiehlt es sich, ein HLP so zu entwerfen, dass möglichst jeder Identifier nur von jeweils einem Knoten im Bus sendend genutzt wird. Vor allem bei Anfragen via Broadcast, bei denen eine Antwort von mehreren Knoten erwartet wird, ist das für einen reibungslosen Betrieb des Bussystems wichtig.

Das HLP CANopen setzt dies konsequent um. Logische Kommunikationskanäle des Protokolls werden dort als Communication Objects (COBs) zusammengefasst und mit je einem CAN-Identifier versehen. Diese Objekte sind im Identifier-Adressbereich so

[9] Time Triggered CAN, siehe auch [5]

verteilt, dass ein sendender Knoten auf den entsprechenden Identifier seine eigene Node-ID aufaddieren kann.

Bei CANopen sind bis zu 127 Knoten in einem Bussystem zulässig. Also sind die einzelnen COB-IDs mit einem Abstand von 128 verteilt. Beispielsweise nutzt das erste Sende-Prozessdatenobjekt (tPDO1) eines Knotens die CAN-ID 180_{16}, wodurch sich ein Adressbreich von 181_{16} bis $1FF_{16}$ für das tPDO1 ergibt. Das erste Empfang-Prozessdatenobjekt (rPDO1) nutzt die CAN-IDs 200_{16} bis $27F_{16}$. Die weiteren PDOs folgen auf den Identifier-Grenzen 280_{16} u. 300_{16}; 380_{16} u. 400_{16}; usw., vgl. [12] Tabelle 67 u. 71, S. 130 u. 137.

Die Anzahl der PDOs hängt vom Geräteprofil ab, welches die CANopen-Spezifikation für den jeweiligen Anwendungsfall weiter spezialisert. Beispielsweise sieht das Profil DSP 402[10] für Motorsteuerungen vier tPDOs und vier rPDOs vor, womit bereits 49 % des Standard-Identifier-Bereichs belegt sind.

Zu den wenigen Ausnahmen von COBs, die keine aufaddierte Node-ID erhalten, zählen Sync- und Timestamp-Nachrichten. Diese werden per Definition nur von einem Knoten versandt, sodass diese Nachrichten nicht weiter individualisiert werden müssen.

Das HLP CANaerospace (CANas) ist gegen eine überschneidende Verwendung von Identifiern durch sendende Knoten schlechter abgesichert als CANopen. Es setzt hierzu deutlich stärker auf die Kompetenz und Weitsicht des Anwenders. Er muss bei der Vergabe von Identifiern zur Normal Operation Data Kommunikation Überschneidungen zwischen den Knoten vermeiden. Ebenfalls ist die parallele Verwendung von Nodes Services ein Risiko, weil Handshake-Konflikte vom Protokoll nicht vollständig ausgeschlossen werden, sondern nur durch das Angebot zahlreicher Service Channel hinreichend vermeidbar sind. Dieser Zusammenhang wird im Abschnitt 5.5 detaillierter untersucht.

Allgemein ist festzustellen, dass CANas die Kompatibilität zwischen Knoten verschiedenener Hersteller durch Freiheitsgrade und Konfigurationsmöglichkeiten generiert. Die Knoten lassen sich schnell und komfortabel soweit anpassen, dass sich aus ihnen ein zuverlässiges Bussystem aufbauen lässt. CANas ist somit auch im größeren Maße auf die jeweilige Anwendung adaptierbar.

CANopen ist restriktiver spezifiziert und generiert seine herstellerübergreifende Kompatibilität durch absichernde Strukturen und Regeln für die Kommunikation. Viel Spielraum für Adaptierungen auf den Anwendungsfall bleibt bei CANopen nicht, die Adaption

[10] https://www.can-cia.org/can-knowledge/canopen/cia402/

geschieht durch zahlreiche spezialisierte Geräteprofile, an die sich die Hersteller halten müssen.

Folgern lässt sich daraus, dass bei CANas mehr Sorgfallt und Verständnis bei der Busauslegung und Konfiguration für einen zuverlässigen Busbetrieb nötig ist, als bei CANopen. Dafür ist das CANas-Protokoll aber auch simpler aufgebaut und somit schneller in die Firmware von Knoten zu integrieren und benötigt je nach Ausmaß der Umsetzung weniger Programmspeicher im Read Only Memory (ROM) eines eingebetteten Systems.

Das Redundanzkonzept von CANas sieht, wie bereits in Abschnitt 3.4 erwähnt wurde, eine Nutzung vom CAN-Standard 2.0B vor, da ein größerer Identifierbereich benötigt wird und 29 Bit Identifier dazu erforderlich sind. Wenn mehrere Knoten als redundante Quellen für eine Information im Bus existieren, sind sie so zu konfigurieren, dass sie mit einem Identifierabstand von 10.000_{16} ihre Information verbreiten.

Die Body Roll Rate mit dem Identifier 304_{16} aus Tabelle B.1 würde beispielsweise bei drei verschiedenen Quellen im Bus unter den IDs 304_{16}, 10.304_{16} und 20.304_{16} verbreitet werden. Der Identifierabstand wird also pro Knoten vergeben, auch wenn dies nicht einheitlich für alle gelieferten Informationen eines Knotens geschehen muss. Der Abstand ist so groß, dass er sämtliche andere Identifierzuordnungen überspannt.

Bei CANopen ist die Priorität durch die Addition der Node-ID auf den CAN-Identifier vom Knoten und vom verwendeten logischen Kommunikationskanal (z. B. PDO oder Service Data Object SDO) abhängig.

Keines der beiden HLPs priorisiert demnach die Nachrichten gemäß ihrer Information, was für das Gesamtsystem eigentlich die Beste Lösung wäre. CANas bietet durch seine Freiheitsgrade und hier speziell durch die Verwendung einer eigenen redundanten Identifier Distribution, die Möglichkeit, dieses Problem zu umgehen. Bei der Umsetzung der BCU wurde dieser Freiheitsgrad genutzt und ist im Zuge der Vorstellung einer eigenen SUAV Identifier Distribution im Abschnitt 5.1 genauer erläutert.

Ein CANopen-Knoten kann nicht beliebig viele verschiedene Informationen zyklisch und verbindungslos, also ohne Client-Server-Handshake Mechanismus, versenden. Dazu stehen pro Knoten nur die PDOs zur Verfügung. Beim Geräteprofil DSP 402 sind das lediglich 4 an der Zahl. Diese PDOs sind am besten mit der NOD-Kommunikation von CANas zu vergleichen.

Ein CANas-Knoten kann sämtliche Informationen der gesamten Identifier-Distribution für Normal Operation Data (NOD) selbstständig versenden. Ein CANopen-Knoten muss

durch PDO-Mapping erst seinen tPDOs andere Informationen zuweisen, falls er mit der Anzahl an transmit Process Data Objects (tPDOs) nicht auskommt.

CANopen-Nachrichten sind weniger selbstidentifizierend als CANas-Nachrichten, da sie keinen eigenen Header im Datenfeld des CAN-Frames vorsehen. Bei den Process Data Objects (PDOs) von CANopen sind dadurch alle 8 Nutzdaten-Bytes für die Nutzdaten verfügbar, während CANas-Knoten mit 4 Nutzdaten-Bytes auskommen müssen. In späteren Versuchen wird sich jedoch zeigen, dass die Datendurchsatzrate keinen Engpass für die SUAV-Anwendung darstellt, siehe 5.1.

Beide HLPs haben unterschiedliche Vor- und Nachteile und stellen valide Optionen für die Wahl des Protokolls der BCU dar. CANas ist durch seine Freiheitsgrade jedoch das passendere HLP und bietet eine höhere Wahrscheinlichkeit zur Anbindung von COTS-Knoten, weil sich das Protokoll im Avionikbereich etabliert hat.

Außerdem ist Implementierungsaufwand durch die geringen Minimalanforderungen bei den Node Services kleiner.

3.5. MicroPython als Skriptsprache für eingebettete Systeme

MicroPython (µP)[11] wurde ursprünglich von Damien George implementiert und wird mittlerweile als Open-Source Projekt von vielen Mitwirkenden weiterentwickelt. Es handelt sich dabei um eine Python-Laufzeitumgebung für eingebettete Mikroprozessorsysteme, welche in C geschrieben ist.

Im Jahr 2013 wurde µP unter der MIT Lizenz veröffentlicht und neben dem ursprünglichen Pyboard (siehe [8]) erscheinen immer mehr Portierungen. Das Unternehmen Adafruit vertreibt eigene Mikrocontrollerplatinen auf denen ein MicroPython Derivat namens CircuitPython[12] läuft.

Der Sourcecode umfasst in der Version 1.8.1 ca. 776.000 Zeilen und bietet mehrere Portierungen inklusive eines eigenen Python-Bytecode-Compilers. Über C Funktionen sind für die µP-Umgebung alle Komponenten erstellt worden, die nötig sind, um Pythonskripte auf eingeschränkten eingebetteten Systemen ausführen zu können. Diese Systeme sind Mikrocontroller ab einer Minimalkonfiguration von 8 kB Random Memory Access (RAM) und 256 kB ROM, vgl. [7].

[11] https://micropython.org
[12] https://github.com/adafruit/circuitpython

Solche Systeme bieten bereits genug Ressourcen, um Pythonskripte auf zwei verschiedene Arten auszuführen. Einerseits ist es möglich, Skripte direkt einzugeben, wobei auf dem System ein Lexer, Parser und Interpreter vor der eigentlichen Ausführung arbeiten muss. Andererseits kann man auch vorinterpretierten Python Bytecode in das System eingeben, welcher direkt ausgeführt werden kann.

Die Bytecode-Variante spart Arbeitsspeicher und Laufzeit, weil die Komponenten zur Interpretierung nicht geladen werden müssen und keine Zeit für die Interpretation aufgewendet werden muss. Arbeitsspeicher und Laufzeit sind auf Mikroprozessorsystemen aufgrund der tendenziell minimalistisch gestalteten Hardware wertvolle Ressourcen. MicroPython nutzt diese Ressourcen auf sparsame Weise und ist dadurch besonders gut als technische Basis für einen Skript-fähigen und somit generischen Busknoten geeignet. In Abschnitt 4.1.3 der System-Design Beschreibung sind weitere Vorteile der Verwendung von Bytecode genannt und dessen Erzeugung für die BCU erklärt.

Einen Interpreter mit einer Laufzeitumgebung für die Skriptsprache Python zu implementieren, ist eine komplexe Aufgabe. Da MicroPython ein OpenSource-Projekt ist und sich daher die Menge der mitwirkenden Entwickler ständig verändern kann, müssen alle Vorgänge und Softwarestrukturen für eine effiziente Zusammenarbeit möglichst selbsterklärend, ausreichend kommentiert oder außerhalb des Source-Codes dokumentiert sein.

Die Anwendung von MicroPython mit Hilfe der fertigen Portierungen ist umfangreich beschrieben. Der Source-Code der jeweiligen Firmwares ist es jedoch nicht. Es zeichnete sich bei der Erstellung dieser Arbeit ab, dass hierin der größte Nachteil von µP besteht.

Die Wartbarkeit von Source-Code ist bei µP der Performanz gegenüber stark untergeordnet, um auf eingebetteten Mikroprozessorsystemen überhaupt eine nutzbare Python-Laufzeitumgebung bieten zu können. Jedoch wird eine Weiterentwicklung bestehender Portierungen und die Erstellung neuer Portierungen durch mangelnde Dokumentation und schlecht erweiterbarer Software-Architekturen unnötig erschwert. Beispielsweise ist die Implementierung von Hardwarezugriffen häufig zu fest mit darüber liegender, Hardware-unabhängiger Logik verknüpft. Die Nachvollziehbarkeit der Firmware wird zusätzlich durch den Einsatz zahlreicher C-Makros verschlechtert.

Auch die Wahl der prozeduralen Programmiersprache C erscheint gegenüber C++ wenig plausibel. Die Skriptsprache Python bietet objektorientierte Programmierung an. Dieses Konzept ist mit Hilfe von eigenen Instanziierungsfunktionen in prozeduraler

C-Programmierung realisiert worden und wäre durch das ebenfalls objektorientierte C++ sicherlich komfortabler und nachvollziehbarer umzusetzen gewesen.

Die Mängel im Bereich Wartbarkeit und Dokumentation führen auch zu Einschränkungen der Debugging-Fähigkeit von µP. So ist es im Verlauf dieser Arbeit nicht gelungen, die Ausführung einzelner Python-Befehle via Einzelschritt-Debugging nachzuvollziehen. Die Laufzeitumgebung wird dadurch zu einer geschlossenen Komponente, die vorerst nicht modifizierbar ist und in der eventuell auftretende Fehler vorerst nicht zu beheben sind, weil dazu die Verständnisgrundlage fehlt.

Besonders positiv hervorzuheben ist die einfache Kapselung von µP innerhalb der umgebenden Firmware. Sobald die feste Verknüpfung zwischen der Anwendungslogik (Lexer, Parser, Compiler, Laufzeitumgebung) und den Routinen für die Hardwarezugriffe durch den Einsatz eines eigenen Hardware Abstraction Layer (HAL) aufgelöst ist, ist die eigentliche Laufzeitumgebung von MicroPython besonders beim Gebrauch des Arbeitsspeichers einfach von seiner Umgebung zu trennen.

In Python instanziierte Objekte unterliegen der Verwaltung einer Garbage Collection. Um diese Funktionalität zu liefern, verfügt MicroPython über einen eigenen Heap, dessen Größe sich bei der Initialisierung zu Beginn der Laufzeit in C festlegen lässt.

Somit lässt sich die Laufzeitumgebung im Heap-Speicherbedarf klar vom Rest einer umgebenden Firmware trennen. Der Stack hingegen ist ungeschützt, sodass die Möglichkeit besteht, die BCU durch ein entsprechend ressourcenintensives Skript mittels Stack-Überlauf außer Funktion zu setzen. Der Stackverbrauch lässt sich in MicroPython zwar überwachen, aber Stacküberläufe werden nicht verhindert.

Es gibt jedoch die Möglichkeit, einen Ausführungsstop bzw. einen definierten Notlauf-Zustand einzuleiten, sobald der Stack eine gewisse Größe erreicht hat. Dies ist möglich, weil sämtliche Python-Funktionsaufrufe in der Firmware über nur sieben C-Funktionen organisiert sind und man bei jedem Aufruf dieser Funktionen den aktuellen Stackpointer auslesen könnte. Untersuchungen hinsichtlich solcher Grenzfälle wurden im Rahmen dieser Arbeit nicht unternommen. Entscheidend für den zuverlässigen Betrieb der BCU ist jedoch die Möglichkeit einer Firmware-Erweiterung, welche die Laufzeitumgebung ab einer bestimmten Stackgröße an ihrer Ausführung hindert. Ist dieses Limit hinreichend klein gewählt, verbleibt eine ausreichende Menge an Arbeitsspeicher für die umgebende Firmware, um die BCU in einen definierten Fehler- oder Resetzustand zu bringen.

Zumindest bei einem drohenden Stacküberlauf durch beispielsweise Rekursion innerhalb eines Pythonskripts lässt sich so die µP-Ausführung rechtzeitig einschränken.

Bei einem Vergleich von µP und anderen kompakten Python-Laufzeitumgebungen, wie PyMite[13], TinyPY[14] und dem Zerynth-Framework[15] fällt auf, dass µP viele Eigenschaften vereint, die andere Interpreter nicht oder nur teilweise aufweisen. MicroPython implementiert die aktuellere Python-Version 3, während andere Umsetzungen zum Teil auf Python 2 aufsetzen und seit mehr als einem Jahr nicht mehr weiterentwickelt werden. Es ist quelloffen und mit der MIT-Lizenz sehr anwenderfreundlich veröffentlicht, es liefert nicht nur die virtuelle Maschine zur Ausführung von Bytecode, sondern auch einen Crosscompiler und die Möglichkeit zur Interpretation von Python-Sourcecode auf dem Mikrocontrollersystem selbst und überzeugt durch Ressourcenschonung, vgl. [16], Abschnitt A.

Daher wurde MicroPython als Technologiebasis für den in dieser Arbeit zu entwickelnden Buskonverter ausgewählt.

[13] PyMite (Python on a chip): https://code.google.com/archive/p/python-on-a-chip/
[14] TinyPY: http://www.tinypy.org/
[15] Zerynth: https://www.zerynth.com/, siehe auch [15]

4. System-Design

Die BCU ist ein ARM Cortex-M4 basiertes Mikrocontroller-System mit einer MicroPython-Laufzeitumgebung. Dem Benutzer wird damit die Möglichkeit gegeben, das Verhalten seiner BCUs mit eigenen Pythonskripten für seine Applikation zu definieren. Die Skripte können zur Laufzeit von beliebiger Stelle aus über CAN in die einzelnen BCUs hineingeladen und ausgeführt werden.

Das dazu nötige Konzept und die daraus entstandene Firmware-Architektur wird in diesem Kapitel beschrieben.

Für die Interaktion zwischen Benutzer und BCU wurde eine Toolchain aufgebaut. Deren Werkzeugkomponenten und die einzelnen Schritte im Entwicklungsprozess von BCU-gestützten Applikationen werden ebenfalls erläutert.

4.1. Firmware-Architektur

Die Basis der Firmware ist in C++ implementiert. Darin ist eine MicroPython-Portierung der Version 1.8.1 eingebettet.

In Abbildung 4.1 ist die Firmware schematisch dargestellt. Die grünen Bestandteile sind alle in C implementiert und gehören zu MicroPython. Die blauen Firmwarekomponenten sind in C++ erstellt und bestehen aus dem HAL, einem Script-Manager zur Verwaltung von Python-Bytecodes und Übertragungsprotokoll Implementierungen.

Ein Ziel der Firmware ist es, dem Benutzer die Implementierung seiner Applikation so einfach wie möglich zu gestalten. Sämtliche Initialisierungsvorgänge der Peripherie, wie z. B. die der Schnittstellen, werden dem Benutzer abgenommen. Ein Nachteil hierbei ist die statische Zuweisung der Hardware-Schnittstellen (UART, I²C, SPI, digital Ein-/Ausgänge) auf die General Purpose Input Output (GPIO) Pins des Mikrocontrollers und die dafür vorgesehenen Steckverbinder. Lediglich die Zahl der digitalen Ein- und Ausgabekanäle könnte man unter Verzicht auf andere Funktionen flexibler und via Python konfigurierbar gestalten. Dies ist jedoch im Rahmen dieser Arbeit nicht vorgesehen.

