Ein Service-Discovery-Protokoll für Netze heterogener Sensoreinheiten

A Service Discovey Protocol for Networks of heterogeneous Sensor Units

Master Thesis

vorgelegt von

Malte Christian Struck

Universität Bremen

Fachbereich 3

Studiengang Informatik

1. Gutachter: Prof. Dr. Rolf Drechsler

2. Gutachter: Dr. Marco Scharringhausen

Abgabedatum: 19.11.2018

Erklärung zur Selbstständigkeit

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst und keine anderen als die angegebenen Hilfsmittel verwendet habe.
Insbesondere versichere ich, dass ich alle wörtlichen und sinngemäßen Übernahmen aus anderen Werken als solche kenntlich gemacht habe.

Bremerhaven, 19.11.2018

Unterschrift

Wir dürfen das Weltall nicht einengen, um es den Grenzen unseres Vorstellungsvermögens anzupassen, wie der Mensch es bisher zu tun pflegte. Wir müssen vielmehr unser Wissen ausdehnen, sodass es das Bild des Weltalls zu fassen vermag.

Sir Francis von Verulam Bacon

Zusammenfassung

Bei einem Weltraumsystem ist im Allgemeinen von Beginn der Mission an bekannt, welche Komponenten eine technische Einheit (Remote Unit) zur Verfügung hat und welche Daten hierfür übertragen werden müssen.
Innerhalb dieser Master Thesis wird mithilfe eines Service Discovery-Protokolls die Möglichkeit geschaffen, eine veränderliche Menge an Instrumenten aus Sensoren und Aktoren zu nutzen.
Eine in dieser Arbeit umgesetzte grafische Benutzeroberfläche sucht im lokalen Subnetz nach Remote Units und passt sich hierbei automatisiert an, indem für jede Einheit ein Widget generiert wird, das für alle angeschlossenen Sensoren und Aktoren ein Anzeige- bzw. Bedienfeld enthält.
Um mehr Sensoren und Aktoren zu nutzen, als durch den Adressraum der genutzten Schnittstelle I^2C möglich sind, wurde eine CAN-I^2C-Bridge entwickelt, die vom I^2C-Adressraum abstrahiert und es ermöglicht, nahezu beliebig viele Komponenten zu nutzen.

Kapitelübersicht

Inhaltsverzeichnis

Abkürzungsverzeichnis

AU Astronomical Unit; Astronomische Einheit (ca. 150.000.000km)

APID Application Process ID

CAN Controller Area Network

CC Command Center

CCSDS Consultative Committee for Space Data Systems

DLR Deutsches Zentrum für Luft- und Raumfahrt

ECSS European Cooperation for Space Standardization

GPIO General Purpose Input / Output

GUI Graphical User Interface

I/O Input / Output

IP Internet Protocol; auch kurzform für IP-Adresse

LSB Least Significant Bit; niederwertigstes Bit

MSB Most Significant Bit; höchstwertiges Bit

NAT Network Address Translation

OUTPOST Open modUlar sofTware PlatfOrm for SpacecrafT

PWM Pulse Width Modulation; Pulsweitenmodulation

ROBEX Robotische Exploration unter extremen Bedingungen

RU Remote Unit

SoC System-on-Chip

SST Service Subtype; auch Subservice Type

ST Service Type

TC Telecommand; Telekommando

TCP Transmission Control Protocol

TM Telemetry; Telemetrie

UDP User Datagram Protocol

VM Virtual Machine; Virtuelle Maschine

ZIF Zero Insertion Force

Abbildungsverzeichnis

Tabellenverzeichnis

List of Algorithms

1 Einleitung

Während meines Studiums begann ich meine Arbeit im Deutschen Forschungszentrum für Luft- und Raumfahrt und arbeitete fortan im Projekt ROBEX (Robotische Exploration unter extremen Bedingungen). Ich machte hierbei viele Erfahrungen über die Ausgestaltung einer Mission und die Teilarbeiten, die zu deren Gelingen führen.
Zunächst befasste ich mich mit der grafischen Benutzeroberfläche, die genutzt wird, um ausgesetzte Einheiten zu kontrollieren. Mit der Zeit übernahm ich auch weitere Aufgaben, zu denen unter anderem die Programmierung der Remote Units, aber auch ingenieurtechnische Leistungen gehörten. Im Laufe des Projektes sammelte ich verschiedene Ideen zur Fortführung, Erweiterung sowie Steigerung der Nutzbarkeit des Projektes. Da ich jedoch zu einer späten Phase in das Projekt einstieg, konnten diese Vorschläge, welche im Anhang im Unterabschnitt 12.2: Robex *Reloaded* festgehalten sind, nicht mehr umgesetzt werden.
Einer der Vorschläge zur Fortführung des Projektes ROBEX betraf die Konsruktion einer Remote Unit auf einer anderen Hardwarebasis als der bisher eingesetzte Compact On-Board Computer (COBC) mit off-the-shelf-Komponenten wie einem ESP8266 oder einem Raspberry Pi. Ein weiterer Vorschlag sah vor, die Entwicklung und Implementierung neuer Funktionen zu vereinfachen, indem ein Teil der Kommunikationskette *Remote Unit - GUI* generisch gehalten wird, sodass eine Veränderung des Setups nicht mehr zu Eingriffen auf beiden Seiten, sondern nur noch auf einer führt. Sinnhafterweise bleibt die Remote Unit an die jeweilige Mission anpassbar, wohingegen sich die grafische Benutzeroberfläche den Fähigkeiten der Remote Unit anpasst.