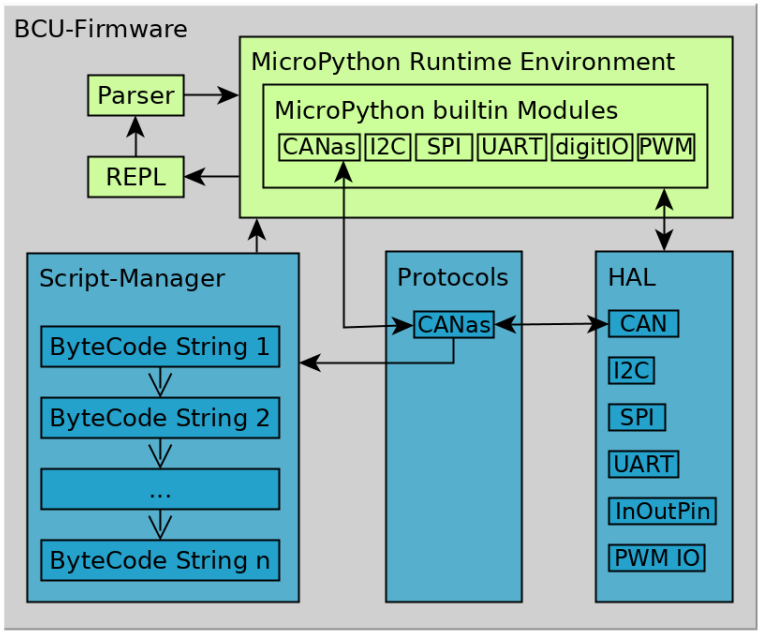


Abbildung 4.1.: Firmware-Architektur der BCU, grün: C-Anteil, hauptsächlich die µP Laufzeitumgebung; blau: C++-Anteil, HAL und Skriptverwaltung

Zu den Aufgaben der Firmware zählt die Konfiguration der Peripherie auf Register-Ebene. Der HAL bietet Objekt-Methoden für den Betrieb sämtlicher Schnittstellen-Peripherie. Durch lose Kopplung ist der HAL so implementiert, dass eine spätere Portierung der Anwendungslogik oberhalb des HAL auf andere Mikrocontroller (µC) möglichst einfach ist.

Die Abbildung 4.1 illustriert die Verknüpfung zwischen µP und der Schnittstellen-Peripherie des µC. Jede Hardware-Schnittstelle ist durch eine eigene Klasse im HAL vertreten (blau). Jede dieser C++-Klassen wird von einem Modul (grün) in MicroPython angesprochen. Die Module bilden die Methodensignaturen der C++-Methoden nach.

Ruft der Anwender beispielsweise eine Methode des I²C-Moduls in seinem Pythonskript auf, wird im Hintergrund der jeweilige C++-Code ausgeführt und dessen Rückgaben in die Python-Laufzeitumgebung geleitet. Die I²C-Schnittstelle verfügt genau wie SPI, UART und CAN über Pufferspeicher für eingehende Daten. In µP kann der Benutzer die Anzahl zwischengespeicherter Daten des Puffers abfragen oder die Daten aus dem Puffer auslesen. Hierzu rufen entsprechende C-Funktionen aus der MicroPython-Laufzeitumgebung ihre in C++ geschriebenen Hilfsmethoden auf. Deren Rückgaben (z. B.: die Anzahl zwischengespeicherter Daten im Puffer) werden von den

C-Funktionen entgegen genommen, in entsprechende Python-Datentypen konvertiert und zurück in die MicroPython-Laufzeitumgebung geleitet.

Der Vorteil dieser modularen Architektur besteht in ihrer übersichtlichen und klaren Trennung zwischen dem Hardware-spezifischen HAL und der darüber liegenden Anwendungslogik, welche Hardware-unabhängig ist. Diese Aufteilung vereinfacht eine erneute Portierung auf andere Hardware. Außerdem können so Anwendungslogik und HAL unabhängig voneinander getestet werden.

4.1.1. Integration von Übertragungsprotokollen

Um eine schnelle Applikationsentwicklung weiter zu unterstützen und die Performanz der BCU möglichst hochzuhalten, bietet es sich an, oft genutzte Grundfunktionen aus der Skriptsprache Python herauszuholen und sie in C++ oder auch Assembler zu implementieren. Das Übertragungsprotokoll CANaerospace ist so eine Grundfunktion. Daher wurde die CAN-Schnittstelle im Rahmen dieser Arbeit nicht einfach nur als ansteuerbare Hardware in Form eines Python-Moduls bereitgestellt, sondern zusätzlich wurde ein CANas-Modul implementiert, welches einige Komfortfunktionen für die Kommunikation mit dem Higher-Layer-Protokoll CANaerospace anbietet. Auf diese Weise werden Implementierungsaufwände beim Benutzer gespart und die Performanz der BCU bei der CANas-Kommunikation erhöht.

Das Konzept ist erweiterbar gestaltet, dass zu einem späteren Zeitpunkt mehr Protokollimplementierungen außerhalb des Rahmens dieser Arbeit folgen können. CANopen wäre hier denkbar, oder Treiber für oft verwendete Komponenten an anderen Hardware-Schnittstellen. Ein weiteres Beispiel für zukünftige Protokollerweiterungen ist eine OneWire Implementierung, die sich für einen der digitalen Ein-/Ausgänge aufschalten lässt.

4.1.2. Interaktiver Interpreter (REPL)

In erster Hinsicht müssen Python-Skripte auf der BCU ausführbar sein, jedoch ist es für einen einfachen Entwicklungsprozess von Vorteil eine Python-Shell zu haben, um Python-Anweisungen interaktiv auszuführen.

Durch die direkte Eingabe von Python-Code lässt sich schnell prüfen, ob man die korrekte Syntax verwendet und ob der eingegebene Ausdruck zu der erwarteten Ausgabe führt. MicroPython bietet dazu eine Routine, die endlos in einem Zyklus aus Einlesen, Auswerten und Ausgeben schleift, eine sogenannte Read-Eval-Print Loop (REPL).

Die REPL liest Klartext-Eingaben vom Benutzer ein und interpretiert diesen als Python-Code. Hierbei wird also zur Laufzeit ein Parser samt Lexer in den Arbeitsspeicher geladen, um den Python-Code in ausführbare Maschinenbefehle umzusetzen. Das kostet vor der eigentlichen Ausführung Laufzeit und schränkt den für die Applikation nutzbaren Adressbereich im RAM ein. Daher ist die REPL als Komfort-Feature bei der Applikationsentwicklung und als gelegentlich nutzbares Kontroll- und Überwachungswerkzeug anzusehen. In der fertigen Applikation empfiehlt es sich, Python-Bytecode zu nutzen, anstatt benötigte Skripte zur Laufzeit als Source-Code in Form von ASCII-Zeichen via REPL in die BCU zu übertragen.

Die Ein- und Ausgabe der REPL ist einem der beiden UARTs der BCU zugeordnet. Im Konzept ist jedoch vorgesehen, die Ein- und Ausgaben später mit einem eigenen CANas-Dienst (Node Service) zu tunneln, damit man die REPLs sämtlicher BCUs seiner Anwendung von zentraler Stelle aus nutzen kann. Auch für Debug-Ausgaben aus Skripten heraus ist die REPL geeignet.

4.1.3. Python-Bytecode und zentrale Skriptverwaltung

Gegenüber normalen Python-Skripten ist Python-Bytecode effizienter in Laufzeit und Arbeitsspeicherbedarf. Dabei handelt es sich um ein Python-Skript, das außerhalb der BCU vorab einen Parser durchlaufen hat und abgesehen von Zeichenketten weitestgehend in Binärcode umgewandelt wurde. Auch der Speicherplatz für Kommentare im Skript wird eingespart, da diese nicht in den Bytecode kopiert werden. Der für die Vorinterpretierung nötige Parser ist Teil des MicroPython Repositorys.

Mit der in dieser Arbeit entwickelten Skriptverwaltung der BCU kann eine beliebige Anzahl von Bytecodes unter einer vier Byte großen Kennung von außen in die BCU geladen werden. Die Anzahl und der Umfang der Bytecodes pro BCU ist aktuell allein durch den Platz im Arbeitsspeicher limitiert. Die Bytecodes werden in einer einfach verketteten Liste im MicroPython-Heap abgelegt.

Im Rahmen dieser Arbeit ist eine flüchtige Speicherung der Bytecodes im RAM gewählt worden, um eine Portierung der ROM-Anbindung vorerst zu umgehen. Für eine effizientere Nutzung des RAM und ein persistentes Speichern der Bytecodes kann eine Aufbewahrung der Codes im µC-eigenen FLASH-Speicher (ROM) oder in einer Speicherkarte implementiert werden, so wie es beispielsweise beim Pyboard umgesetzt wurde, siehe [8].

Mit einer Größe von 40 kB beschränkt der µP-Heap das Nutzungspotenzial der BCU. Um den µP-Heap zu vergrößern, ohne den zuverlässigen Betrieb der BCU zu gefährden,

bedarf es umfangreicher Analysen, Tests und sehr wahrscheinlich auch Anpassungen im Linkerskript der Firmware. Aus zeitlichen Gründen werden diese Aufgaben vorerst nicht angegangen. Die aktuelle Größe des µP-Heaps reicht für eine erste Evaluierung des Konzepts definitiv aus und die Speicherung der Bytecodes im Heap stellt einen akzeptablen Kompromiss zwischen Implementierungsaufwand und Funktionalität dar.

Auf lange Sicht lohnt es sich jedoch, die Nutzung des RAM zu optimieren und die Skripte nicht im µP-Heap zu verwalten, weil die verfügbare Menge des µP-Heaps eine stark limitierende Größe für die Ausführbarkeit von Python-Bytecodes ist.

Der Flash-Speicher, also der ROM, ist im für die BCU gewählten Mikrocontroller eine deutlich unkritischere Ressource als der RAM. Die Firmware belegt mit weniger als 500 kB nicht ganz die Hälfte des ROM. Einen Schreibzugriff auf den ROM zu implementieren und die Skriptverwaltung daran anzubinden, würde also mit über 500 kB einen über zehn mal so großen Speicherplatz für Bytecodes generieren, als aktuell der µP-Heap liefert. Außerdem stünde der µP-Heap dann in Gänze der Laufzeitumgebung zur Verfügung. Die Implementierung des ROM-Zugriffs ist daher ein wichtiger möglicher Entwicklungsschritt für die BCU, kann aber nicht innerhalb dieser Arbeit umgesetzt werden.

Die Verwaltung der Bytecodes findet in der Script-Manager Klasse statt. Sie beinhaltet die verkettete Liste der Bytecode-Objekte und sucht zu gegebener Kennung das passende Skript heraus. Außerdem gibt sie an, ob und welches Skript aktuell ausgeführt wird.

Für die Übertragung von Bytecodes wurde ein eigener CANas Dienst (Node Service) implementiert. Dieser wurde bei der Entwicklung der BCU regelmäßig eingesetzt und ist eine schnelle und komfortable Lösung, um alle BCUs eines Busses von einer zentralen Stelle aus automatisiert mit Anweisungen zu versorgen.

Es können auch andere Hardware-Schnittstellen für die Bytecode-Übertragung genutzt oder ein schreibender Zugriff auf den Script-Manager von µP aus ermöglicht werden, beides wird jedoch nicht im Rahmen dieser Arbeit implementiert.

4.2. Aufbau und Umsetzung der Toolchain

Dieser Abschnitt erläutert mit Hilfe von Abbildung 4.2 den Ablauf beim Entwicklungsprozess einer BCU basierten Applikation.

Über einen Busanschluss an beliebiger Stelle des CAN-Bussystems ist es möglich, jede BCU mit neuen Bytecodes zu versorgen oder deren REPLs anzusteuern. Das hat

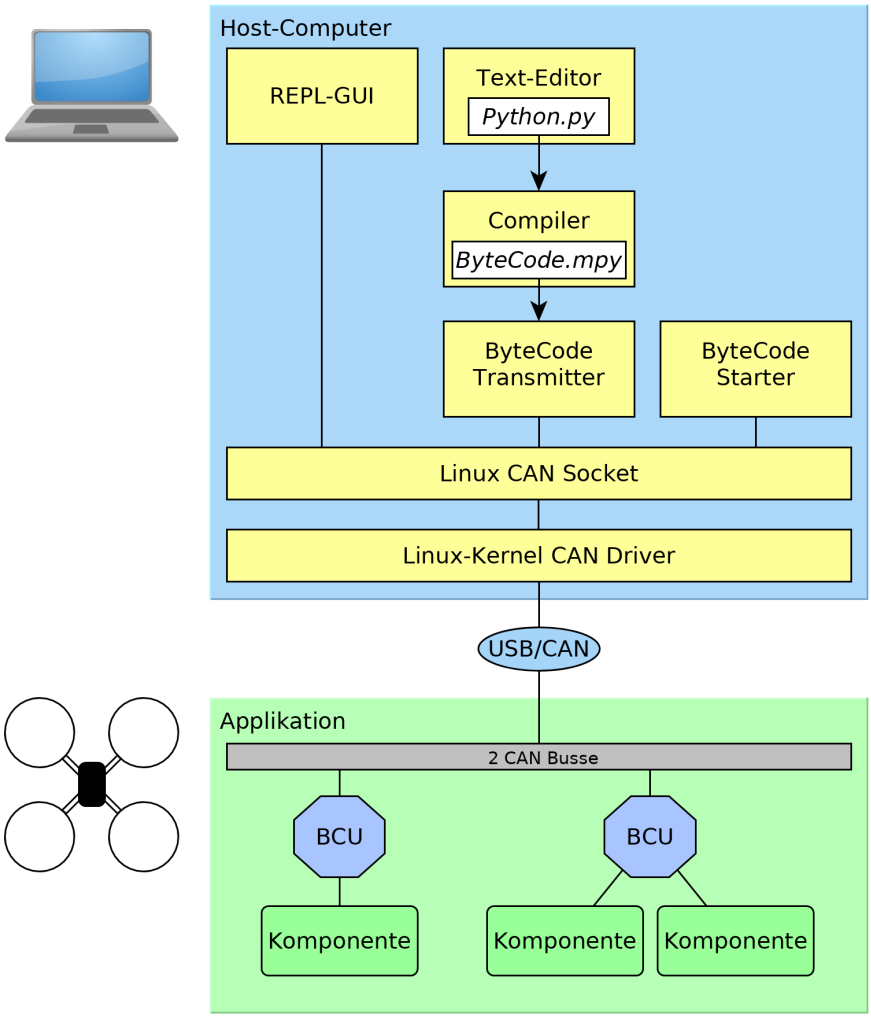


Abbildung 4.2.: Toolchain zur Entwicklung einer BCU basierten Applikation, mit Hilfe von SocketCAN werden Python-Bytecodes an die BCUs übertragen und ausgeführt

den Vorteil, dass kein direkter Zugang zu den BCUs benötigt wird, um Änderungen in ihrem Verhalten vorzunehmen und verschafft dem Anwender Flexibilität bei der Montage von BCUs innerhalb seiner Applikation.

Die Toolchain basiert auf dem Einsatz von Computern mit Linux Kerneln, weil diese das SocketCAN Framework enthalten. Damit ist es möglich, herstellerunabhängig USB/CAN-Brücken auf Basis der Berkeley Socket Application Programming Interface (API) zu betreiben. Die Grundlage hierzu ist die Integration einiger CAN-Treiber in den Linux Kernel im Zuge der Einführung von SocketCAN. Die Verwendung von USB/CAN-Brücken erhält dadurch den gleichen Komfort, wie die Nutzung von Ethernet-Schnittstellen. Das heißt, es entfallen aufwändige Installationen und es gibt im ausreichenden Maße Hardware-unabhängige Open-Source Software zur Überwachung und Manipulation von CAN-Bussystemen.

Das Beispiel auf Abbildung 4.2 soll die Funktionsweise der Toolchain und den Entwicklungszyklus verdeutlichen. Dieser besteht aus drei Schritten:

Schritt 1: Skripterstellung
Ein Python-Skript wird in einem vom Benutzer frei wählbaren Texteditor am PC erstellt.

Schritt 2: Kompilierung zum Bytecode
Ein Compiler auf dem PC liest das Python-Skript ein und wandelt es in Python-Bytecode um. Syntaxfehler werden schon hier detektiert, bevor die BCU beteiligt ist. Der Compiler selbst ist ein Linux-Kommandozeilenprogramm und sein Quellcode liegt dem MicroPython Repository bei, sodass dieser bei Bedarf modifiziert, mit einer grafischen Oberfläche versehen oder zusammen mit anderen Werkzeugen der Toolchain in ein gemeinsames Programm integriert werden kann.

Beim Kompilat handelt es sich um eine Bytecodedatei mit der Endung „mpy“, dessen Inhalt für die BCU ausführbar ist.

Schritt 3: Übertragung und Ausführung
Der Bytecode muss nun vom PC in die BCU kopiert und ausgeführt werden. Für beide Vorgänge wurde je ein Kommandozeilenprogramm von wenigen hundert Zeilen Quellcode geschrieben, „ByteCode Transmitter“ und „ByteCode Starter“. Zur Identifizierung werden die Bytecodes unter einer vier Byte großen Kennung abgespeichert. Die Kennung wird von der „Script-Manager“ Klasse ausgelesen, welche in Abschnitt 4.1.3 näher beschrieben ist.

Benötigte Informationen wie Dateipfad, Bytecodekennung und Node-ID des Zielknotens nehmen die Anwendungen als Kommandozeilenparameter entgegen. Dadurch ist die gesamte Toolchain bestehend aus Compiler, Transmitter und Starter bereits in ihrer ersten Version über Makefiles automatisierbar.