Notwendig für diese Art der Service Discovery ist ein Protokoll, welches die GUI über Veränderungen der Remote Unit informiert und entsprechende Anpassungen der grafischen Oberfläche automatisiert erlaubt. Darüber hinaus soll das Protokoll eine Skalierbarkeit erlauben, indem innerhalb einer Domäne alle Remote Units aufgefunden werden können. Somit ergeben sich erste Rahmenanforderungen für diese Master Thesis.

In dieser Arbeit werden alle Komponenten, die zur Umsetzung des beschriebenen Protokolls notwendig sind, entwickelt und evaluiert. Es wurde eine Remote Unit auf Basis eines Raspberry Pi entworfen, die in ihrem gesendeten Datenstrom (neben den eigentlichen Messdaten) Informationen über ihr Setup - d.h. die an sie angeschlossenen Sensoren - enthält. Ein in dieser Arbeit festgelegtes Protokoll definiert dabei die Codierung von Sensortypen und gemessenen Daten. Diese wiederum werden von einer grafischen Benutzeroberfläche ausgewertet und resultieren in einer angepassten Anzeige.

Aufgrund der verwendeten Sensorschnittstelle ergibt sich eine Einschränkung bezüglich der gleichzeitig nutzbaren Sensoren. Um diese Einschränkung zu umgehen, wurde eine weitere Kommunikationsschnittstelle zwischen Remote Unit und Sensor konstruiert und implementiert. Diese ermöglicht, nahezu beliebig viele Sensoren zu nutzen und wissenschaftliche Missionen entsprechend ihrem Anspruch auszustatten.

Da kommunikationstechnisch zwischen einer Sensorabfrage und einer Aktoranweisung kein großer Unterschied besteht und es für wissenschaftliche Missionen sinnvoll sein kann, im Umfeld der Messinstrumente Manipulationen durchzuführen, wurde das Protokoll zur Service Discovery - sowie die Fähigkeiten der Benutzeroberfläche und der Remote Unit - dahingehend angepasst, dass auch Aktoren erkannt, dem Nutzer angezeigt und gesteuert werden können.

In dieser Arbeit werden zunächst technische Grundlagen, insbesondere im Kontext von ROBEX-artigen Missionen, erklärt, sodass mithilfe dieser im Anschluss an eine Protokolldefinition die Implementierung von GUI, Remote Unit sowie der genutzen Sensorschnittstelle und dem Kommunikationsadapter zur Adressraumerweiterung von Sensoren und Aktoren beschrieben wird. Hierbei findet eine Evaluation des Protokolls bereits durch die Anwendung bei der Kommunikation zwischen der entwickelten Remote Unit und der implementierten GUI statt. Ergebnisse einer systematischen Untersuchung der Forschungsfrage, ob und wie das *Protokoll zur Service Discovery für Netze heterogener Sensoreinheiten* nutzbar ist, werden am Ende dieser Arbeit vorgestellt.

2 Allgemeine Grundlagen

In diesem Abschnitt werden Grundlagen dargelegt, die im Kontext dieser Arbeit stehen. Zunächst wird die Kommunikation in Netzen anhand des OSI-Referenzmodells und des darauf aufbauen TCP/IP-Referenzmodells beschrieben. Im Anschluss wird die Datenübertragung bei Weltraummissionen fokussiert, wobei auf Standards sowohl zur Kommunikation als auch zum Service-Konzept eingegangen wird. Weiterhin soll die Bibliothek *OUTPOST* erläutert werden, die eine Teilmenge der vorangegangenen Kommunikations- und Servicestandards implementiert. Zuletzt wird die Kommunikation mithilfe des Controller Area Network (CAN) beschrieben.

2.1 OSI-Referenzmodell

Bei voneinander entfernten Systemen muss die Frage gestellt werden, ob und in welcher Form Daten zwischen diesen übertragen werden.
Im Folgenden wird das OSI-Referenzmodell dargestellt, welches eine Datenübertragung mit verschiedenen Schichten zwischen entfernten Systemen modelliert. Der Aufbau in Schichten ermöglicht die Abstraktion von darunterliegenden Schichten, sodass das Protokoll einer Schicht nur schichtspezifische Aufgaben bearbeiten muss.

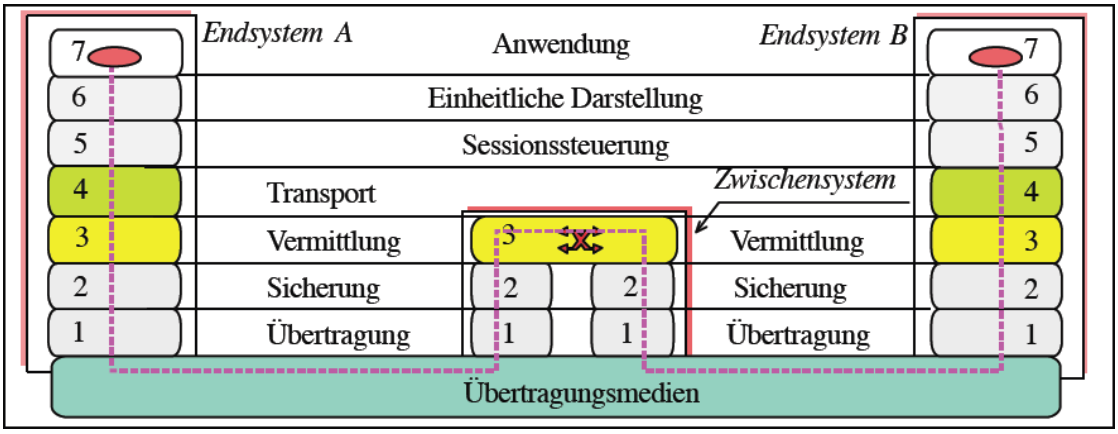


Abbildung 1: OSI-Referenz-Modell: Zerlegung der Kommunikationsaufgabe in 7 Schichten
Quelle: [BH14], S. 25.

Abbildung 1 zeigt die Kommunikation von Anwendungen auf den Endsystemem A und B sowie eine dazwischen geschaltete Übertragungsstation. Es ist für die Anwendungen auf den Endsystemen nicht relevant, wie darunterliegende Übertragungsschichten aufgebaut sind. Alle

Schichten haben eine spezielle Funktion, die im Folgenden (von unten nach oben) erläutert werden.

Physical Layer
Die Übertragung über die physikalische Schicht (engl. Physical Layer, im Bild Schicht 1) definiert die Bitübertragung auf einem Medium, d.h. ihre Eigenschaften und Abläufe beim Datentransfer. Ein Beispiel ist die elektrische Übertragung digitaler Signale. Diese Signale sind sowohl zeitlich als auch bezüglich ihres Wertes analog und werden erst durch ein geeignetes Sampling und eine Quantisierung zum abstrakten digitalen Wert, der anschließend durch eine höhere Schicht weiterverarbeitet werden kann.

Data Link Layer
Die zweite Schicht, welche sich über dem Physical Layer befindet, wird als Sicherungsschicht (engl. Data Link Layer) bezeichnet. Sie ermöglicht eine sichere (korrekte) Datenübertragung zwischen zwei benachbarten Geräten (Knoten). Bits werden in Frames zusammengefasst und mit einer Prüfsumme versehen, was die Erkennung nicht korrekt übertragener Daten ermöglicht.

Network Layer
Die Netzwerkschicht bzw. Vermittlungsschicht (engl. Network Layer) sorgt für die Übermittlung von Paket genannten Datenblöcken zwischen Endsystemen, die nicht zwingend direkt, sondern auch über Zwischenstationen miteinander verbunden sind.

Transport Layer
Um eine virtuelle Ende-zu-Ende-Verbindung bereitzustellen, gibt es die Transportschicht (Transport Layer). Ziel ist es, fehlerhafte Datenübertragungen zu erkennen und gegebenenfalls zu korrigieren. Sie ist die letzte netzorientierte Schicht.

Session Layer
Die Sitzungsschicht (engl. Session Layer) ist die erste anwendungsorientierte Schicht. Sie regelt den Auf- und Abbau sowie gegebenenfalls die Wiederherstellung von Verbindungen. Innerhalb dieser Schicht finden Synchronisationsprozesse innerhalb der Kommunikation statt.

Presentation Layer
Die Darstellungsschicht (engl. Presentation Layer) regelt die einheitliche Darstellung der übertragenen Informationen, also etwa den benutzten Zeichensatz. Ebenfalls können auf dieser Ebene Daten komprimiert oder verschlüsselt werden.

Application Layer
In der Anwendungsschicht (engl. Application Layer) finden anwendungsspezifische Protokolle ihren Platz.

2.2 TCP/IP-Referenzmodell

Der TCP/IP-Stack beschreibt eine vereinfachte Umsetzung des OSI-Referenzmodells, bei dem die obersten drei Schichten (Application Layer bis Session Layer) zu einer anwendungsorientierten Schicht zusammengefasst sind. Weiterhin beinhaltet das Modell keine Spezifikation zur Bitübertragung auf dem Physical Layer. Es basiert auf Überlegungen zum ARPANET und ist heute Grundlage zur Nutzung des Internets sowie für lokale (LAN, Local Area Network) und gebietsübergreifende (WAN, Wide Area Network) Netzwerke. Ziel ist es, innerhalb eines teilweise vermaschten Netzes Daten übertragen zu können, auch wenn einzelne Knoten ausfallen. Tanenbaum ([TW12]) beschreibt das TCP/IP-Referenzmodell als Vier-Schichten-Modell ohne Berücksichtigung der physikalischen Übertragung. Abbildung 2 zeigt den Vergleich des OSI-Referenzmodells zum TCP/IP-Referenzmodell nach Tanenbaum. Abbildung 3 zeigt die vier Schichten des TCP/IP-Referenzmodells nach Tanenbaum und einige darin verwendete Protokolle.

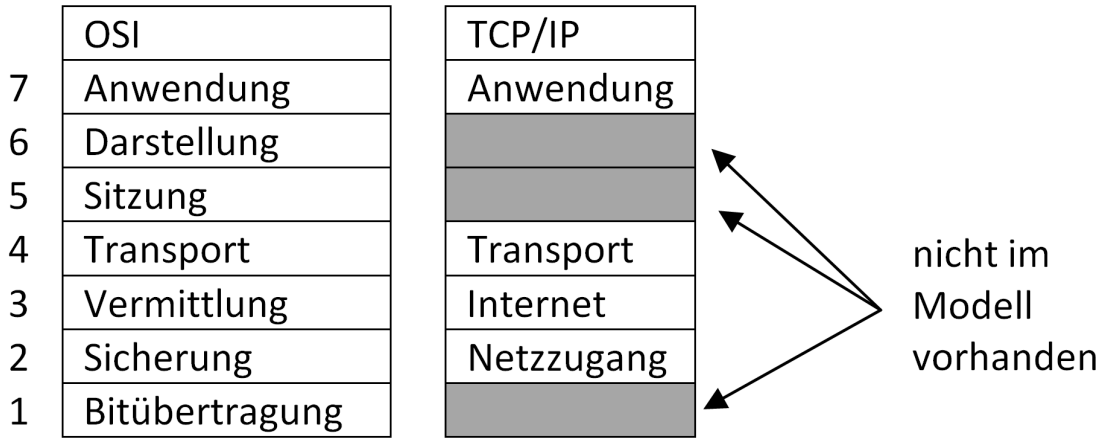


Abbildung 2: Schichten des TCP/IP-Referenzmodells im Vergleich zum OSI-Referenzmodell
Quelle: Eigenes Bild, in Anlehnung an [TW12], S. 72.