Es kann zu jeder Zeit nur ein Bytecode ausgeführt werden, aber alle Bytecodes einer BCU teilen sich einen gemeinsamen Namensraum. So sind beispielsweise „import“-Anweisungen und instanziierte Objekte eines Bytecodes auch für alle anderen gültig.

Getestet wurde die Toolchain bisher mit zwei USB/CAN-Brücken der Hersteller PEAK und ESD, welche auf Abbildung 4.3 zu sehen sind.

Abbildung 4.3.: USB/CAN-Brücken zur Kommunikation zwischen BCU und PC: IPEH-002021 von PEAK(links) und CAN-USB/Micro von ESD (rechts)

4.2.1. Fernzugriff auf interaktiven Interpreter

Neben der Übertragung fertiger Python-Skripte in Form von Bytecode ist im Konzept ein zentrales Werkzeug zur Ansteuerung der REPLs aller am Bus angeschlossenen BCUs vorgesehen. Im Rahmen dieser Arbeit wird dieses Werkzeug, nachfolgend REPL-GUI genannt.

In Abschnitt 4.1.2 ist das Tunneln von Ein- und Ausgaben der REPL erklärt. Das REPL-GUI Werkzeug ist ein Endpunkt des Tunnelns via CAN. GUI steht für Graphical User Interface. Mit dem Werkzeug soll es möglich sein, die Ausgaben der REPL einzusehen und Eingaben abzusetzen, so als wäre der PC direkt via UART mit der BCU verbunden. Somit kann die BCU in den Entwicklungsprozess von Bytecodes

eingebunden werden und diesen vereinfachen. Dies ist einer der Hauptunterschiede zum Calvin-Framework aus dem IoT-Bereich, welches nur die Ausführung von Python-Bytecode auf den Mikrocontrollersystemen vorsieht, vgl. [16], Abschnitt A.

Für das Aufschalten auf eine BCU soll es entsprechende Oberflächenelemente geben. Eine solche Anwendung könnte beispielsweise so aussehen, wie auf Abbildung A.1 dargestellt. Hierbei handelt es sich um eine funktionslose Entwurfsskizze, die mit dem Qt-Creator[1] erstellt wurde. Auf eine detaillierte Umsetzung wurde verzichtet.

[1] Qt-Creator, eine IDE aus dem Qt-Framework: https://www.qt.io/

5. Umsetzung und Verifikation

Dieses Kapitel beschreibt die technische Umsetzung der zuvor im System-Design vorgestellten Konzepte. Für die Zuordnung zwischen CAN-Identifier und Information wurde eine auf SUAV spezialisierte Identifer Distribution erarbeitet. Vorteile und Hintergründe für diese Abweichung vom in der CANaerospace-Spezifikation vorgeschlagenen Standard werden in diesem Kapitel näher beleuchtet.

Von einer ursprünglich geplanten Simulation der CAN-Buskommunikation wurde im Verlauf der Arbeit Abstand genommen. Der Hintergrund dieser Entscheidung wird im zweiten Abschnitt des Kapitels erläutert. Die aus der Aufgabenstellung abgeleiteten Anforderungen an die Hardware, dessen Aufbau, sowie die Auswahl des Mikroprozessorsystems wird im dritten Abschnitt thematisiert. Im vierten Abschnitt wird die Firmware-Implementierung beschrieben. Hierbei wird der Einsatz von Dependency Injection näher erklärt. Auch die Absicherung der Mikrocontroller-Ressourcen durch die Kapselung der µP-Laufzeitumgebung wird erläutert.

Das Kapitel endet mit einer kritischen Betrachtung von CANaerospace. Hierzu werden die Ergebnisse eines Experiments vorgestellt, welches die Auswirkungen einer Mehrfachnutzung von CAN-Identifiern untersucht.

5.1. CANaerospace Identifier Distribution für SUAVs

Die CANas Standard Identifier Distribution ist eine Zuordnung zwischen CAN-Identifiern und den gebräuchlichsten Luftfahrzeugdaten und soll in Kombination mit dem Konzept selbstidentifizierender Nachrichten eine möglichst aufwandsfreie Integration von Busknoten verschiedener Hersteller in einem CANaerospace-Bus gewährleisten, vgl. [20] S. 24.

Da es nicht ausgeschlossen ist, dass zu einem späteren Zeitpunkt Avionikkomponenten aus der manntragenden Luftfahrt an das SUAV-System angebunden oder Analysewerkzeuge für die Buskommunikation aus diesem Bereich genutzt werden, erscheint der Versuch, bei der Übertragung von Normal Operation Data (NOD) mit der

Standard Identifier Distribution möglichst kompatibel zu sein, zunächst als sinnvoller Ansatz.

Parameter, wie beispielsweise die Drehraten des Luftfahrzeugs, sind ohne Einschränkung auf SUAVs übertragbar und so werden die Parameter im Bussystem des SUAV mit dem gleichen Datentypen und mit dem gleichen Identifier versendet. Für Parameter, die nicht in der Standard Identifier Distribution zu finden sind, werden CAN-Identifier in den benutzerdefinierten Bereichen verwendet.

Der logische Kommunikationskanal für NOD erstreckt sich über den Identifier-Bereich 300 bis 1799. Im Gegensatz zu CANopen bietet CANas mit NOD den Vorteil, 1500 Nachrichten innerhalb eines logischen Kommunikationskanals unabhängig vom Absender untereinander zu priorisieren.

Der Teilabschnitt 1500 bis 1799 des Identifier-Bereichs ist mit benutzerdefinierten Parametern belegbar. Mit 300 möglichen Einträgen sind also 20% des gesamten NOD-Bereichs vom Benutzer definierbar und der Großteil mit 1200 Einträgen wird von der Standard Identifier Distribution inklusive ungenutzter aber reservierter Einträge eingenommen.

Daraus ist die in Tabelle B.1 (siehe Anhang A) dargestellte Zuordnung entstanden. Grün markierte Einträge liegen im benutzerdefinierten Bereich. Dazu zählen auch Einträge, die speziell für die UAS-Architektur aus dem Kontext dieser Arbeit definiert sind. Beispiele hierfür sind Ergebnisse aus der Sensorfusion (ID: 1520 bis 1529) und zwei Arten von Heartbeat-Nachrichten (ID: 1510 und 1570).

Für die Unmanned Aircraft System (UAS)-Architektur wurden insgesamt 72 NOD Identifier zugeordnet. Nur 21 Identifier konnten aus der Standard Identifier Distribution von CANaerospace übernommen werden. Der größte Anteil mit 71 % und 51 Nachrichten musste im benutzerdefinierten NOD-Bereich angesiedelt werden. Das Bussystem im SUAV wäre also nur eingeschränkt mit der CANas Standard Identifier Distribution kompatibel.

Bei der Zuordnung von Tabelle B.1 sind die meisten Informationen mit nur einem Identifier vertreten, obwohl sie je nach Bauart des SUAV durchaus von acht verschiedenen Sensorquellen redundant auf den Bus geschrieben werden müssen.

Bei einem Multirotor-SUAV in Form eines Octocopters und einer Architektur, bei der in allen acht Armen jeweils die gleichen Komponenten verbaut sind (siehe Abb. 2.1), müssten beispielsweise die gemessenen Beschleunigungen von acht Accelerometern zur Plausibilitätsprüfung und zur Sensorfusion über den Bus verbreitet werden. Löst man diesen Identifier-Mangel über das Redundanzkonzept von CANaerospace, verliert man

die ursprüngliche Priorisierung der Informationen bei der Arbitrierung, siehe Abschnitt 3.4 und Abschnitt 3.4.1.

Im benutzerdefinierten Bereich ist nicht genug Platz, um dort alle benötigten Informationen durch vielfache Zuordnung priorisiert und gleichzeitig ausreichend redundant unterzubringen.

Zusätzlich fällt auf, dass die Einträge aus der Standard Identifier Distributionen nicht durchgehend mit den Prioritätsbewertungen der eigenen Zuordnung übereinstimmen. Die Priorität aller Einträge für das Flugsystem ist vierstufig von Stufe 3 (geringe Priorität) bis Stufe 0 (essenziell für einen sicheren Flugbetrieb) eingeteilt.

Beispielsweise sind die Sollwerte für die Motordrehzahlen, welche von der Control Service Allocator Unit (CAU) ausgegeben werden, ca. der Hälfte aller Nachrichten untergeordnet, obwohl sie für die Steuerung des SUAV unverzichtbar sind.

Daher ist es angebracht, die gewünschte Kompatibilität mit der Standard Identifier Distributionen aufzugeben, um eine eigene SUAV Identifier Distribution zu erstellen, welche die eigenen definierten Prioritäten einhält und somit die Sicherheit für CANas basierte SUAV erhöht. Ziel dabei ist es, alle nicht grün markierten Einträge aus Tabelle B.1 zu übernehmen und konfliktfrei zu vervielfältigen. Dadurch wäre für bestimmte Informationen eine Übereinstimmung mit der Standardzuordnung vorhanden. Eine Kompatibilität mit CANas Komponenten aus der manntragenden Avionik ist trotz Abweichung von der Standard Identifier Distribution durchaus möglich und wahrscheinlich, da diese Komponenten bei Unterstützung des CAN Identifier Setting Service (CSS) die Möglichkeit bieten, die verwendeten CAN-Identifier für das jeweilige Bussystem anzupassen. Der benutzerdefinierte Bereich bliebe zu 100 % verfügbar und trotz Redundanz bliebe die Priorisierung der Informationen erhalten.

Eine nach diesem Ansatz erstellte Zuordnung ist im Anhang in Tabelle B.2 zu finden. Zwischen den vier Priorisierungsgruppen (0 bis 3) ist je ein Abschnitt im Identifier-Adressbereich für spätere Erweiterungen frei gelassen worden. Insgesamt umfasst die Zuordnung bisher mit 347 genutzten Identifiern 29 % der möglichen Einträge außerhalb des benutzerdefinierten Bereichs.

Wie die Nachrichten angeordnet sind, ist in Grafik B.1 (siehe Anhang) illustriert. Die Darstellung zeigt den gesamten Identifier-Adressbereich für NOD. Der kleinste Identifier 300 mit der höchsten Priorität steht oben in der ersten Pixelzeile und der größte Identifier 1799 mit der geringsten Priorität steht unten.

Um die Echtzeitfähigkeit der Kommunikation zu garantieren, wurde eine theoretische Buslastbetrachtung erstellt. Hierzu wurde die Identifier Zuordnung von Tabelle B.1 aus

dem ersten Ansatz unter Verwendung des CANas-Redundanzkonzeptes mit 29-Bit Identifiern als Basis genutzt. Redundant auftretende Informationen erzeugen somit durch verlängerte Frames mehr Buslast. Für die Bitrate 1 Mbit/s angenommen. Die Datenlast wurde für ein SUAV mit vier Armen und acht Motoren modelliert, wie es auf Abbildung 2.1 zu sehen ist. Redundante Daten werden also vierfach übertragen.

	Nachrichtenlänge l	max. Nachr./s	Buslast
Standard	125 Bits	8000	46 %
Extended	144 Bits	6944	52 %

Tabelle 5.1.: Theoretische Buslastbetrachtung für ein SUAV mit vier Armen und acht Motoren, einer Bitrate von 1 Mbit/s und einer Datenlast von 3645 Nachrichten/s

Die für die Flugregelung essenziellen Daten werden mit 50Hz übertragen und machen den Großteil der Datenlast aus. Andere Informationen werden mit Aktualisierungsraten zwischen 0,1 Hz und 10 Hz modelliert. Die erwartete Nachrichtenlast n summiert sich dadurch auf ca. 3645 Nachrichten pro Sekunde.

Die resultierende Buslast wurde für Standard 11-Bit und Extended 29-Bit Identifier und Datenframes mit 8 Nutzdatenbytes berechnet, zu sehen in Tabelle 5.1. Die Buslast ergibt sich aus $\frac{n*l}{Bitrate}$.

Um die theoretischen Aussagen über die Buslast verifizieren zu können, wurde eine Firmware für die BCU erstellt, die den modellierten Datenstrom über eine ihrer beiden CAN-Schnittstellen aussendet. Um eventuell auftretende Latenzen zu vermeiden, nutzt die Firmware C++-Methoden für die Generierung der Nachrichten und verzichtet auf den Einsatz von MicroPython.

Die Buslast wurde mit dem CAN-Analyzer usbMicro von ESD (siehe Abb. 4.3) ermittelt. Wie auf dem Screenshot in Abbildung 5.1 zu sehen ist, liegt der Messwert mit 48 % zwischen den beiden theoretischen Werten von Tabelle 5.1. Dies liegt an der gemischten Buslast aus Standard- und Extended-Dataframes.

Die Messung hat gezeigt, dass sich mit der theoretischen Buslastbetrachtung ausreichend präzise Vorhersagen treffen lassen, um bei der Busauslegung genügend Laufzeit für alle Nachrichten einplanen zu können.

Overview	General
Bus load	48.31 %
kBit/Second	483.087 /s
Frames/Second	3754 /s
Total Frames	2941492
Error Frames	0
Frame Rate	Receive
Standard	1011 /s
Standard RTR	0 /s
Extended	2687 /s
Extended RTR	0 /s
Total	3698 /s

Abbildung 5.1.: Screenshot der Buslastmessung durch den CAN-Analyzer microUSB und dem Programm CANreal von ESD

5.2. Simulation von CAN-Bussystemen

Um die Eignung von CANaerospace für die Systemarchitektur besser evaluieren zu können, war zunächst eine Simulation des Bussystems geplant. Dabei sollte die eigene Protokollimplementierung, die später in der Firmware die CAN-Kommunikation übernimmt, Bestandteil der simulierten CAN-Knoten sein.

Die dazu notwendige Simulationsumgebung muss in kurzer Zeit einzurichten sein und darf nur geringe Aufwände bei der Einarbeitung erzeugen. Außerdem soll sie frei verfügbar und möglichst quelloffen sein, damit man ihre Funktionsweise bei Bedarf überprüfen und ggf. ändern kann. Eine effiziente Möglichkeit zur Darstellung und Analyse der Simulationsergebnisse ist ebenfalls gefordert.

Das Verhalten der zu simulierenden Knoten soll sich durch C++-Klassen definieren lassen, um so die Simulation direkt mit den CANaerospace Firmwarekomponenten zu betreiben. Auf diese Weise dient die Simulation zusätzlich als Komponententest für die Protokollumsetzung und es wird Implementierungsaufwand eingespart.

Eine Recherche hat ergeben, dass nur das Simulations-Framework OMNet++ in Kombination mit dem CAN-Busmodell FiCo4OMNeT[1] die Anforderungen erfüllt. Das Framework bietet eine generische Simulationsumgebung mit grafischer Oberfläche und eine Entwicklungsumgebung für die Erstellung von Netzwerk-Modellen. FiCo4OMNeT ist ein solches Modell. Es ist modular aufgebaut und bildet das Verhalten eines CAN-Busses ab. Die Arbitrierung und andere Grundlagen sind in verschiedenen C++-

[1] Fieldbus Communication For OMNeT, siehe [17]

Klassen umgesetzt. Über entsprechende Softwareschnittstellen lassen sich in das Modell eigene Knoten-Klassen integrieren, welche die zu evaluierende CANaerospace-Implementierung beinhalten können.

Während der Einarbeitung in das Modell und der Entwicklung der CANaerospace Knoten fiel jedoch auf, dass im Modell das Busverhalten bei der Arbitrierung in einem speziellen Fall nicht hinreichend abgebildet ist. Wie in Abschnitt 3.4.1 bereits erwähnt wurde, kann es bei unvorsichtiger Verwendung des CANas Protokolls vorkommen, dass mehrere Knoten zur gleichen Zeit eine Nachricht mit dem gleichen Identifier und unterschiedlichen Nutzdaten zu senden versuchen. Die Nachrichten unterscheiden sich somit nur außerhalb des Arbitrierungsfeldes und es käme zu einer unaufgelösten Kollision. Beide Knoten würden ihre Transmission Error Counts inkrementieren und danach einen erneuten Sendeversuch unternehmen.

Die Simulation hingegen gibt in diesem Fall nur eine warnende Textnotiz aus und lässt den in der internen Logik als erstes registrierten Sender erfolgreich seine Nachricht senden. Es kommt in der Simulation somit nicht zu einer Kollision. Dieses Verhalten ist in der Klasse „CanBusLogic" des Modells implementiert. Das Listing C.1 (siehe Anhang) zeigt die betreffende Methode „grantSendingPermission". Die ausgegebene Warnung ist in Zeile 34 zu sehen.

Durch die Quelloffenheit des Modells, besteht die Möglichkeit einer Anpassung zur korrekten Kollisionsabbildung. Da zu diesem Zeitpunkt bereits mehrere Knoten als Hardware und die USB/CAN-Brücken (siehe Abbildung 4.3) zur Verfügung standen, war es ab sofort möglich die nötigen Erkenntnisse durch Messung und Manipulation an einem realen CANas-Bussystem zu gewinnen. Der Aufwand einer Simulationsanpassung in Kombination mit der vorherigen Einarbeitungszeit in die Simulationsumgebung und in das CAN-Busmodell lässt sich somit nicht mehr rechtfertigen. Daher wurde die Simulation des Bussystems nicht durchgeführt.

5.3. Aufbau der Hardware

Für die Firmwareentwicklung der BCU wird eine Mikrocontroller-Platine benötigt. Da einige Randbedingungen, wie Beschaffenheit der Steckverbinder, zu Beginn nicht feststanden, wurde der Hardware-Entwurf in zwei Schritte unterteilt.

Der erste Schritt verwendet eine COTS-Platine, auf welcher der µC verlötet ist. Hierfür wurde eine Adapter-Platine nach den vom Autor dieser Arbeit entwickelten Vorgaben gefertigt. Die Bestückung mit der nötigen Peripherie wurde vom Autor selbst

durchgeführt. Hierzu zählen im Wesentlichen Pull-Up Widerstände, Steckverbinder, Load-Switches für 3 verschiedene Versorgungsspannungen der externen Komponenten und zwei CAN-Transceiver.