Die häufig im Kontext von Internet und Heimnetz genutzten Protokolle sind IP (Internet Protocol) auf der Vermittlungsschicht sowie TCP (Transport Control Protocol) und UDP (User Datagram Protocol) auf der Transportschicht. Das Internet Protocol sorgt dafür, dass Datenpakete zwischen Rechnern ausgetauscht werden können, d.h. ein Pfadfinden und -nutzen

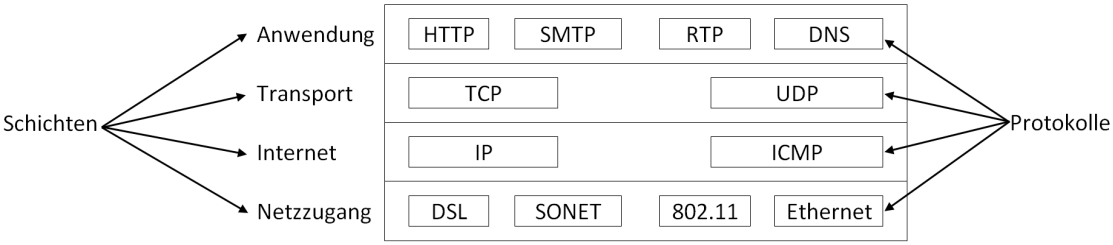


Abbildung 3: Schichten des TCP/IP-Referenzmodells und verwendete Protokolle
Quelle: Eigenes Bild, in Anlehnung an [TW12], S. 73.

stattfindet (Routing). TCP sorgt für eine Transportsicherung und ist verbindungsorientiert. Dies meint, dass eine TCP-Verbindung zunächst aufgebaut und am Ende geschlossen werden muss sowie dass die erneute Übertragung verlorengegangener Pakete sichergestellt wird. UDP ist hingegen verbindungslos und bietet keine Transportkontrolle. Beiden Protokollen ist gemein, dass sie einen 16-Bit-Wert bereitstellen, der Port genannt wird und für die lokale (auf einem Gerät stattfindende) Adressierung zuständig ist. Die Protokolle IP, TCP und UDP werden im Folgenden vorgestellt.

2.2.1 Internet Protocol (IP)

Das Internet Protocol sorgt für die Vermittlung von Daten zwischen Endgeräten. Es gibt zur Zeit die zwei gebräuchlichen Varianten IPv4 und IPv6, welche den Protokollversionen 4 und 6 des Internet Protocols entsprechen. Der IPv4-Header, der übertragenen Nutzdaten vorangestellt wird und zur Vermittlung notwendige Informationen enthält, wird exemplarisch erläutert.

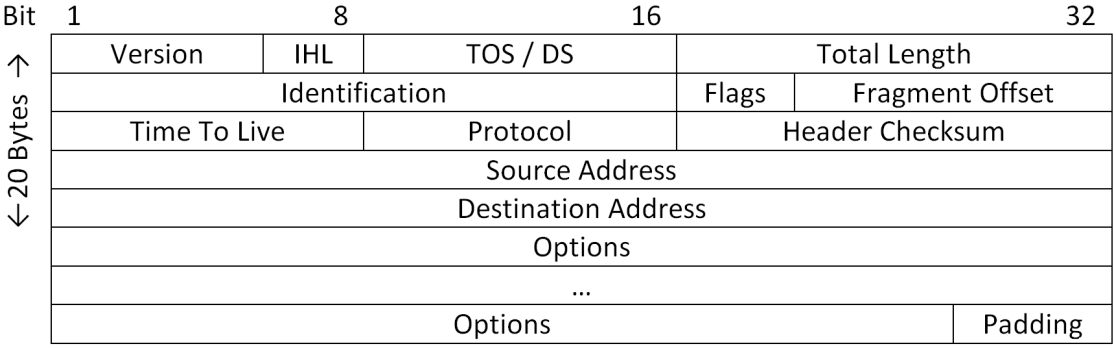


Abbildung 4: Aufbau eines IPv4-Headers und eines IPv4-Paketes
Quelle: Eigenes Bild, in Anlehnung an [BH14], S. 65.

Die Bedeutung der einzelnen Felder wird nachfolgend erklärt.

- **Version**
  Zu Beginn eines IP-Paketes ist die Version des Protokolls enthalten. Sie ist bei IPv4 4 (und bei IPv6 6).

- **IHL**
  Die Internet-**H**eader-**L**ength gibt die Länge des Paket-Headers in 32-Bit-Worten an. Sie beträgt mindestens fünf, was einer Header-Length von 20 Bytes entspricht.

- **TOS/DS T**ype **o**f **S**ervice/**D**ifferentiated **S**ervice wird verwendet, um eine Quality-of-Service-Funktion zu ermöglichen. Somit können Pakete priorisiert werden.

- **Total Length**
  Total Length beschreibt die Gesamtlänge des IP-Paketes in Bytes. Mithilfe von 16 Bits ergibt sich eine Gesamtlänge von maximal 65535 Bytes, was abzüglich der Mindestgröße des Headers von 20 Bytes eine Nutzlast von maximal 65515 Bytes erlaubt.

- **Identification**
  Dieses Feld wird benutzt, wenn IP-Pakete segmentiert bzw. fragmentiert werden und dient als „Seriennummer“ eines Paketes. Somit ist es möglich, ein großes Datenpaket in mehrere kleine aufzuteilen und anschließend wieder zusammenzufügen.

- **Flags**
  Das Feld Flags ist drei Bits groß und gibt an, ob das Paket (weiter) fragmentiert werden darf und ob weitere Fragmente folgen.

- **Fragment Offset**
  Wenn das Flag gesetzt ist, welches angibt, dass weitere Fragmente folgen, gibt das Fragment Offset an, an welcher Stelle der Originaldaten das entsprechende Fragment einzufügen ist.

- **Time To Live (TTL)**
  Die TTL gibt an, wieviele weitere Netzknoten das IP-Paket passieren darf (Hops). An jedem Knoten wird diese Zahl um 1 dekrementiert. Eine TTL von 0 bedeutet, dass das Paket verworfen werden muss. Hierdurch lässt sich unter anderem verhindern, dass ein Paket „im Kreis“ geschickt wird. Auch kann dieses Feld so genutzt bzw. interpretiert werden, dass es die Grenzen eines Teilnetzes nicht verlässt (wenn die TTL unterhalb eines bestimmten Schwellenwertes liegt).

- **Protocol**
  Das Feld Protocol stellt Informationen über das in der höheren Schicht eingesetzte

Protokoll zur Verfügung. Beispielsweise hat TCP die Protokollnummer 6 und UDP die Protokollnummer 17.

- **Header Checksum**
  Um zu prüfen, ob der IP-Header (inklusive eventueller Optionen) korrekt übertragen wurde, wird eine Checksumme gebildet und in diesem Feld gespeichert. Die Checksumme der Nutzlast (beispielsweise ein TCP- oder UDP-Paket) wird in dieser selbst gespeichert.

- **Source Address**
  Das Feld Source Address gibt einen 32-Bit-Wert an (bei IPv4), der eine eindeutige Zuordnung zum Quellrechner erlaubt.

- **Destination Address**
  Dieses Feld gibt die eindeutige Adresse des Zielrechners an.

- **Options**
  Das nicht notwendigerweise vorhandene Feld Options wird genutzt, etwa um Zeitmarken oder Routing-Informationen zu speichern.

- **Padding**
  Da die Größe des IP-Headers immer ein Vielfaches von 32 Bit sein muss, werden bei Optionen, die diesem Raster nicht entsprechen, Bits aufgefüllt, sodass die Headergröße einem Vielfachen von 32 Bit entspricht.

IPv6

Die Unterschiede bei IPv6 liegen im Wesentlichen in der Größe der Adressangabe, die nicht mehr 32 Bit, sondern 128 Bit beträgt. Dies ermöglicht, mehr Adressen als bisher zu vergeben, was durch den auf etwa vier Milliarden Adressen begrenzten Adressraum und der steigenden Zahl der Geräte im Internet nötig wurde.

2.2.2 Transmission Control Protocol (TCP)

TCP ist ein in der Transportschicht verortetes, verbindungsorientiertes Protokoll zur Absicherung des Datentransports (im Sinne der Erkennung eines Paketverlustes). Wird ein TCP-Paket übertragen, quittiert der Empfänger dieses mit einem weiteren Paket, einem Acknowledgement. Empfängt der Sender nach einer festgelegten Zeitspanne kein Acknowledgement, geht er davon aus, dass das Paket verloren ist und sendet dieses erneut.

Abbildung 5 zeigt den Header eines TCP-Paketes. Auch ist mit TCP eine Überlastungsüberwachung (Flusskontrolle) möglich, d.h. wenn die Übertragungskapazität eines empfangenden oder zwischenliegenden Knotens ausgeschöpft[1] ist, kann der Sender veranlasst werden, Pakete langsamer zu versenden. Weiterhin können Daten in Segmente zerlegt werden, um sie beim Empfänger wieder zusammenzusetzen.

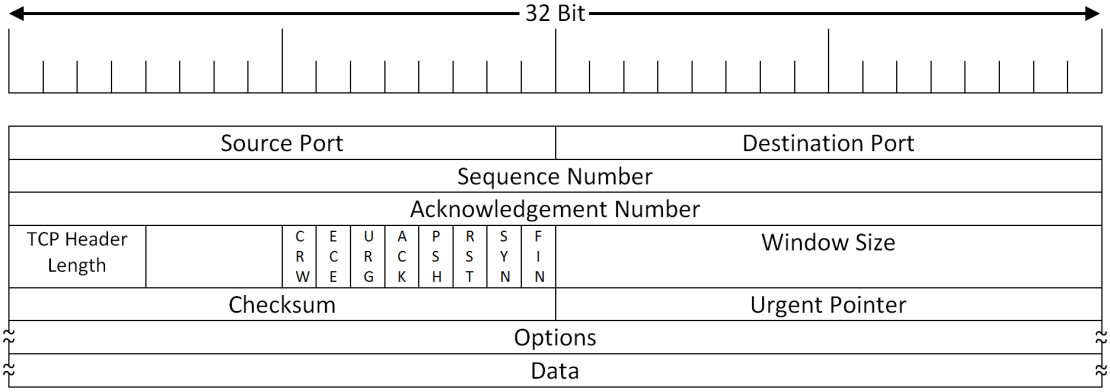


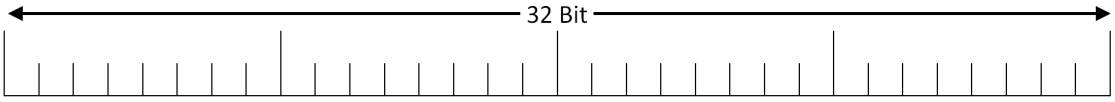
Abbildung 5: TCP-Header
Quelle: Eigenes Bild, in Anlehnung an [TW12], S. 633.