Im zweiten Schritt wird das Commercial Off-The-Shelf (COTS)-Bauteil und die Adapter-Platine zu einer Leiterplatte vereint. Dies geschieht, sobald die Schnittstellen vollständig definiert sind. Bei diesem Schritt entfallen bestimmte Bauelemente, wie der Linearspannungsregler und die USB-Schnittstelle. Auch Größe und Anzahl von LEDs werden sich verringern. Ein Ziel bei der Entwicklung ist es, die Baugröße der BCU klein zu halten. Bei diesem zweiten Schritt ist eine Verringerung der Baugröße durch die entfallenden Bauelemente zu erwarten, diese wird jedoch durch den Einbau eines SD-Speicherkartensteckplatzes kompensiert.

Die Speicherkarte soll die Verwendung der BCU als Datenlogger ermöglichen. Mit einer erwarteten Schreibrate von mehr als 2Mbit/s können die Nachrichten beider CAN-Busse zusammen mit anderen Daten in Echtzeit mitgeschrieben werden. Außerdem eignet sich eine SD-Karte für die persistente Speicherung und Veränderung von Python-Skripten zur Laufzeit, so wie es beispielsweise mit dem Pyboard möglich ist, vgl. [8].

Dieser zweite Schritt erfolgt im Nachgang dieser Arbeit, nachdem das erarbeitete Konzept hinreichend validiert ist.

5.3.1. Anforderungen an die Adapterplatine

Je kleiner die BCU-Platine gestaltet ist, desto besser ist sie für die Verwendung in SUAVs geeignet. Vor allem bei Multirotorsystemen, die ihren Auftrieb ohne Tragflügel direkt durch ihre Antriebsleistung generieren müssen, wirkt zusätzliches Gewicht immer verkürzend auf die Flugzeit. Daher lohnt es sich, Bauraum und Gewicht zu sparen.

Einer geringen Baugröße stehen Anforderungen aus den Bereichen Fertigung und Wartbarkeit entgegen. Surface Mount Device (SMD) Bauteile sind erlaubt, jedoch muss die Platine vollständig manuell bestückbar gestaltet sein. Außerdem ist es von Vorteil Steckverbinder mit Kontakten im Rastermaß 2.54mm zu wählen, sodass während der Entwicklung einfache Laborkabel (Breadboard Jumper) aufgesteckt werden können.

5.3.2. Auswahl des Mikrocontrollers

Für den Betrieb der µP-Laufzeitumgebung wird in der einfachsten Variante mindestens 2 Kilobyte Arbeitsspeicher (RAM) benötigt. Um genug Funktionsumfang implementieren zu können und um möglichst umfangreiche und komplexe Python-Skripte verarbeiten

zu können, ist es jedoch sinnvoll, einen µC mit deutlich mehr RAM zu verwenden. Mit mindestens 8 kB Arbeitsspeicher wird sichergestellt, für spätere Erweiterungen genug Ressourcen übrig zu haben, vgl. [7]. Bei der Auswahl des Mikrocontrollers hinsichtlich seiner RAM- und ROM-Größen diente das Pyboard (siehe [8]) zur Orientierung.

Der µC muss folgende Anforderungen erfüllen:

- Debugging-Funktionen mit Speicherauslesefunktion und Einzelschrittausführung
- Große Auswahl verschachtelbarer, priorisierbarer Interrupts
- Integrierte Schnittstellen im µC:
  - Zwei oder mehr UARTs
  - Ein SPI mit mindestens 2 Chip Select Leitungen
  - Eine I²C Schnittstelle
  - Zwei CAN-Controller (Bitrate: 1MBit/s)
  - DMA-Fähigkeit möglichst aller Schnittstellen
  - Anbindung einer SD-Karte mit über 2Mbit/s (z. B. SDIO)
- Energiesparfunktionen (Teilabschaltbarkeit, Taktfrequenzkonfiguration)

5.3.3. Mikrocontroller-Platine Mini-M4

Die in Abschnitt 5.3.2 spezifizierten Anforderungen werden von der Entwicklungsplatine Mini-M4 des Herstellers MikroElektronika hinreichend erfüllt. Der verwendete µC STM32F415RG verfügt über alle geforderten Schnittstellen, vgl. Datenblatt [3], S. 1 und bietet in ausreichendem Maße Möglichkeiten zur Implementierung von DMA-Zugriffen auf die einzelnen Schnittstellen, vgl. Datenblatt [3], Abschnitt 2.2.8, S. 23 ff. Außerdem verfügt er über priorisierbare und verschachtelbare Interruptquellen, vgl. Datenblatt [3], Abschnitt 2.2.10, S. 24 ff.

Zusätzlich ist die Platine mit 17,78 mm x 50,8 mm kompakt gebaut und wiegt ca. 6g. Die Stiftkontakte an der Unterseite sind kompatibel zu einem DIP40 Gehäuse, wodurch sich die Platine auf einem Steckbrett im Rastermaß 2.54 mm einsetzen lässt.

Die Platine bietet eine USB-Buchse mit Anbindung an die OTG-USB-Schnittstelle des µC. Diese kann zur Spannungsversorgung, Programmierung via integriertem USB-Bootloader (DFU) oder je nach Implementierung zur Laufzeit für eigene Zwecke genutzt werden.

Die herausgeführten µC-Pins und die onboard-Peripherie ist auf Abbildung 5.2 zu sehen. Im unteren Bereich der Aufnahme erkennt man den Reset-Taster und darüber drei Light Emitting Diodes (LEDs), zwei davon sind vom µC ansteuerbar, die dritte leuchtet bei anliegender Versorgungsspannung.

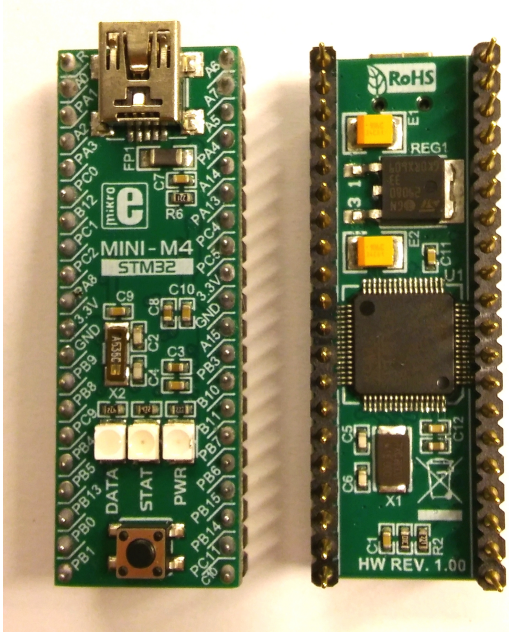


Abbildung 5.2.: Mikrocontroller-Platine Mini-M4 [24] von Mikro Elektronika, Ober- u. Unterseite, das gezeigte PCB ist mit dem Mikrocontroller STM32F415 und der nötigen Minimalperipherie wie z. B. den Quarzen bestückt und vereinfacht dadurch die Entwicklung einer ersten BCU-Platine

5.3.4. Adapterplatine für das Mini-M4 Board

Die Platine auf Abbildung 2.3 stellt den ersten Entwicklungsschritt aus Abschnitt 5.3 dar. Die Aufnahme zeigt das noch unbestückte PCB.

Der Autor dieser Arbeit hat die Anforderungen an das PCB-Layout entwickelt und den Erstellungsprozess für das Layout geführt. Das Layout und das PCB selbst wurde jedoch extern angefertigt.

Auf Abbildung 5.3 sieht man die vollständig bestückte Platine mit dem Mini-M4 Board, den Wannensteckern für die externen Komponenten, den austauschbaren CAN-Terminator-Widerständen und den austauschbaren Pull-Up/Down Widerständen für die beiden I²C-Leitungen und die SPI-MISO-Leitung.

Die CAN-Terminatoren austauschbar zu gestalten, ermöglicht Flexibilität in der Bustopologie und der BCU Position innerhalb eines CAN-Busses.

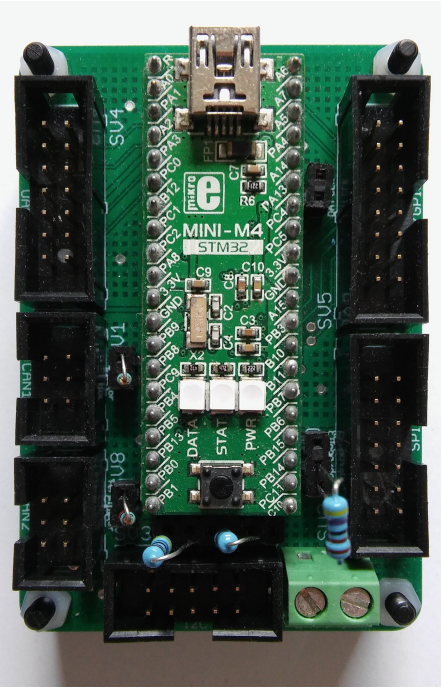


Abbildung 5.3.: Adapterplatine fertig bestückt [23], mit Mini-M4 Entwicklungsboard und den flexibel aufsteckbaren Pull-Up-/Down- und Terminator-Widerständen

5.4. Firmware-Implementierung

Die Umsetzung der Firmware wird hier näher beschrieben. Dabei wird auf die Portierung von MicroPython auf die Hardware eingegangen und das Konzept des verwendeten HAL vorgestellt. Er ist, wie der größte Teil der Firmware, nach dem Prinzip der losen Kopplung mit Hilfe von Dependency Injection gestaltet, siehe [14].

5.4.1. Lose Kopplung und Abstraktion von Hardwarezugriffen

Durch lose Kopplung in der Softwarearchitektur werden miteinander kollaborierende Klassen einander nicht fest zugewiesen, sondern über Softwareschnittstellen austauschbar verbunden. Dies erhöht die Wartbarkeit, sowie die Portierbarkeit und vereinfacht

die Implementierung von Komponententests, weil sich die zu testende Komponente leichter isolieren lässt und man ihre eigentlichen Kollaborateure durch Komponenten einer Testumgebung ersetzen kann.

Realisiert wird diese lose Kopplung in der Firmware über Constructor-Injection. Damit wird eine spezielle Art von Dependency Injection bezeichnet, bei der alle Kollaborateure einer Klasse als Parameter in ihrem Constructor übergeben werden. Die Parametertypen können Interface-Klassen sein. Dadurch kann man die Kollaborateure beliebig austauschen, da man jedes Objekt übergeben kann, sofern dessen Klasse das Interface implementiert. Der komfortable Austausch vereinfacht Firmware-Änderungen und Komponententests.

Die Firmware basiert auf einem objektorientierten HAL, der vollständig mit Constructor-Injection arbeitet. Dieser wurde nicht vom Autor entwickelt, sondern entstand im Jahr 2014 im Rahmen einer Arbeit über Flugregler, siehe [9]. Er bietet Klassen zur Nutzung von SPI-, I²C- und UART-Schnittstellen im synchronen Betrieb oder asynchronen Betrieb via Interrupts oder Direct Memory Access (DMA) Zugriff. Auch Zugriff auf GPIO Pins und Hardware-Timer bietet der HAL in Klassen abstrahiert an.

Für die Implementierung der BCU wurden Klassen zum Interrupt-gestützten Betrieb beider CAN-Schnittstellen des µC vom Autor dieser Arbeit hinzugefügt. Außerdem sind elementare Bestandteile des Übertragungsprotokolls CANaerospace (CANas) mit in den HAL aufgenommen worden. Bei der Protokollintegration wurde eine Möglichkeit geschaffen, CANas später gegen andere Übertragungsprotokolle zur Laufzeit austauschen zu können. Dieser Ansatz ist in Abschnitt 5.4.3 näher beschrieben.

5.4.2. Kapselung der MicroPython-Laufzeitumgebung

Mit Hilfe von zahlreichen Macros lassen sich sämtliche Funktionen und Erweiterungen von MicroPython vom Kompilat ausschließen und es gibt eine Minimal-Konfiguration im Sourcecode-Repository, die es Entwicklern vereinfachen soll, MicroPython auf neue Hardware zu portieren, bzw. es in andere Umgebungen einzubetten.

Die MicroPython-Implementierung bringt jedoch eine eigene C Hardware-Abstraktionsschicht mit, welche nur bedingt austauschbar umgesetzt ist. Um diese von den darüber liegenden Komponenten zu trennen, damit sie mit dem eigenen C++-HAL ersetzt werden kann, wurde der größte Teil der verfügbaren Zeit für die BCU-Implementierung aufgewendet.

Der verwendete ARM Cortex M Mikrocontroller kann sämtliche Peripheriefunktionen, wie externe Interrupts, UART-, SPI- oder andere Schnittstellen auf verschiedene

GPIO-Pins schalten. Diese Flexibilität ist auch in der MicroPython-Implementierung und somit in dessen HAL verankert. Der Benutzer kann sich beispielsweise ein CAN- oder UART-Objekt in Python instanziieren und eine entsprechende Schnittstelle auf den dafür geeigneten Pins initialisieren. Die BCU hingegen soll diesen Freiheitsgrad nicht bieten. Sie soll möglichst simpel im Betrieb sein und den Benutzer insofern einschränken, dass er mit einer einzelnen BCU möglichst nicht dazu in der Lage ist, die gesamte Buskommunikation zu blockieren. Daher müssen die originalen Implementierungen von CAN, UART, SPI, I²C, GPIO, PWM und Timern durch Anbindungen an die eigene C++ Hardwareabstraktionsschicht (HAL) ersetzt werden. Diese Anbindungen sind in C implementiert und in der Abbildung 4.1 in Abschnitt 4.1 als „MicroPython builtin Modules" dargestellt. Aufrufe werden von MicroPython aus von diesen Modulen an die entsprechenden C++ Methoden delegiert und anders herum werden via Callback-Mechanismen Interrupt-Ereignisse von der Hardware über diese Module in die MicroPython-Laufzeitumgebung weitergeleitet. Im Klassendiagramm auf Abbildung 5.4 in Abschnitt 5.4.3 ist dieser bidirektionale Übergang zwischen µP-Interpreter und der restlichen C++ Firmware als Notiz „MicroPython C calls" angedeutet.

Diese Module unterstützen keine Instanziierungsoperation mehr und weichen damit von der ursprünglichen MicroPython-Implementierung ab. Dadurch wird die gewünschte Einschränkung des Benutzers zur Absicherung der Buskommunikation garantiert. Beide CAN-Controller des Mikrocontrollers sind über die gesamte Laufzeit aktiv und entsprechenden GPIO-Pins zugewiesen. Dadurch ist es von µP aus nicht mehr möglich, den CAN-Bus dauerhaft mit einem dominanten Pegel zu blockieren, weil sich die Leitungen zum CAN-Transceiver ausschließlich über die CAN-Module ansteuern lassen und man diese Leitungen nicht mehr als digitalen Ausgang nutzen kann. Im folgenden Abschnitt 5.4.3 ist die Implementierung der CAN-Module näher beschrieben.

Ein weiterer Vorteil besteht in der vereinfachten Nutzungsweise der Schnittstellen. Der Anwender kann die Module nach einem entsprechenden Python Import-Aufruf sofort nutzen, ohne sich um eine Instanziierung zu kümmern. Das Ausmaß der eingesparten Aufwände beim Benutzer wird bei Betrachtung der Schnittstellen-Optionen im Datenblatt des µC deutlich. Die GPIO-Pins des µC müssen so zugewiesen werden, dass sich die Schnittstellen nicht gegenseitig ausschließen. Nicht jede Ressource kann jedem Pin zugewiesen werden. Es müssen Timer und Timerkanäle, DMA-Controller und deren Streams, sowie auch Konfigurationsmöglichkeiten von Sonderfunktionen, wie z. B. Clock-Leitungen konfliktfrei zugeordnet werden. Die nötige Recherche im Datenblatt und die Koordinationsarbeit bleibt dem Benutzer bei der Verwendung der BCU erspart.

5.4.3. Integration von CANaerospace und anderen Protokollen

Die Einbettung der beiden CAN-Controller in die Firmware nimmt gegenüber anderer Hardware-Schnittstellen einen besonderen Stellenwert für die BCU ein. Zum Einen ist die redundante CANaerospace-Kommunikation eine Voraussetzung für die Nutzung aller anderen BCU-Schnittstellen innerhalb des SUAV als Gesamtsystem. Zum Anderen basiert die zentrale Skriptverwaltung und damit die Verhaltenssteuerung aller BCUs des SUAV auf einer Bytecode-Verteilung via CANas-Protokoll.

Die CAN-Controller nur als ansteuerbare Hardware in der MicroPython-Umgebung anzubieten, wie es bei UART, SPI und anderen Schnittstellen umgesetzt ist, reicht hier nicht aus. Wegen der Skriptverwaltung muss auch die Firmware selbst CAN-Nachrichten auslesen, interpretieren und versenden können, möglichst ohne der µP-Laufzeitumgebung CAN-Nachrichten vorzuenthalten oder die Nutzung der CAN-Schnittstellen einzuschränken.

Der hierzu entwickelte Lösungsansatz ist in diesem Abschnitt genauer erklärt.

Die Basis der CAN-Kommunikation bildet die Klasse CANnodeRAW, siehe Abbildung 5.4. Sie abstrahiert beide CAN-Controller gemeinsam und stellt die Grundfunktionen der Hardware zur Verfügung. Eingehende Nachrichten müssen die Empfangsfilter des µC passieren und liegen danach in einem von zwei Hardware-FIFO-Speichern zum Auslesen bereit. Gefiltert wird nach Inhalten im Arbitrierungsfeld der Nachricht. In welchem FIFO-Speicher die Nachrichten landen, hängt davon ab, welchen Filter sie passiert haben. Grundsätzlich kann also jeder CAN-Controller eingehende Nachrichten in jeden FIFO-Speicher schreiben.

Je nach Applikation kann es sinnvoll sein, die Filter via Python konfigurierbar zu gestalten. Im Rahmen dieser Arbeit wurde dafür jedoch keine Schnittstelle implementiert und die Filter statisch so konfiguriert, dass alle Nachrichten die Filter passieren und somit einen Empfangsinterrupt auslösen.

Die Klasse CANnodeRAW hat Zugriff auf zwei CAN_IRQ Instanzen, welche je einen CAN-Controller abstrahieren. Je nachdem mit welchem Controller eine Nachricht zu versenden ist, wird in CANnodeRAW die sendMsg-Methode des entsprechenden CAN_IRQ-Objekts aufgerufen.

Für den Empfang von Nachrichten bietet die CAN_IRQ-Klasse die Möglichkeit, Callback-Methoden zu registrieren, welche vom CAN_IRQ-Objekt aufgerufen werden und die empfangene Nachricht als Parameter übergeben bekommen. Die CANnodeRAW-Klasse enthält vier Callbacks und registriert diese bei den CAN_IRQ Instanzen, um so die Basis zur Verarbeitung von CAN-Nachrichten zu schaffen.

Die Firmware muss eingehende Nachrichten für die Skriptverwaltung auslesen können und dem Benutzer sollen die Nachrichten in zwei logischen Ebenen angeboten werden können. Die untere Ebene bietet eingehende Nachrichten ohne Vorinterpretierung an, das heißt Nachrichtentypen werden nicht unterteilt und es gibt nur eine Callback-Methode für alle eingehenden Nachrichten. Die obere Ebene liefert eine vorinterpretierte Nachricht aus.

Bei Protokollen wie CANas oder CANopen können Nachrichten dann einzelnen logischen Kommunikationskanälen (LCCs) zugeordnet und über entsprechende Callbacks empfangen werden. Bei CANas beispielsweise kann der Benutzer sich nicht nur für Nachrichten vom Typ „CanRxMsg" mit einem Callback registrieren, sondern er kann Nachrichten vom Typ „CANasMsg" erhalten, welcher die Informationen der Nachricht bereits in Nutzdaten und einzelnen Header-Bestandteilen eingeordnet hat. Im Rahmen einer ersten Umsetzung schlägt sich dies im unterschiedlichen Aufbau der Tupel nieder, welche die Nachrichteninhalte in µP im Callback ausliefern. Bei CANas kann es somit einzelne Callbacks für verschiedene Nodeservices, Emergency-Nachrichten und Normal Operation Data geben.

Bei CANopen ist wieder ein anderes Format der Tupel denkbar mit Callback-Methoden für verschiedene PDOs und anderen LCCs.

Umgesetzt wird dieses Konzept der variablen Nutzung von Übertragungsprotokollen durch eine auf CAN-Nachrichten beschränkte Form des Observer-Patterns, siehe [6], Kapitel 5. Hierzu ist die CANnodeRAW-Klasse von der CANrxDistributor-Klasse abgeleitet und enthält damit variable Listen und Methoden zur Registierung beliebig vieler Observer für den Nachrichtenempfang pro FIFO und CAN-Controller.

Jede Klasse, die das Interface 2CANrxObserver implementiert, kann die attach-Methode des Distributors aufrufen und sich als Callback-Objekt registrieren. Die CANrxRAWuPConnection-Klasse ist von diesem Interface abgeleitet und bietet eine CAN-Anbindung ohne höheres Protokoll und ohne Vorinterpretierung an. Bei Verwendung dieser Anbindung muss die gesamte Protokoll-Logik in Python umgesetzt werden.

Die Klasse CANnodeAS bietet die wichtigsten Grundfunktionen von CANaerospace an. Sie ist ebenfalls vom CANrxObserver abgeleitet und soll dem Benutzer Entwicklungsaufwände einsparen. Beispielsweise stellt sie eine Methode zum Versenden von Normal Operation Data Nachrichten (NOD) zur Verfügung und zählt beim Versand selbstständig die Message Number hoch. Außerdem werden die für das UAV definierten Heartbeats automatisch im benutzerdefinierten Intervall versendet. Im Konzept

ist auch vorgesehen, die Nachrichten wie zuvor erwähnt vorsortiert auf verschiedene Callbacks zu verteilen.

Im Rahmen dieser Arbeit existiert nur eine CANaerospace-Implementierung, aufgrund des Observer-Patterns ist es jedoch einfach möglich weitere koexistierende Protokoll-Implementierungen in die Firmware zu integrieren und in MicroPython anzubinden.

Genau wie CANaerospace würde auch jede weitere Protokollklasse eine Referenz zur ScriptManager-Klasse erhalten, um die zentrale Skriptverwaltung zu ermöglichen.

Die in diesem Abschnitt beschriebene Klassenstruktur ist in Abbildung 5.4 als Klassendiagramm dargestellt. Die Klassen, die über Aufrufe von C Funktionen direkt mit der MicroPython-Laufzeitumgebung interagieren, sind mit der Notiz '"MicroPython C Calls“ markiert.

Eine Anbindung des ScriptManagers selbst an MicroPython steht noch aus. Durch eine entsprechende Firmwareanpassung ließe sich diese aber nachrüsten. Dadurch wäre die zentrale Skriptverwaltung unabhängig vom CAN-Bus erweiterbar. Ein zu Beginn der Laufzeit vorhandenes Skript könnte beispielsweise eine über SPI oder I²C angeschlossene Bytecodequelle auslesen und den so empfangenen Bytecode via ScriptManager abspeichern und ausführen. Eine solche Quelle kann ein Ethernet-Controller oder ein Transceiver-Modul für drahtlose Verbindungen sein.

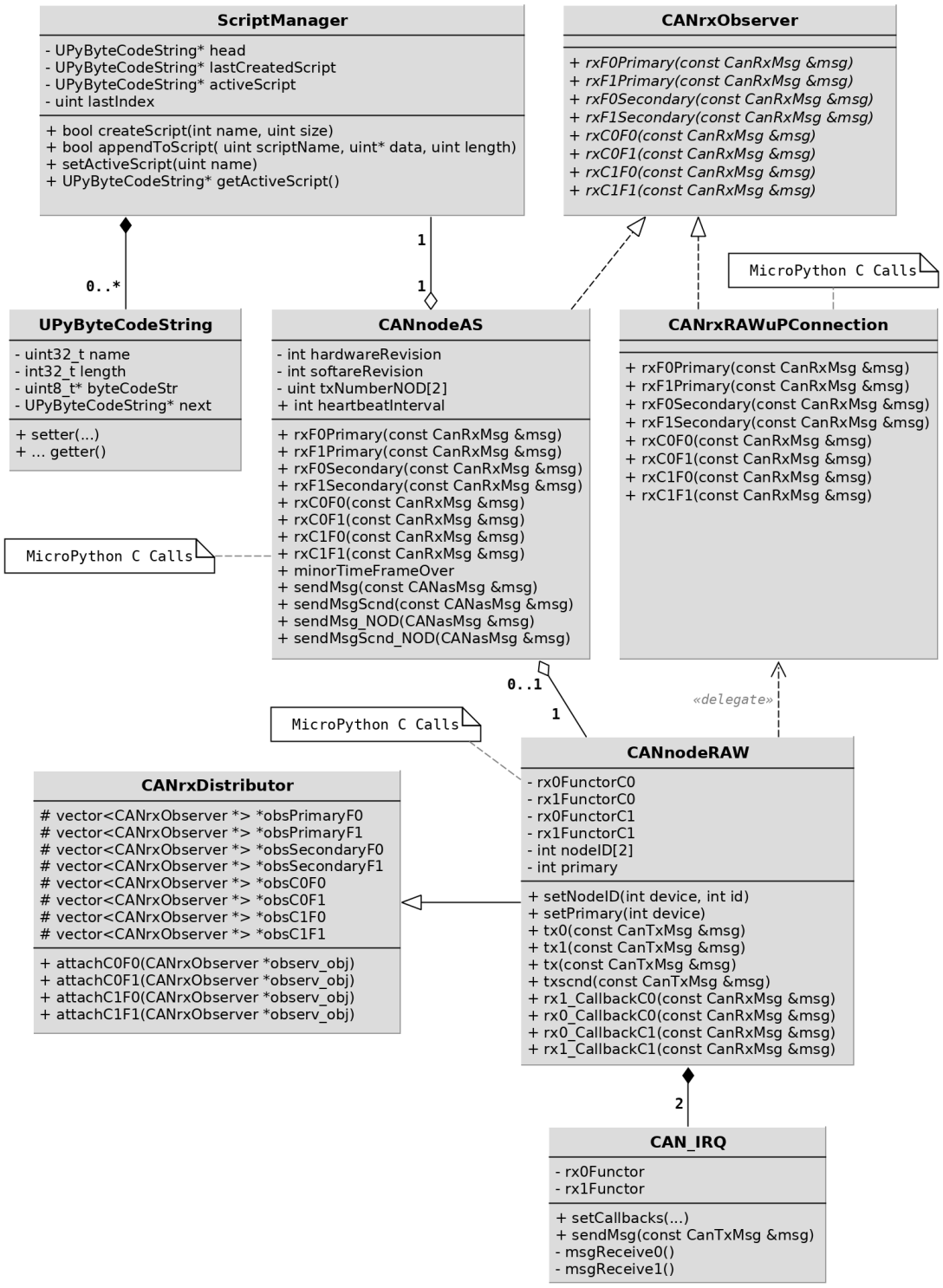


Abbildung 5.4.: Klassendiagramm der CAN- und ScriptManager-Anbindung – Durch das Observer-Pattern ist eine einfach erweiterbare Architektur zur parallelen Nutzung mehrerer Protokolle geschaffen worden.

5.5. Kritische Betrachtung von CANaerospace

Eine problematische Eigenschaft von CANaerospace besteht in der Provokation unauflösbarer Nachrichtenkollisionen bei unbedachter Anwendung in Bussystemen mit ausreichend hoher Knotenanzahl. Wie diese Konflikte entstehen und welche Auswirkungen sie auf die Buskommunikation haben, wird in diesem Kapitel analysiert. Entsprechende Empfehlungen für die sichere Nutzung von CANaerospace werden im Anschluss aufgezeigt.

Bei den Node Services, welche vor allem bei der Inbetriebnahme des Bussystems in der Initialisierungsphase zum Einsatz kommen, kann es zu Handshake-Konflikten und damit zu unauflösbaren Kollisionen kommen, obwohl sich alle Knoten gemäß der Spezifikation verhalten. Nach der Initialisierung finden Node Services eher selten und daher kaum parallel Anwendung. Das liegt daran, dass die Node Services aus der Spezifikation Konfigurationsmöglichkeiten anbieten und keine normalen Prozessdaten transportieren. Das entschärft das Problem. Es ist aber laut Spezifikation nicht verboten, auch später während des Fluges Node Services zu verwenden. Außerdem gibt es die Möglichkeit, benutzerdefinierte Node Services für beliebige Zwecke zu implementieren, welche je nach Anwendungszweck auch zum Gebrauch während des Flugbetriebs verleiten können.

Wie relevant die zuverlässige Echtzeitkommunikation des Bussystems für die Sicherheit des Flugbetriebs ist, hängt davon ab, welche technischen Komponenten darüber vernetzt sind. Bei einem SUAV, wie es im Kapitel 2 beschrieben ist, hätte ein gleichzeitiges Versagen der Kommunikation auf beiden CAN-Bussen einen sofortigen Kontrollverlust des Fluggerätes zur Folge.

Node Services dienen zur verbindungsorientierten Kommunikation zwischen zwei Knoten. Die Verbindung wird durch genau eine Nachricht (Request) initiiert und wird je nach Dienst entweder durch eine Antwortnachricht (Response) des im Request addressierten Knotens aufgelöst oder ist, wenn keine Antwort verlangt wird, sofort beendet. Im Request wird die Node-ID des Empfängers und im Response die Node-ID des Absenders versendet, sodass in beiden Nachrichten die gleiche Node-ID angegeben ist. Dadurch ist die Verbindung innerhalb eines Service Channels eindeutig identifizierbar.

Ein Service Channel verfügt über zwei CAN-Identifier, einen für Requests und einen für Responses. Die Spezifikation empfiehlt, jedem Knoten, der Requests versendet, einen eigenen Service Channel dafür zu vergeben, vgl. [20], Abschnitt 4.

Für die Kommunikation aller Node Services stehen 52 Service Channel zur Verfügung. Besteht ein Bussystem aus mehr als 52 Knoten, welche alle Node Service Requests versenden, steht nicht mehr jedem Knoten ein eigener Service Channel zur Verfügung. CANas stellt dadurch nicht mehr die sofortige Kollisionsauflösung durch die CAN-Arbitrierung sicher, weil es nun vorkommen kann, dass verschiedene Knoten ihre Service Requests und Responses mit den gleichen Identifiern zur gleichen Zeit versenden.

Um solche Kollisionsquellen auszuschließen, muss der Anwender die Zahl der Requests versendenden Knoten eigenverantwortlich unter 53 halten oder sicherstellen, dass die Requests zu unterschiedlichen Zeiten erfolgen. Requests beinhalten laut Spezifikation immer eine Node-ID zur Adressierung. Dadurch werden Kollisionen bei Responses durch Broadcast-Requests vermieden.

Die verbindungslose Kommunikationsart Normal Operation Data (NOD) ist ebenfalls anfällig für unauflösbare Kollisionen, wenn die Knoten eines Bussystems so konfiguriert sind, dass sie die gleichen Identifier zum Versand von NOD nutzen.

In Abschnitt 3.3 wurde die Auswirkung von Übertragunsfehlern bereits beschrieben. Abgesehen vom Verlust der Echtzeitfähigkeit beim Informationsaustausch, stellen die Knoten ihre CAN-Kommunikation ggf. ganz ein.

Um dem entgegen zu wirken, wurden die automatisch unternommenen, erneuten Sendeversuche in den CAN-Controllern der BCU standardmäßig abgeschaltet. Die BCU-Firmware generiert eine von der Node-ID abhängige Wartezeit zwischen fehlgeschlagenen Sendeversuchen. Jede Node-ID kommt in einem Bussystem nur einmal vor, weshalb die Wartezeit eine Maßnahme zur Reduzierung der Kollisionen darstellt.

In einem Experiment mit NOD-Nachrichten lässt sich der Kommunikationsabbruch trotzdem leicht provozieren. Hierzu wurde das im Listing 5.1 gezeigte Python-Skript geschrieben, welches mit einer Frequenz von 100 Hz Heartbeat-Nachrichten auf dem CAN-Bus versendet. Zusätzlich versendet es zwei Nachrichten mit den Identifiern 20_{16} und 40_{16}. Nach einer Wartezeit von 1 Millisekunde werden die beiden Nachrichten wiederholt. Durch die Ausführungszeit und die Wartezeit ergibt sich eine Wiederholfrequenz der beiden Nachrichten von ca. 1001 Hz.

Dieses Skript wird in Python-Bytecode umgewandelt und auf drei BCUs in einem CAN-Bus parallel ausgeführt.

```
1  import canas as c
2  import canraw as r
3  import time
4
5  c.setComponentID(16 + r.getNodeID(0))
6  c.setHealthState(32 + r.getNodeID(0))
7  c.setHeartbeatInterval(0)
8
9  b1 = bytes([10, 11, 12, 13, 14, 15, 16, r.getNodeID(0)])
10 b2 = bytes([17, 18, 19, 20, 21, 22, 23, r.getNodeID(0)])
11
12 while True:
13         #r.tx0(b1, 0x20 + r.getNodeID(0), False, False)
14         #r.tx0(b2, 0x40 + r.getNodeID(0), False, False)
15         r.tx0(b1, 0x20, False, False)
16         r.tx0(b2, 0x40, False, False)
17         time.sleepMS(1)
```

Listing 5.1: Python-Skript für einen Versuch mit nicht auflösbaren Kollisionen, BCUs mit diesem Skript versenden Heartbeats und 2 Nachrichten in hoher Frequenz

Es werden also drei Identifier in hoher Frequenz von drei verschiedenen BCUs in einem Bussystem sendend genutzt. Dadurch entstehen im Verhältnis zu den erfolgreichen Sendeversuchen viele unlösbare Nachrichtenkollisionen.

Das Skript führt bei einer Wartezeit von 1 Millisekunde in Zeile 17 zu einer Gesamtbuslast von 37 % (3200 $\frac{Nachrichten}{Sekunde}$). Fünf Versuche wurden durchgeführt, jedes mal wurde nach unter 30 Sekunden mindestens ein Knoten durch zu häufige Kollisionen in den Bus-Off Zustand versetzt und stellte seine Buskommunikation ein.

Bei einer Wartezeit von 2 Millisekunden betrug die Versuchsdauer bei fünf Versuchen bis zu 2 Minuten, bis ein Knoten auf die gleiche Art ausfiel. Die Buslast lag bei 26 % (2270 $\frac{Nachrichten}{Sekunde}$).

Nutzt man anstatt der Zeilen 15 und 16 die beiden auskommentierten Zeilen oberhalb für den Versand der Nachrichten, läuft das Skript trotz des immer noch mehrfach verwendeten Identifiers für die Heartbeats, ohne dass ein Knoten die Kommunikation einstellt. Durch die Addition der Node-IDs auf die CAN-Identifier in den Zeilen 15 und 16 werden kollidierende Sendeversuche wieder durch die bitweise Arbitrierung lösbar.

Das Experiment verdeutlicht, wie wichtig es ist, CAN-Identifier nicht sendend in mehreren Knoten eines Bussystems zu nutzen. Lässt sich dies nicht vermeiden, muss es in ausreichender zeitlicher Distanz geschehen.

6. Fazit und Ausblick

Das im Rahmen dieser Arbeit entwickelte Konzept für die geforderte Bus-Konverter-Einheit BCU erfüllt alle Anforderungen der Aufgabenstellung. Zur Evaluierung des Konzepts wurde eine funktionsfähige BCU in Hard- und Software umgesetzt und die zur Nutzung notwendige Toolchain auf Basis von GNU/Linux erstellt.