Die beiden Felder Source Port und Destination Port sind jeweils ein 16-Bit-Wert und können als eine Art Hausnummer innerhalb eines Rechners verstanden werden. Mithilfe des Ports kann das Paket einer bestimmten Anwendung, genauer: einem vorher beim Betriebssystem angemeldeten Socket dieser Anwendung, zugeordnet und entsprechend weitergeleitet werden. Die Sequenznummer (engl. Sequence Number) ist eine fortlaufende Nummer, die zusammen mit der Bestätigungsnummer (Acknowledgement Number) verwendet wird, um verlorengegangene Pakete zu identifizieren.

2.2.3 User Datagram Protocol (UDP)

Das User Datagram Protocol (UDP) ist ein Protokoll zur verbindungslosen Übertragung von Daten. Es ist wesentlich einfacher aufgebaut als TCP; beispielsweise verzichtet es auf einen Verbindungsauf- und abbau.
Der Header eines UDP-Paketes ist mit 8 Bytes im Vergleich zu dem eines TCP-Paketes (mindestens 20 Bytes) kleiner und enthält neben Angaben zum Quell- und zum Zielport auch solche über die Größe des gesamten UDP-Paketes (inkl. Header) und eine Prüfsumme. Abbildung 6 zeigt den Aufbau eines UDP-Headers.

[1] Dieses wird erkannt, indem Pakete verworfen werden müssen und der Absender hierüber eine Information erhält.



Source Port	Destination Port
UDP Length	UDP Checksum

Abbildung 6: Header eines UDP-Pakets
Quelle: Eigenes Bild, in Anlehnung an [TW12], S. 617.

2.3 Datenübertragung im Kontext von Weltraummissionen

Im Allgemeinen ist es bei Weltraummissionen notwendig, Daten zu übertragen. Bei vielen Missionen erfüllen die ausgebrachten Geräte nicht die Anforderung, autonom zu agieren und müssen dementsprechend ferngesteuert werden. Auch selbststeuernde Maschinen - etwa solche zur Erkundung ferner Planeten wie dem Mars, bei denen eine Fernsteuerung aufgrund der Übertragungsdauer von in diesem Fall bis zu 1250 Sekunden[2] nicht sinnvoll ist - benötigen zur Übertragung der Ergebnisse wissenschaftlicher Messungen eine Kommunikationsmöglichkeit.

Weltraummissionen werden im Allgemeinen erst möglich, wenn viele Menschen zusammenarbeiten. Um dieses zu erleichtern, ist es sinnvoll, gemeinsame Schnittstellen zu definieren, mithilfe derer aus einzelnen Teilarbeitsleistungen[3] ein zur Missionserfüllung notwendiges Produkt (beispielsweise ein Weltraumteleskop) entsteht. Es ist meines Erachtens sinnvoll, missionsübergreifend Standards zu definieren, sodass beispielsweise nicht bei jeder Mission neu festgelegt werden muss, wie etwa Kommunikation zwischen verschiedenen Geräten abläuft. Weiterhin sind bei vielen Missionen bestimmte Anforderungen gleich, etwa die regelmäßig stattfindende Übertragung von Housekeeping-Daten (beispielsweise Batteriespannung und -temperatur).
Daher finden sich - analog zu Standardisierungsgremien der Industrie wie etwa ISO oder DIN - solche, die die Raumfahrt betreffen. Zu nennen sind im Kontext dieser Arbeit vor allem die *European Cooperation for Space Standardization* (ECSS)[4] und das *Consultative Committee for Space Data Systems* (CCSDS)[5]. Diese definieren Nachrichtenformate und Standard Services.

[2] Bei einem Abstand von maximal ca. 1 AU (Erde) + 1,5 AU (Mars) = ca. 375.000.000km.
[3] Hierbei sind sowohl Hardware als auch Software gemeint.
[4] `http://ecss.nl/`
[5] `https://public.ccsds.org/default.aspx`

Im Folgenden werden Aspekte zur Datenübertragung von Telekommandos und Telemetriedaten beschrieben sowie standardisierte Dienste an Bord von Weltraumsystemen vorgestellt.

2.3.1 Allgemeines

Im Folgenden wird die Kommunikation zwischen einer Basisstation auf der Erde sowie einer entfernten technischen Einheit im Weltraum betrachtet. Diese entspricht einer Master-Slave-Architektur, bei der einzelne Aktionen vom Master (der Basisstation) initiiert werden. Die entfernte Einheit (Remote Unit) wird per Telekommando angewiesen, einen Befehl auszuführen und sendet währenddessen sowie im Anschluss Telemetrie (etwa mit Messergebnissen oder dem Ausführungsstatus) an die Basisstation.
Neben diesem sind grundsätzlich auch andere Szenarien denkbar, die etwa die Kommunikation zwischen Remote Units betreffen; auf die Spezifika dieser Variante wird in dieser Arbeit jedoch nicht eingegangen.