Die BCU verfügt über die gewünschte Flexibilität für den generischen Einsatz und bietet darüber hinaus das Potenzial, Aufgaben automatisiert und teilautonom zu lösen. Über die MicroPython-Laufzeitumgebung lässt sich das Verhalten der BCU beliebig definieren. Auch die geforderte Rohdatenverarbeitung ist mit der Skriptsprache Python innerhalb der BCU umsetzbar. Die Performanz dieses Ansatzes ist im Nachgang dieser Arbeit in den Bereichen Durchsatzraten der Hardware-Schnittstellen, Rechenleistung innerhalb der Python-Laufzeitumgebung und Größe bzw. Komplexität der Python-Bytecodes noch genauer zu ermitteln. Beispielsweise fehlen Erfahrungswerte zum Heap-Speicherbedarf, der bei der Implementierung einer Rohdatenverarbeitung. Wie kritisch der Laufzeitbedarf des Garbage-Collectors bei einer Echtzeiterfassung von Daten ist und wie gut sich die Garbage-Collection durch Optimierungen in den Pythonskripten vermeiden lässt, ist ebenfalls noch nicht im Rahmen dieser Arbeit evaluiert worden. Sicherlich lassen sich Heap- und Stackgröße und deren Position im RAM durch Optimierung der Firmware und speziell des Linkerskripts erhöhen, um das Potenzial der BCU weiter zu steigern. Mit vermehrtem Heap-Speicher, können größere bzw. zahlreichere Python-Objekte allokiert werden, was umfangreichere Pythonskripte ermöglicht. Wie groß der Stack für einen sicheren Betrieb sein muss und wie viele Kilobytes man dem C++-Heap und dem MicroPython-Heap für eine optimale Nutzung zuweisen sollte, ist im Nachgang dieser Arbeit noch zu evaluieren. Die Firmware hat einen modularen Aufbau, der durch den Einsatz von loser Kopplung und dem zuvor beschriebenen HAL mit minimalem Aufwand erweitert werden kann. Die ursprüngliche MicroPython-Implementierung hatte in diesem Bereich, sowie bei der Dokumentation und Wartbarkeit des Sourcecodes große Defizite.

Mit MicroPython ist der Aktionsradius des Benutzers auch in ausreichender Weise einschränkt worden, um einen sicheren und zuverlässigen Betrieb der BCU zu garantie-

ren. Durch einen separaten Heap mit eigener Speicherverwaltung werden unerlaubte Schreibzugriffe auf sicherheitsrelevante Bereiche der Firmware vermieden. Auch die Verwendung der GPIO-Pins des Mikrocontrollers lässt sich nur innerhalb der durch die Firmware vorgegebenen Freiheitsgrade ändern, wodurch eine fixe Schnittstellendefinition garantiert wird. Die CAN-Kommunikation kann daher nur noch durch eine Flut von hoch priorisierten Nachrichten oder das Ändern der Bitrate gestört werden, dies ist aber ein akzeptabler Kompromiss zwischen Betriebssicherheit und Benutzerfreiheit.

Stacküberläufe werden aktuell noch nicht verhindert. Wie bereits beschrieben wurde, lassen sich diese in Zukunft jedoch auf einfache Weise absichern.

CANaerospace hat sich als geeignetes Übertragungsprotokoll zur Vernetzung der Komponenten der SAFRAN-Flugsystemarchitektur erwiesen. Die zahlreichen Freiheitsgrade dieses CAN-Protokolls erfordern ein hohes Maß an Sorgfalt bei der Busauslegung und dessen Konfiguration, um eine zuverlässige und echtzeitfähige Kommunikation zu garantieren. Jedoch bieten sie auch die Möglichkeit, das Protokoll für den Anwendungsfall zu optimieren. Dieses Potenzial wurde bei der Erarbeitung einer auf SUAVs spezialisierten CAN-Identifier Distribution umfangreich genutzt. Die Distribution regelt eine redundante Informationsverbreitung innerhalb einer dem SAFRAN Projekt entsprechenden Flugsystemarchitektur. Mit dem CANaerospace Protokoll werden die Nachrichten gemäß ihrer Informationspriorität und unabhängig vom Absender arbitriert.

Die Erkenntnisse aus dieser Arbeit zeigen, dass sich CANaerospace zur Verwendung im SUAV-Bereich eignet. Ob es sich dort etablieren wird, bleibt abzuwarten. Durch die BCU wird aufgezeigt, wie ein CANaerospace-Busknoten für SUAVs umgesetzt werden kann. Sie bildet eine solide Basis für die Realisierung einer redundanten Flugsystemarchitektur.

Literaturverzeichnis

[1] ISO/IEC JTC 1. ISO-7498-1 – Open Systems Interconnection – Basic Reference Model, 1994.

[2] Robert Bosch GmbH. CAN Specification, Version 2.0, 1991.

[3] Robert Bosch GmbH. Datasheet STM32F415xx STM32F17xx, DocID022063 Rev 8, http://www.st.com/content/ccc/resource/technical/document/datasheet/98/9f/89/73/01/b1/48/98/DM00035129.pdf/files/DM00035129.pdf/jcr:content/translations/en.DM00035129.pdf, 2016, Aufruf 07.07.2017.

[4] Steve Corrigan. Introduction to the Controller Area Network (CAN), Application Report SLOA101B, Texas Instruments, August 2002.

[5] Thomas Führer, Bernd Müller, Werner Dieterle, Florian Hartwich, Robert Hugel, Michael Walther, and Robert Bosch GmbH. Time Triggered Communication on CAN.

[6] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[7] Damien P. George, Paul Sokolovsky, and contributors. FAQ - Wiki, Github, https://github.com/micropython/micropython/wiki/FAQ, Aufruf 14.12.2017 2017.

[8] Damien P. George, Paul Sokolovsky, and contributors. Quick reference for the pyboard, http://docs.micropython.org/en/latest/pyboard/pyboard/quickref.html, Aufruf 26.06.2017 2017, Version 1.9.1.

[9] Hagen Hasberg. Ein Testkonzept für Flugregler, Bachelorthesis. Juni 2014.

[10] Christian Hornemann and Ingo Goldstein. Design und Aufbau eines Flugroboters mit einer dezentralen und redundanten Systemarchitektur, Masterprojekt, 2017.

[11] Elizabeth Howell and NASA/Jim Ross. BUDGET 2015: Flying SOFIA Telescope To Be Shelved For ‚Higher-Priority' Programs Like Cassini, https://www.universetoday.com/110007/budget-2015-flying-sofia-telescope-to-be-shelved-for-higher-priority-programs-like-cassini/, 2010, Aufruf 05.01.2018.

[12] CAN in Automation (CiA). CANopen application layer and communication profile, Version 4.2.0, 2011.

[13] ISO. INTERNATIONAL STANDARD ISO 11783, Tractors and machinery for agriculture and forestry, 2002.

[14] Martin Fowler. Inversion of Control Containers and the Dependency Injection pattern, https://www.martinfowler.com/articles/injection.html, 2004, Aufruf 12.01.2018.

[15] Daniele Mazzei, Giacomo Baldi, and Gabriele Montelisciani. A Full Stack for Quick Prototyping of IoT Solutions, DOI: 10.1109/CIOT.2016.7872915, 2016.

[16] Amardeep Mehta, Rami Baddour, Fredrik Svensson, Harald Gustafsson, and Erik Elmroth. Calvin Constrained - A Framework for IoT Applications in Heterogeneous Environments, DOI: 10.1109/ICDCS.2017.181. 2017.

[17] Till Steinbach, Hermand Dieumo Kenfack, Franz Korf, and Thomas C. Schmidt. An Extension of the OMNeT++ INET Framework for Simulating Real-time Ethernet with High Accuracy. In *SIMUTools 2011 – 4th International OMNeT++ Workshop*, pages 375–382, New York, USA, March 21-25 2011. ACM DL.

[18] Michael Stock. Fachgespräch mit Michael Stock bei Stock Flight Systems, Schützenweg 8a, 82335 Berg/Farchach, durchgeführt von Ulrich ter Horst, 18. Mai 2017.

[19] Stock Flight Systems. CANaerospace Presentation – CANaerospace/AGATE data bus, http://www.stockflightsystems.com/tl_files/downloads/canaerospace/CANaerospace_Presentation.pdf, 2005, Aufruf 05.01.2018.

[20] Stock Flight Systems. CANaerospace, Revision 1.7, 2006.

[21] Stock Flight Systems. CANaerospaceNEWS, 01/04, http://www.stockflightsystems.com/tl_files/downloads/canaerospace/CANaerospace_News_0104.pdf, 2016, Aufruf 13.12.2017.

[22] Ulrich ter Horst. Foto - Adapterplatine für MiniM4, Juni 2017.

[23] Ulrich ter Horst. Foto - Adapterplatine mit MiniM4, Juli 2017.

[24] Ulrich ter Horst. Foto - Mikrocontroller-Platine MiniM4, Ober- u. Unterseite, Hersteller: Mikro Elektronika, Dezember 2017.

[25] Ulrich ter Horst. Systemarchitektur, veränderte Grafik aus interner Kommunikation, August 2017.

A. REPL-GUI

Die Abbildung A.1 zeigt die Oberfläche einer Anwendung für den Remote-Zugriff auf die Read-Eval-Print Loop (REPL) via CAN-Bus.

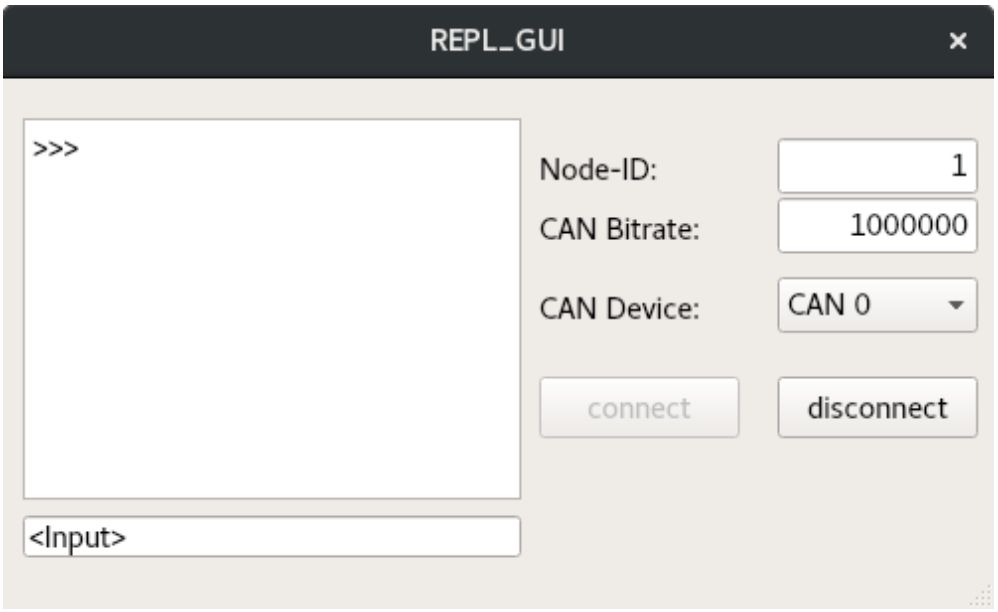


Abbildung A.1.: Entwurf einer möglichen REPL-GUI

B. Identifier Distributionen - NOD

Die Tabelle B.1 zeigt den ersten Entwurf einer CAN-Identifier Zuordnung für UAVs. Hierbei wird hauptsächlich der benutzerdefinierte Bereich genutzt, um vollständig mit der Standardzuordnung von CANaerospace kompatibel zu sein.

ID_{16}	ID_{10}	Information/Dienst	Einheit	Quelle	Priorität	Datentyp
12C	300	Acceleration longitudinal	g	IMU	0 critical	FLOAT
12D	301	Acceleratoin lateral	g	IMU	0 critical	FLOAT
12E	302	Acceleration normal	g	IMU	0 critical	FLOAT
12F	303	Body Pitch Rate (Q)	Degree/s	IMU	0 critical	FLOAT
130	304	Body Roll Rate (P)	Degree/s	IMU	0 critical	FLOAT
131	305	Body Yaw Rate (R)	Degree/s	IMU	0 critical	FLOAT
137	311	Body Pitch Angle	Degree	IMU	0 critical	FLOAT
138	312	Body Roll Angle	Degree	IMU	0 critical	FLOAT
140	320	Baro corrected Altitude (AGL)	m	BSU	2 normal	FLOAT
141	321	Heading Angle rel. to North	Degree	IMU	0 critical	FLOAT
14F	335	Baro Temperature	K/10	BSU	3 low	FLOAT
153	339	Total Pressure	hPa	BSU	2 normal	FLOAT
190	400	RC-Channel 0 Roll	interval $[-1;+1]$	RCU	1 high	FLOAT
191	401	RC-Channel 1 Pitch	interval $[-1;+1]$	RCU	1 high	FLOAT
193	403	RC-Channel 2 Throttle	interval $[-1;+1]$	RCU	1 high	FLOAT
194	404	RC-Channel 3 Yaw	interval $[-1;+1]$	RCU	1 high	FLOAT
398	920	Battery Voltage	Volt	PMU	2 normal	FLOAT
40C	1036	Latitude	Degree	GSU	2 normal	FLOAT
40D	1037	Longitude	Degree	GSU	2 normal	FLOAT
40E	1038	GPS Altitude (MSL/WGS84)	m	GSU	3 low	FLOAT
40F	1039	GPS Ground Speed	m/s	GSU	3 low	FLOAT
5DC	1500	Ultrasonic Height	m	USU	1 high	FLOAT
5DD	1501	Distance to Obstacle horiz.	m	USU	1 high	FLOAT
5DE	1502	Distance to Obstacle above	m	USU	1 high	FLOAT
5E6	1510	Heartbeat – SHU		SHU	1 high	UCHAR4
5F0	1520	SensorFusion Roll	Degree	SHU	0 critical	FLOAT
5F1	1521	SensorFusion Pitch	Degree	SHU	0 critical	FLOAT
5F2	1522	SensorFusion Yaw	Degree	SHU	0 critical	FLOAT
5F3	1523	SensorFusion Latitude	Degree	SHU	1 high	FLOAT
5F4	1524	SensorFusion Longitude	Degree	SHU	1 high	FLOAT

ID_{16}	ID_{10}	Information/Dienst	Einheit	Quelle	Priorität	Datentyp
5F6	1526	SensorFusion Height (AGL)	m	SHU	1 high	FLOAT
5F7	1527	SensorFusion Altitude (MSL)	m	SHU	1 high	FLOAT
5F8	1528	SensorFusion Airspeed	m/s	SHU	1 high	FLOAT
5F9	1529	SensorFusion Climbrate	m/s	SHU	1 high	FLOAT
604	1540	Desired Roll	interval $[-1;+1]$	FCU	1 high	FLOAT
605	1541	Desired Pitch	interval $[-1;+1]$	FCU	1 high	FLOAT
606	1542	Desired Yaw	interval $[-1;+1]$	FCU	1 high	FLOAT
607	1543	Desired Throttle	interval $[-1;+1]$	FCU	1 high	FLOAT
608	1544	CAU type1 – Motor A1 & A2	2*int. $[0;2^{15}-1]$	CAU	0 critical	SHORT2
609	1545	CAU type1 – Motor B1 & B2	2*int. $[0;2^{15}-1]$	CAU	0 critical	SHORT2
60A	1546	CAU type1 – Motor C1 & C2	2*int. $[0;2^{15}-1]$	CAU	0 critical	SHORT2
60B	1547	CAU type1 – Motor D1 & D2	2*int. $[0;2^{15}-1]$	CAU	0 critical	SHORT2
60C	1548	CAU type2 – Motor A1	int. $[0;2^{31}-1]$	CAU	0 critical	ULONG
60D	1549	CAU type2 – Motor A2	int. $[0;2^{31}-1]$	CAU	0 critical	ULONG
60E	1550	CAU type2 – Motor B1	int. $[0;2^{31}-1]$	CAU	0 critical	ULONG
60F	1551	CAU type2 – Motor B2	int. $[0;2^{31}-1]$	CAU	0 critical	ULONG
610	1552	CAU type2 – Motor C1	int. $[0;2^{31}-1]$	CAU	0 critical	ULONG
611	1553	CAU type2 – Motor C2	int. $[0;2^{31}-1]$	CAU	0 critical	ULONG
612	1554	CAU type1 – Motor D1	int. $[0;2^{31}-1]$	CAU	0 critical	ULONG
613	1555	CAU type1 – Motor D2	int. $[0;2^{31}-1]$	CAU	0 critical	ULONG
615	1557	Magnetic Field Strength N	nT	FSM	2 normal	FLOAT
616	1558	Magnetic Field Strength E	nT	FSM	2 normal	FLOAT
617	1559	Magnetic Field Strength D	nT	FSM	2 normal	FLOAT
618	1560	RPM & Temperature Motor A1	1/min & K/10	MCU	2 normal	SHORT2
619	1561	RPM & Temperature Motor A2	1/min & K/10	MCU	2 normal	SHORT2
61A	1562	RPM & Temperature Motor B1	1/min & K/10	MCU	2 normal	SHORT2
61B	1563	RPM & Temperature Motor B2	1/min & K/10	MCU	2 normal	SHORT2
61C	1564	RPM & Temperature Motor C1	1/min & K/10	MCU	2 normal	SHORT2
61D	1565	RPM & Temperature Motor C2	1/min & K/10	MCU	2 normal	SHORT2
61E	1566	RPM & Temperature Motor D1	1/min & K/10	MCU	2 normal	SHORT2
61F	1567	RPM & Temperature Motor D2	1/min & K/10	MCU	2 normal	SHORT2
622	1570	Heartbeat		***	3 low	UCHAR4
623	1571	Battery Cell State A & B	mV	PMU	3 low	SHORT2
624	1572	Battery Cell State C & D	mV	PMU	3 low	SHORT2
625	1573	Battery Cell State E & F	mV	PMU	3 low	SHORT2
626	1574	Battery Cell State G & H	mV	PMU	3 low	SHORT2
627	1575	GPS Status		GSU	3 low	ULONG
628	1576	GPS Speed N	m/s	GSU	3 low	FLOAT
629	1577	GPS Speed E	m/s	GSU	3 low	FLOAT
62A	1578	GPS Speed D	m/s	GSU	3 low	FLOAT
62B	1579	GPS Track	Degree	GSU	3 low	FLOAT

ID_{16}	ID_{10}	Information/Dienst	Einheit	Quelle	Priorität	Datentyp
640	1600	GPS Time (UTC + X seconds)	sec	GSU	3 low	ULONG

Tabelle B.1.: Std. ID Distribution mit UAV-Nachrichten im benutzerdefinierten Bereich - NOD

Die nachfolgende Tabelle B.2 zeigt den zweiten Entwurf einer CAN-Identifier Zuordnung für UAVs. Auf die Kompatibilität mit der CANaerospace Standardzuordnung ist hier verzichtet worden, um eine für UAVs ausreichende Redundanz der zu übertragenden Daten zu erzielen, ohne die Priorisierung der Informationen durcheinander zu bringen.

ID_{16}	ID_{10}	Information/Dienst	Einheit	Quelle	Priorität	Datentyp
136	310	Acceleration longitudinal	g	IMU	0 critical	FLOAT
137	311	Acceleration longitudinal	g	IMU	0 critical	FLOAT
138	312	Acceleration longitudinal	g	IMU	0 critical	FLOAT
139	313	Acceleration longitudinal	g	IMU	0 critical	FLOAT
13A	314	Acceleration longitudinal	g	IMU	0 critical	FLOAT
13B	315	Acceleration longitudinal	g	IMU	0 critical	FLOAT
13C	316	Acceleration longitudinal	g	IMU	0 critical	FLOAT
13D	317	Acceleration longitudinal	g	IMU	0 critical	FLOAT
13F	319	Baro Correction	hPa	FCU	3 low	FLOAT
140	320	Acceleratoin lateral	g	IMU	0 critical	FLOAT
141	321	Acceleratoin lateral	g	IMU	0 critical	FLOAT
142	322	Acceleratoin lateral	g	IMU	0 critical	FLOAT
143	323	Acceleratoin lateral	g	IMU	0 critical	FLOAT
144	324	Acceleratoin lateral	g	IMU	0 critical	FLOAT
145	325	Acceleratoin lateral	g	IMU	0 critical	FLOAT
146	326	Acceleratoin lateral	g	IMU	0 critical	FLOAT
147	327	Acceleratoin lateral	g	IMU	0 critical	FLOAT
14A	330	Acceleration normal	g	IMU	0 critical	FLOAT
14B	331	Acceleration normal	g	IMU	0 critical	FLOAT
14C	332	Acceleration normal	g	IMU	0 critical	FLOAT
14D	333	Acceleration normal	g	IMU	0 critical	FLOAT
14E	334	Acceleration normal	g	IMU	0 critical	FLOAT
14F	335	Acceleration normal	g	IMU	0 critical	FLOAT
150	336	Acceleration normal	g	IMU	0 critical	FLOAT
151	337	Acceleration normal	g	IMU	0 critical	FLOAT
154	340	Body Roll Angle	Degree	IMU	0 critical	FLOAT
155	341	Body Roll Angle	Degree	IMU	0 critical	FLOAT
156	342	Body Roll Angle	Degree	IMU	0 critical	FLOAT
157	343	Body Roll Angle	Degree	IMU	0 critical	FLOAT
158	344	Body Roll Angle	Degree	IMU	0 critical	FLOAT
159	345	Body Roll Angle	Degree	IMU	0 critical	FLOAT

ID_{16}	ID_{10}	Information/Dienst	Einheit	Quelle	Priorität	Datentyp
15A	346	Body Roll Angle	Degree	IMU	0 critical	FLOAT
15B	347	Body Roll Angle	Degree	IMU	0 critical	FLOAT
15E	350	Body Pitch Angle	Degree	IMU	0 critical	FLOAT
15F	351	Body Pitch Angle	Degree	IMU	0 critical	FLOAT
160	352	Body Pitch Angle	Degree	IMU	0 critical	FLOAT
161	353	Body Pitch Angle	Degree	IMU	0 critical	FLOAT
162	354	Body Pitch Angle	Degree	IMU	0 critical	FLOAT
163	355	Body Pitch Angle	Degree	IMU	0 critical	FLOAT
164	356	Body Pitch Angle	Degree	IMU	0 critical	FLOAT
165	357	Body Pitch Angle	Degree	IMU	0 critical	FLOAT
168	360	Heading Angle rel. to North	Degree	IMU	0 critical	FLOAT
169	361	Heading Angle rel. to North	Degree	IMU	0 critical	FLOAT
16A	362	Heading Angle rel. to North	Degree	IMU	0 critical	FLOAT
16B	363	Heading Angle rel. to North	Degree	IMU	0 critical	FLOAT
16C	364	Heading Angle rel. to North	Degree	IMU	0 critical	FLOAT
16D	365	Heading Angle rel. to North	Degree	IMU	0 critical	FLOAT
16E	366	Heading Angle rel. to North	Degree	IMU	0 critical	FLOAT
16F	367	Heading Angle rel. to North	Degree	IMU	0 critical	FLOAT
172	370	Body Roll Rate (P)	Degree/sec	IMU	0 critical	FLOAT
173	371	Body Roll Rate (P)	Degree/sec	IMU	0 critical	FLOAT
174	372	Body Roll Rate (P)	Degree/sec	IMU	0 critical	FLOAT
175	373	Body Roll Rate (P)	Degree/sec	IMU	0 critical	FLOAT
176	374	Body Roll Rate (P)	Degree/sec	IMU	0 critical	FLOAT
177	375	Body Roll Rate (P)	Degree/sec	IMU	0 critical	FLOAT
178	376	Body Roll Rate (P)	Degree/sec	IMU	0 critical	FLOAT
179	377	Body Roll Rate (P)	Degree/sec	IMU	0 critical	FLOAT
17C	380	Body Pitch Rate (Q)	Degree/sec	IMU	0 critical	FLOAT
17D	381	Body Pitch Rate (Q)	Degree/sec	IMU	0 critical	FLOAT
17E	382	Body Pitch Rate (Q)	Degree/sec	IMU	0 critical	FLOAT
17F	383	Body Pitch Rate (Q)	Degree/sec	IMU	0 critical	FLOAT
180	384	Body Pitch Rate (Q)	Degree/sec	IMU	0 critical	FLOAT
181	385	Body Pitch Rate (Q)	Degree/sec	IMU	0 critical	FLOAT
182	386	Body Pitch Rate (Q)	Degree/sec	IMU	0 critical	FLOAT
183	387	Body Pitch Rate (Q)	Degree/sec	IMU	0 critical	FLOAT
186	390	Body Yaw Rate (R)	Degree/sec	IMU	0 critical	FLOAT
187	391	Body Yaw Rate (R)	Degree/sec	IMU	0 critical	FLOAT
188	392	Body Yaw Rate (R)	Degree/sec	IMU	0 critical	FLOAT
189	393	Body Yaw Rate (R)	Degree/sec	IMU	0 critical	FLOAT
18A	394	Body Yaw Rate (R)	Degree/sec	IMU	0 critical	FLOAT
18B	395	Body Yaw Rate (R)	Degree/sec	IMU	0 critical	FLOAT
18C	396	Body Yaw Rate (R)	Degree/sec	IMU	0 critical	FLOAT
18D	397	Body Yaw Rate (R)	Degree/sec	IMU	0 critical	FLOAT

ID_{16}	ID_{10}	Information/Dienst	Einheit	Quelle	Priorität	Datentyp
190	400	SensorFusion Roll	Degree	SHU	0 critical	FLOAT
191	401	SensorFusion Roll	Degree	SHU	0 critical	FLOAT
192	402	SensorFusion Roll	Degree	SHU	0 critical	FLOAT
193	403	SensorFusion Roll	Degree	SHU	0 critical	FLOAT
194	404	SensorFusion Roll	Degree	SHU	0 critical	FLOAT
195	405	SensorFusion Roll	Degree	SHU	0 critical	FLOAT
196	406	SensorFusion Roll	Degree	SHU	0 critical	FLOAT
197	407	SensorFusion Roll	Degree	SHU	0 critical	FLOAT
19A	410	SensorFusion Pitch	Degree	SHU	0 critical	FLOAT
19B	411	SensorFusion Pitch	Degree	SHU	0 critical	FLOAT
19C	412	SensorFusion Pitch	Degree	SHU	0 critical	FLOAT
19D	413	SensorFusion Pitch	Degree	SHU	0 critical	FLOAT
19E	414	SensorFusion Pitch	Degree	SHU	0 critical	FLOAT
19F	415	SensorFusion Pitch	Degree	SHU	0 critical	FLOAT
1A0	416	SensorFusion Pitch	Degree	SHU	0 critical	FLOAT
1A1	417	SensorFusion Pitch	Degree	SHU	0 critical	FLOAT
1A4	420	SensorFusion Yaw	Degree	SHU	0 critical	FLOAT
1A5	421	SensorFusion Yaw	Degree	SHU	0 critical	FLOAT
1A6	422	SensorFusion Yaw	Degree	SHU	0 critical	FLOAT
1A7	423	SensorFusion Yaw	Degree	SHU	0 critical	FLOAT
1A8	424	SensorFusion Yaw	Degree	SHU	0 critical	FLOAT
1A9	425	SensorFusion Yaw	Degree	SHU	0 critical	FLOAT
1AA	426	SensorFusion Yaw	Degree	SHU	0 critical	FLOAT
1AB	427	SensorFusion Yaw	Degree	SHU	0 critical	FLOAT
1AE	430	CAU type1 – Motor A1 & A2	2*int. $[0; 2^{15} - 1]$	CAU	0 critical	SHORT2
1AF	431	CAU type1 – Motor B1 & B2	2*int. $[0; 2^{15} - 1]$	CAU	0 critical	SHORT2
1B0	432	CAU type1 – Motor C1 & C2	2*int. $[0; 2^{15} - 1]$	CAU	0 critical	SHORT2
1B1	433	CAU type1 – Motor D1 & D2	2*int. $[0; 2^{15} - 1]$	CAU	0 critical	SHORT2
1B2	434	CAU type2 – Motor A1	int.$[0; 2^{31} - 1]$	CAU	0 critical	ULONG
1B3	435	CAU type2 – Motor A2	int.$[0; 2^{31} - 1]$	CAU	0 critical	ULONG
1B4	436	CAU type2 – Motor B1	int.$[0; 2^{31} - 1]$	CAU	0 critical	ULONG
1B5	437	CAU type2 – Motor B2	int.$[0; 2^{31} - 1]$	CAU	0 critical	ULONG
1B6	438	CAU type2 – Motor C1	int.$[0; 2^{31} - 1]$	CAU	0 critical	ULONG
1B7	439	CAU type2 – Motor C2	int.$[0; 2^{31} - 1]$	CAU	0 critical	ULONG
1B8	440	CAU type1 – Motor D1	int.$[0; 2^{31} - 1]$	CAU	0 critical	ULONG
1B9	441	CAU type1 – Motor D2	int.$[0; 2^{31} - 1]$	CAU	0 critical	ULONG
1F4	500	RC-Channel 0 Roll	int. $[-1; +1]$	RCU	1 high	FLOAT
1F5	501	RC-Channel 0 Roll	int. $[-1; +1]$	RCU	1 high	FLOAT
1F7	503	RC-Channel 1 Pitch	int. $[-1; +1]$	RCU	1 high	FLOAT
1F8	504	RC-Channel 1 Pitch	int. $[-1; +1]$	RCU	1 high	FLOAT
1FA	506	RC-Channel 2 Throttle	int. $[-1; +1]$	RCU	1 high	FLOAT
1FB	507	RC-Channel 2 Throttle	int. $[-1; +1]$	RCU	1 high	FLOAT

ID_{16}	ID_{10}	Information/Dienst	Einheit	Quelle	Priorität	Datentyp
1FD	509	RC-Channel 3 Yaw	int. $[-1;+1]$	RCU	1 high	FLOAT
1FE	510	RC-Channel 3 Yaw	int. $[-1;+1]$	RCU	1 high	FLOAT
208	520	Ultrasonic Height	m	USU	1 high	FLOAT
209	521	Ultrasonic Height	m	USU	1 high	FLOAT
20A	522	Ultrasonic Height	m	USU	1 high	FLOAT
20B	523	Ultrasonic Height	m	USU	1 high	FLOAT
20C	524	Ultrasonic Height	m	USU	1 high	FLOAT
20D	525	Ultrasonic Height	m	USU	1 high	FLOAT
20E	526	Ultrasonic Height	m	USU	1 high	FLOAT
20F	527	Ultrasonic Height	m	USU	1 high	FLOAT
212	530	Distance to Obstacle horizontal	m	USU	1 high	FLOAT
213	531	Distance to Obstacle horizontal	m	USU	1 high	FLOAT
214	532	Distance to Obstacle horizontal	m	USU	1 high	FLOAT
215	533	Distance to Obstacle horizontal	m	USU	1 high	FLOAT
216	534	Distance to Obstacle horizontal	m	USU	1 high	FLOAT
217	535	Distance to Obstacle horizontal	m	USU	1 high	FLOAT
218	536	Distance to Obstacle horizontal	m	USU	1 high	FLOAT
219	537	Distance to Obstacle horizontal	m	USU	1 high	FLOAT
21C	540	Distance to Obstacle above	m	USU	1 high	FLOAT
21D	541	Distance to Obstacle above	m	USU	1 high	FLOAT
21E	542	Distance to Obstacle above	m	USU	1 high	FLOAT
21F	543	Distance to Obstacle above	m	USU	1 high	FLOAT
220	544	Distance to Obstacle above	m	USU	1 high	FLOAT
221	545	Distance to Obstacle above	m	USU	1 high	FLOAT
222	546	Distance to Obstacle above	m	USU	1 high	FLOAT
223	547	Distance to Obstacle above	m	USU	1 high	FLOAT
225	549	Heartbeat – SHU		SHU	1 high	UCHAR4
226	550	SensorFusion Latitude	Degree	SHU	1 high	FLOAT
227	551	SensorFusion Latitude	Degree	SHU	1 high	FLOAT
228	552	SensorFusion Latitude	Degree	SHU	1 high	FLOAT
229	553	SensorFusion Latitude	Degree	SHU	1 high	FLOAT
22A	554	SensorFusion Latitude	Degree	SHU	1 high	FLOAT
22B	555	SensorFusion Latitude	Degree	SHU	1 high	FLOAT
22C	556	SensorFusion Latitude	Degree	SHU	1 high	FLOAT
22D	557	SensorFusion Latitude	Degree	SHU	1 high	FLOAT
230	560	SensorFusion Longitude	Degree	SHU	1 high	FLOAT
231	561	SensorFusion Longitude	Degree	SHU	1 high	FLOAT
232	562	SensorFusion Longitude	Degree	SHU	1 high	FLOAT
233	563	SensorFusion Longitude	Degree	SHU	1 high	FLOAT
234	564	SensorFusion Longitude	Degree	SHU	1 high	FLOAT
235	565	SensorFusion Longitude	Degree	SHU	1 high	FLOAT
236	566	SensorFusion Longitude	Degree	SHU	1 high	FLOAT

ID_{16}	ID_{10}	Information/Dienst	Einheit	Quelle	Priorität	Datentyp
237	567	SensorFusion Longitude	Degree	SHU	1 high	FLOAT
23A	570	SensorFusion Height (AGL)	m	SHU	1 high	FLOAT
23B	571	SensorFusion Height (AGL)	m	SHU	1 high	FLOAT
23C	572	SensorFusion Height (AGL)	m	SHU	1 high	FLOAT
23D	573	SensorFusion Height (AGL)	m	SHU	1 high	FLOAT
23E	574	SensorFusion Height (AGL)	m	SHU	1 high	FLOAT
23F	575	SensorFusion Height (AGL)	m	SHU	1 high	FLOAT
240	576	SensorFusion Height (AGL)	m	SHU	1 high	FLOAT
241	577	SensorFusion Height (AGL)	m	SHU	1 high	FLOAT
244	580	SensorFusion Altitude (MSL)	m	SHU	1 high	FLOAT
245	581	SensorFusion Altitude (MSL)	m	SHU	1 high	FLOAT
246	582	SensorFusion Altitude (MSL)	m	SHU	1 high	FLOAT
247	583	SensorFusion Altitude (MSL)	m	SHU	1 high	FLOAT
248	584	SensorFusion Altitude (MSL)	m	SHU	1 high	FLOAT
249	585	SensorFusion Altitude (MSL)	m	SHU	1 high	FLOAT
24A	586	SensorFusion Altitude (MSL)	m	SHU	1 high	FLOAT
24B	587	SensorFusion Altitude (MSL)	m	SHU	1 high	FLOAT
24E	590	SensorFusion Airspeed	m/sec	SHU	1 high	FLOAT
24F	591	SensorFusion Airspeed	m/sec	SHU	1 high	FLOAT
250	592	SensorFusion Airspeed	m/sec	SHU	1 high	FLOAT
251	593	SensorFusion Airspeed	m/sec	SHU	1 high	FLOAT
252	594	SensorFusion Airspeed	m/sec	SHU	1 high	FLOAT
253	595	SensorFusion Airspeed	m/sec	SHU	1 high	FLOAT
254	596	SensorFusion Airspeed	m/sec	SHU	1 high	FLOAT
255	597	SensorFusion Airspeed	m/sec	SHU	1 high	FLOAT
258	600	SensorFusion Climbrate	m/sec	SHU	1 high	FLOAT
259	601	SensorFusion Climbrate	m/sec	SHU	1 high	FLOAT
25A	602	SensorFusion Climbrate	m/sec	SHU	1 high	FLOAT
25B	603	SensorFusion Climbrate	m/sec	SHU	1 high	FLOAT
25C	604	SensorFusion Climbrate	m/sec	SHU	1 high	FLOAT
25D	605	SensorFusion Climbrate	m/sec	SHU	1 high	FLOAT
25E	606	SensorFusion Climbrate	m/sec	SHU	1 high	FLOAT
25F	607	SensorFusion Climbrate	m/sec	SHU	1 high	FLOAT
262	610	Desired Roll	int. $[-1; +1]$	FCU	1 high	FLOAT
263	611	Desired Roll	int. $[-1; +1]$	FCU	1 high	FLOAT
264	612	Desired Roll	int. $[-1; +1]$	FCU	1 high	FLOAT
265	613	Desired Roll	int. $[-1; +1]$	FCU	1 high	FLOAT
26C	620	Desired Pitch	int. $[-1; +1]$	FCU	1 high	FLOAT
26D	621	Desired Pitch	int. $[-1; +1]$	FCU	1 high	FLOAT
26E	622	Desired Pitch	int. $[-1; +1]$	FCU	1 high	FLOAT
26F	623	Desired Pitch	int. $[-1; +1]$	FCU	1 high	FLOAT
276	630	Desired Yaw	int. $[-1; +1]$	FCU	1 high	FLOAT