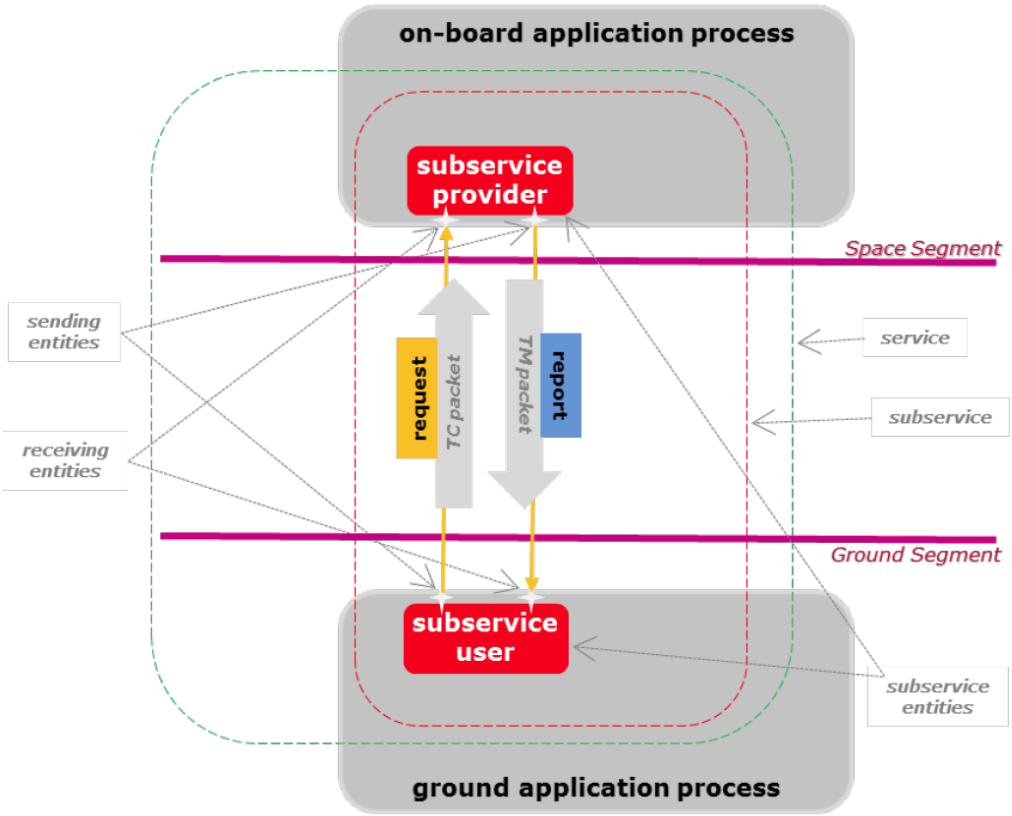


Abbildung 7: Service Konzept des Packet Utilization Standards
Quelle: [SS16], S. 19.

Abbildung 7 zeigt die zugrundeliegende Aufteilung in zwei Segmente (Ground Segment und Space Segment), die über Kommunikationskanäle miteinander verbunden sind. Die Bodenstation sendet Anfragen in Form von Telekommandos (**T**ele-**C**ommand-Packet) und empfängt Telemetrie (**T**ele-**M**etry-Packet). Erkennbar ist auch eine Hierarchie von Applications („application process“), Services und Subservices. Ein Tripel aus Application Process Identifier (im Folgenden APID genannt), Service-Bezeichner (mit Service Type beschrieben) und Subservice-Bezeichner (als Service Subtype bezeichnet) kennzeichnet eindeutig eine bestimmte Aktion bzw. eine bestimmte Anfrage.
Nachfolgend wird eine Kommunikationsarchitektur beschrieben, die es ermöglicht, Telekommandos und Telemetrie zwischen Bodenstation und Remote Unit auszutauschen. Anschließend werden Standard-Services vorgestellt, die im Kontext vieler Missionen nützlich sind, etwa den Service zur Bestätigung des Empfangs und der Ausführung von Befehlen (*Verification Service*).
In Abbildung 8 ist erkennbar, dass ECSS und CCSDS analog zu den Schichtenmodellen der vorangegangenen Abschnitte fünf Schichten zur Kommunikation definieren und darin verortete Protokolle nennen.

2.3.2 Pakettypen

Es bietet sich an, ein für viele Missionen einheitliches Paketformat zu wählen, um zum Einen die Zusammenarbeit zwischen Institutionen - etwa dem DLR und der japanischen Weltraumagentur JAXA - zu erleichtern und zum Anderen Programmbestandteile in zukünftigen Anwendungen wiederzuverwenden. Das in Abbildung 9 gezeigte Format eines Space Packets bietet die Möglichkeit, zwei (1 Bit steht für den Packet Type) unterschiedliche Subtypen zu definieren: Telekommandopakete (Telecommand packets / TCs) und Telemetriepakete (Telemetry packets/ TMs).
Erkennbar ist hierbei, dass die APID schon innerhalb des *Packet Primary Header*s angegeben wird; der Service Type und Service Subtype befinden sich im optionalen[6] *Packet Secondary Header*. Eine Sequenznummer dient zusammen mit der APID und dem Pakettyp der eindeutigen Kennzeichnung eines Pakets. Innerhalb des *user data field* können Nutzdaten hinzugefügt werden, bei Telekommandos etwa Parameter, die bei der Ausführung benötigt werden, oder bei Telemetriepaketen Daten einer Messung.

Telecommand Packets

Ein Telecommand Packet ist ein Space Packet, bei dem der Packet Type den Wert 1 hat. Der

[6] Alle Pakete außer CPDU-Packets (Telekommando zum direkten Hardwarezugriff) und Spacecraft Time Packets besitzen einen Secondary Header.