ID_{16}	ID_{10}	Information/Dienst	Einheit	Quelle	Priorität	Datentyp
277	631	Desired Yaw	int. $[-1;+1]$	FCU	1 high	FLOAT
278	632	Desired Yaw	int. $[-1;+1]$	FCU	1 high	FLOAT
279	633	Desired Yaw	int. $[-1;+1]$	FCU	1 high	FLOAT
280	640	Desired Throttle	int. $[-1;+1]$	FCU	1 high	FLOAT
281	641	Desired Throttle	int. $[-1;+1]$	FCU	1 high	FLOAT
282	642	Desired Throttle	int. $[-1;+1]$	FCU	1 high	FLOAT
283	643	Desired Throttle	int. $[-1;+1]$	FCU	1 high	FLOAT
2BC	700	Baro corrected Altitude (AGL)	m	BSU	2 normal	FLOAT
2BD	701	Baro corrected Altitude (AGL)	m	BSU	2 normal	FLOAT
2BE	702	Baro corrected Altitude (AGL)	m	BSU	2 normal	FLOAT
2BF	703	Baro corrected Altitude (AGL)	m	BSU	2 normal	FLOAT
2C0	704	Baro corrected Altitude (AGL)	m	BSU	2 normal	FLOAT
2C1	705	Baro corrected Altitude (AGL)	m	BSU	2 normal	FLOAT
2C2	706	Baro corrected Altitude (AGL)	m	BSU	2 normal	FLOAT
2C3	707	Baro corrected Altitude (AGL)	m	BSU	2 normal	FLOAT
2C6	710	Total Pressure	hPa	BSU	2 normal	FLOAT
2C7	711	Total Pressure	hPa	BSU	2 normal	FLOAT
2C8	712	Total Pressure	hPa	BSU	2 normal	FLOAT
2C9	713	Total Pressure	hPa	BSU	2 normal	FLOAT
2CA	714	Total Pressure	hPa	BSU	2 normal	FLOAT
2CB	715	Total Pressure	hPa	BSU	2 normal	FLOAT
2CC	716	Total Pressure	hPa	BSU	2 normal	FLOAT
2CD	717	Total Pressure	hPa	BSU	2 normal	FLOAT
2D0	720	Battery Voltage	Volt	PMU	2 normal	FLOAT
2D1	721	Battery Voltage	Volt	PMU	2 normal	FLOAT
2D2	722	Battery Voltage	Volt	PMU	2 normal	FLOAT
2D3	723	Battery Voltage	Volt	PMU	2 normal	FLOAT
2D4	724	Battery Voltage	Volt	PMU	2 normal	FLOAT
2D5	725	Battery Voltage	Volt	PMU	2 normal	FLOAT
2D6	726	Battery Voltage	Volt	PMU	2 normal	FLOAT
2D7	727	Battery Voltage	Volt	PMU	2 normal	FLOAT
2DA	730	Latitude	Degree	GSU	2 normal	FLOAT
2DB	731	Latitude	Degree	GSU	2 normal	FLOAT
2DC	732	Latitude	Degree	GSU	2 normal	FLOAT
2DD	733	Latitude	Degree	GSU	2 normal	FLOAT
2DE	734	Latitude	Degree	GSU	2 normal	FLOAT
2DF	735	Latitude	Degree	GSU	2 normal	FLOAT
2E0	736	Latitude	Degree	GSU	2 normal	FLOAT
2E1	737	Latitude	Degree	GSU	2 normal	FLOAT
2E4	740	Longitude	Degree	GSU	2 normal	FLOAT
2E5	741	Longitude	Degree	GSU	2 normal	FLOAT
2E6	742	Longitude	Degree	GSU	2 normal	FLOAT

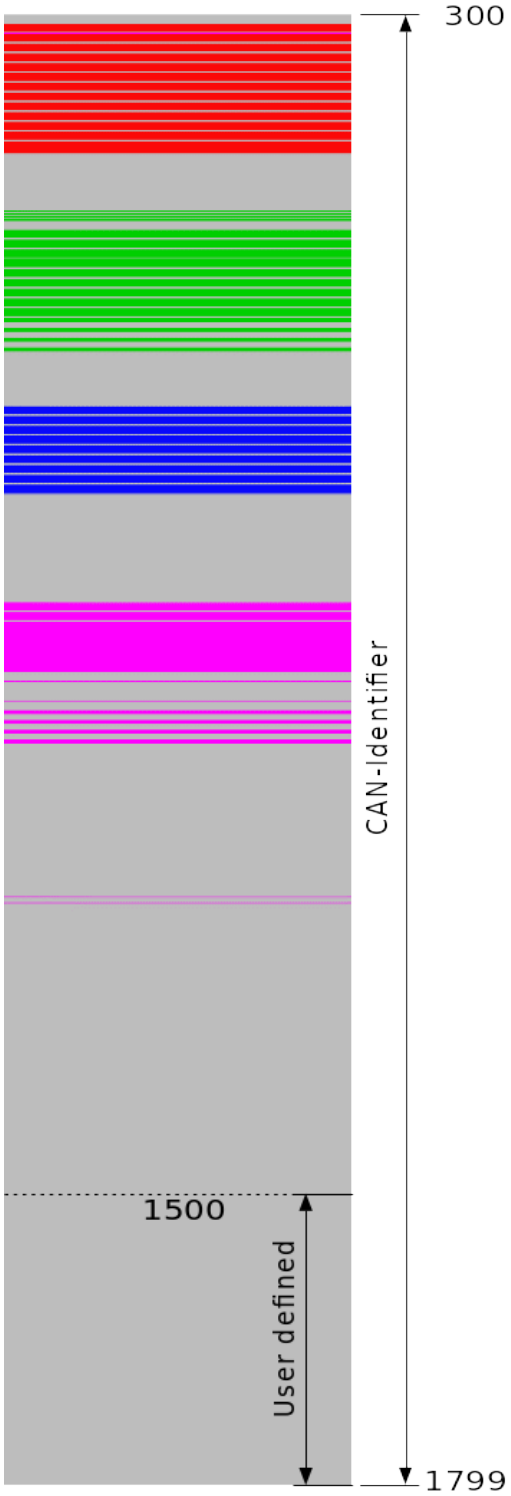
ID_{16}	ID_{10}	Information/Dienst	Einheit	Quelle	Priorität	Datentyp
2E7	743	Longitude	Degree	GSU	2 normal	FLOAT
2E8	744	Longitude	Degree	GSU	2 normal	FLOAT
2E9	745	Longitude	Degree	GSU	2 normal	FLOAT
2EA	746	Longitude	Degree	GSU	2 normal	FLOAT
2EB	747	Longitude	Degree	GSU	2 normal	FLOAT
2EE	750	Magnetic Field Strength N	nT	FSM	2 normal	FLOAT
2EF	751	Magnetic Field Strength N	nT	FSM	2 normal	FLOAT
2F0	752	Magnetic Field Strength N	nT	FSM	2 normal	FLOAT
2F1	753	Magnetic Field Strength N	nT	FSM	2 normal	FLOAT
2F2	754	Magnetic Field Strength N	nT	FSM	2 normal	FLOAT
2F3	755	Magnetic Field Strength N	nT	FSM	2 normal	FLOAT
2F4	756	Magnetic Field Strength N	nT	FSM	2 normal	FLOAT
2F5	757	Magnetic Field Strength N	nT	FSM	2 normal	FLOAT
2F8	760	Magnetic Field Strength E	nT	FSM	2 normal	FLOAT
2F9	761	Magnetic Field Strength E	nT	FSM	2 normal	FLOAT
2FA	762	Magnetic Field Strength E	nT	FSM	2 normal	FLOAT
2FB	763	Magnetic Field Strength E	nT	FSM	2 normal	FLOAT
2FC	764	Magnetic Field Strength E	nT	FSM	2 normal	FLOAT
2FD	765	Magnetic Field Strength E	nT	FSM	2 normal	FLOAT
2FE	766	Magnetic Field Strength E	nT	FSM	2 normal	FLOAT
2FF	767	Magnetic Field Strength E	nT	FSM	2 normal	FLOAT
302	770	Magnetic Field Strength D	nT	FSM	2 normal	FLOAT
303	771	Magnetic Field Strength D	nT	FSM	2 normal	FLOAT
304	772	Magnetic Field Strength D	nT	FSM	2 normal	FLOAT
305	773	Magnetic Field Strength D	nT	FSM	2 normal	FLOAT
306	774	Magnetic Field Strength D	nT	FSM	2 normal	FLOAT
307	775	Magnetic Field Strength D	nT	FSM	2 normal	FLOAT
308	776	Magnetic Field Strength D	nT	FSM	2 normal	FLOAT
309	777	Magnetic Field Strength D	nT	FSM	2 normal	FLOAT
30C	780	RPM & Temperature Motor A1	1/min & K/10	MCU	2 normal	SHORT2
30D	781	RPM & Temperature Motor A2	1/min & K/10	MCU	2 normal	SHORT2
30E	782	RPM & Temperature Motor B1	1/min & K/10	MCU	2 normal	SHORT2
30F	783	RPM & Temperature Motor B2	1/min & K/10	MCU	2 normal	SHORT2
310	784	RPM & Temperature Motor C1	1/min & K/10	MCU	2 normal	SHORT2
311	785	RPM & Temperature Motor C2	1/min & K/10	MCU	2 normal	SHORT2
312	786	RPM & Temperature Motor D1	1/min & K/10	MCU	2 normal	SHORT2
313	787	RPM & Temperature Motor D2	1/min & K/10	MCU	2 normal	SHORT2
315	789	MavLink	binary data	TU	2 normal	UCHAR4
384	900	Baro Temperature		BSU	3 low	FLOAT
385	901	Baro Temperature		BSU	3 low	FLOAT
386	902	Baro Temperature		BSU	3 low	FLOAT
387	903	Baro Temperature		BSU	3 low	FLOAT

ID_{16}	ID_{10}	Information/Dienst	Einheit	Quelle	Priorität	Datentyp
388	904	Baro Temperature		BSU	3 low	FLOAT
389	905	Baro Temperature		BSU	3 low	FLOAT
38A	906	Baro Temperature		BSU	3 low	FLOAT
38B	907	Baro Temperature		BSU	3 low	FLOAT
38E	910	GPS Altitude (MSL/WGS84)	m	GSU	3 low	FLOAT
38F	911	GPS Altitude (MSL/WGS84)	m	GSU	3 low	FLOAT
390	912	GPS Altitude (MSL/WGS84)	m	GSU	3 low	FLOAT
391	913	GPS Altitude (MSL/WGS84)	m	GSU	3 low	FLOAT
392	914	GPS Altitude (MSL/WGS84)	m	GSU	3 low	FLOAT
393	915	GPS Altitude (MSL/WGS84)	m	GSU	3 low	FLOAT
394	916	GPS Altitude (MSL/WGS84)	m	GSU	3 low	FLOAT
395	917	GPS Altitude (MSL/WGS84)	m	GSU	3 low	FLOAT
398	920	GPS Ground Speed	m/s	GSU	3 low	FLOAT
399	921	GPS Ground Speed	m/s	GSU	3 low	FLOAT
39A	922	GPS Ground Speed	m/s	GSU	3 low	FLOAT
39B	923	GPS Ground Speed	m/s	GSU	3 low	FLOAT
39C	924	GPS Ground Speed	m/s	GSU	3 low	FLOAT
39D	925	GPS Ground Speed	m/s	GSU	3 low	FLOAT
39E	926	GPS Ground Speed	m/s	GSU	3 low	FLOAT
39F	927	GPS Ground Speed	m/s	GSU	3 low	FLOAT
3A0	928	GPS Status		GSU	3 low	ULONG
3A1	929	GPS Status		GSU	3 low	ULONG
3A2	930	GPS Status		GSU	3 low	ULONG
3A3	931	GPS Status		GSU	3 low	ULONG
3A4	932	GPS Status		GSU	3 low	ULONG
3A5	933	GPS Status		GSU	3 low	ULONG
3A6	934	GPS Status		GSU	3 low	ULONG
3A7	935	GPS Status		GSU	3 low	ULONG
3A8	936	GPS Speed N	m/s	GSU	3 low	FLOAT
3A9	937	GPS Speed N	m/s	GSU	3 low	FLOAT
3AA	938	GPS Speed N	m/s	GSU	3 low	FLOAT
3AB	939	GPS Speed N	m/s	GSU	3 low	FLOAT
3AC	940	GPS Speed N	m/s	GSU	3 low	FLOAT
3AD	941	GPS Speed N	m/s	GSU	3 low	FLOAT
3AE	942	GPS Speed N	m/s	GSU	3 low	FLOAT
3AF	943	GPS Speed E	m/s	GSU	3 low	FLOAT
3B0	944	GPS Speed E	m/s	GSU	3 low	FLOAT
3B1	945	GPS Speed E	m/s	GSU	3 low	FLOAT
3B2	946	GPS Speed E	m/s	GSU	3 low	FLOAT
3B3	947	GPS Speed E	m/s	GSU	3 low	FLOAT
3B4	948	GPS Speed E	m/s	GSU	3 low	FLOAT
3B5	949	GPS Speed E	m/s	GSU	3 low	FLOAT

ID_{16}	ID_{10}	Information/Dienst	Einheit	Quelle	Priorität	Datentyp
3B6	950	GPS Speed D	m/s	GSU	3 low	FLOAT
3B7	951	GPS Speed D	m/s	GSU	3 low	FLOAT
3B8	952	GPS Speed D	m/s	GSU	3 low	FLOAT
3B9	953	GPS Speed D	m/s	GSU	3 low	FLOAT
3BA	954	GPS Speed D	m/s	GSU	3 low	FLOAT
3BB	955	GPS Speed D	m/s	GSU	3 low	FLOAT
3BC	956	GPS Speed D	m/s	GSU	3 low	FLOAT
3BD	957	GPS Track	Degree	GSU	3 low	FLOAT
3BE	958	GPS Track	Degree	GSU	3 low	FLOAT
3BF	959	GPS Track	Degree	GSU	3 low	FLOAT
3C0	960	GPS Track	Degree	GSU	3 low	FLOAT
3C1	961	GPS Track	Degree	GSU	3 low	FLOAT
3C2	962	GPS Track	Degree	GSU	3 low	FLOAT
3C3	963	GPS Track	Degree	GSU	3 low	FLOAT
3C4	964	GPS Time (UTC + X seconds)	sec	GSU	3 low	ULONG
3C5	965	GPS Time (UTC + X seconds)	sec	GSU	3 low	ULONG
3C6	966	GPS Time (UTC + X seconds)	sec	GSU	3 low	ULONG
3C7	967	GPS Time (UTC + X seconds)	sec	GSU	3 low	ULONG
3C8	968	GPS Time (UTC + X seconds)	sec	GSU	3 low	ULONG
3C9	969	GPS Time (UTC + X seconds)	sec	GSU	3 low	ULONG
3CA	970	GPS Time (UTC + X seconds)	sec	GSU	3 low	ULONG
3D4	980	System Shutdown	cmd	SHU	3 low	UCHAR

Tabelle B.2.: Neue ID Distribution für UAVs - NOD

Die nachfolgende Grafik B.1 stellt die Verteilung der einzelnen CAN-Nachrichten innerhalb des Identifierbereichs von Normal Operation Data dar. Hier ist der zweite Entwurf der Identifierdistribution geplottet. Die einzelnen Prioritätsklassen sind farblich markiert.



Farbe	Priorität
Rot	0 - critical
Grün	1 - high
Blau	2 - normal
Magenta	3 - low
Grau	nicht belegt

Abbildung B.1.: Neue ID Distribution für UAVs - NOD

C. Kollisionsabbildung im Simulationsmodell

Das unten stehende Listing C.1 zeigt die Arbitrierungslogik im CAN-Bus Simulationsmodell FiCo4OMNeT. Hier ist die Diskrepanz zwischen realem Busverhalten und dessen Abbildung in der Simulation zu erkennen.

Stand:
10.07.2017

Datei:
https://core4inet.core-rg.de/trac/browser/FiCo4OMNeT/src/fico4omnet/bus/can/CanBusLogic.cc

```
1  void CanBusLogic::grantSendingPermission() {
2      currentSendingID = INT_MAX;
3      sendingNode = NULL;
4
5      for (std::list<CanID*>::iterator it = ids.begin(); it != ids.end(); ++it) {
6          CanID *id = *it;
7          if (id->getCanID() < currentSendingID) {
8              currentSendingID = id->getCanID();
9              sendingNode = dynamic_cast<CanOutputBuffer*> (id->getNode());
10             currsit = id->getSignInTime();
11         }
12     }
13
14     int sendcount = 0;
15     bool nodeFound = false;
16     for (std::list<CanID*>::iterator it = ids.begin(); it != ids.end(); ++it) {
17         CanID *id = *it;
18         if (id->getCanID() == currentSendingID) {
19             if (id->getRtr() == false) { //Data-Frame
20                 sendcount++;
21                 if (!nodeFound) {
22                     nodeFound = true;
23                     sendingNode = dynamic_cast<CanOutputBuffer*> (id->getNode());
24                     currsit = id->getSignInTime();
25                     eraseids.push_back(it);
26                 }
27             } else {
```

```
28                  eraseids.push_back(it);
29              }
30          }
31      }
32
33      if (sendcount > 1) {
34          cComponent::bubble("More_than_one_node_sends_with_the_same_ID.");
35      }
36      if (sendingNode != NULL) {
37          CanOutputBuffer* controller = check_and_cast<CanOutputBuffer *>(
38                  sendingNode);
39          controller->receiveSendingPermission(currentSendingID);
40      } else {
41          idle = true;
42          getDisplayString().setTagArg("tt", 0, "state:_idle");
43          bubble("state:_idle");
44      }
45  }
```

Listing C.1: Arbitrierung im FiCo4OMNet Model

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 19. Januar 2018 Ulrich ter Horst