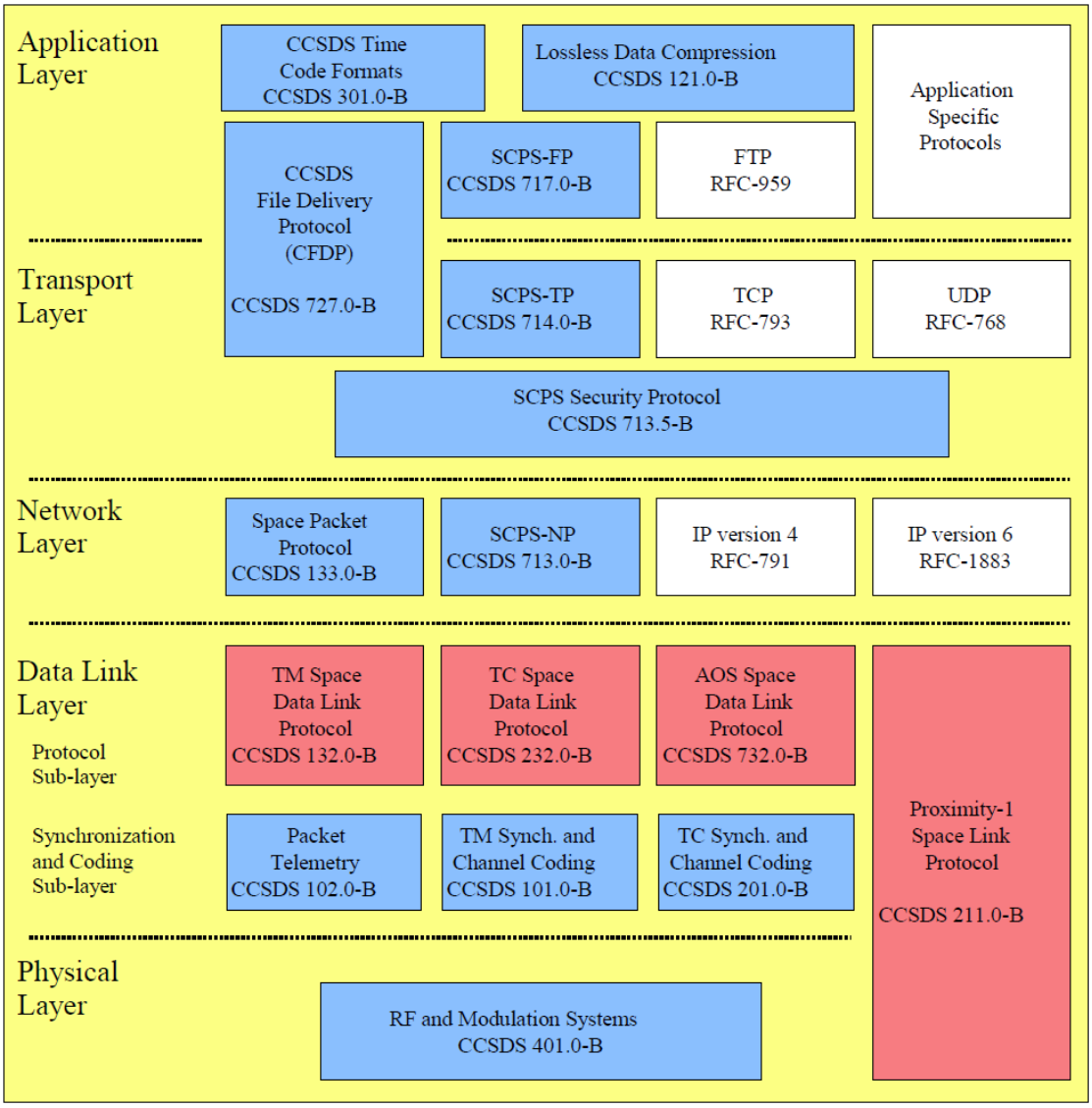


Abbildung 8: Modell eines Schichtaufbaus zur Übertragung von Daten nach ECSS und CCSDS
Quelle: [SS03], S. 29.

packet primary header							packet data field	
packet version number	packet ID			packet sequence control		packet data length	packet secondary header	user data field
	packet type	secondary header flag	application process ID	sequence flags	packet sequence count or packet name			
3 bits	1 bit	1 bit	11 bits	2 bits	14 bits	16 bits	variable	variable
2 octets				2 octets		2 octets	1 to 65536 octets	

Abbildung 9: The Space Packet Structure
Quelle: [SS16], S. 438.

Secondary Header (siehe Abbildung 10) eines Telekommandos beinhaltet neben der PUS[7]-Versionsnummer weitere Informationen. Acknowledgement Flags geben an, zu welchem Zeitpunkt (Empfang, Beginn der Ausführung, innerhalb der Ausführung, Beendigung der Ausführung) eine Bestätigung (ACK) gesendet bzw. ein Nichterfolg (NACK) gemeldet werden soll. Service Type ID und Message Subtype ID identifizieren Service und Subservice innerhalb einer Application. Source ID entspricht einer Auskunft über den Absender des Telekommandos und Spare meint den Bereich, der möglicherweise mit Füllbits aufgefüllt werden muss, um eine bestimmte Paketgröße zu erhalten.

TC packet PUS version number	acknowledgement flags	message type ID		source ID	spare
		service type ID	message subtype ID		
enumerated (4 bits)	enumerated (4 bits)	enumerated (8 bits)	enumerated (8 bits)	enumerated (16 bits)	fixed-size bit-string
					optional

Abbildung 10: Secondary Header eines Telekommandopaketes
Quelle: [SS16], S. 442.

Die Übertragung von Telekommandos kann innerhalb von weiteren Strukturen geschehen. Dies bietet sich an, um mehrere Telekommandos zusammen zu verschicken oder eine Priorisierung und Weiterverteilung anhand von virtuellen Kanälen vorzunehmen. Hierzu wird einem

[7] Packet Utilization Standard

Telekommando ein (optionaler) TC Segment Header vorangestellt, der wiederum in einem TC Transfer Frame eingebettet ist. Hier finden sich weitere Routing-Informationen, etwa die *Virtual Channel ID* oder die Spacecraft ID des empfangenden Weltraumgerätes.
Anzumerken ist, dass im Kontext dieser Arbeit Telekommandos (TC Source Packets) nicht in TC-Segmente sowie TC-Transfer-Frames eingebettet, sondern als UDP-Nutzlast gesendet werden. Das Einbetten von TC Source Packets wurde bereits in der Mission ROBEX erprobt, in der Telekommandos in TCP-Paketen eingebettet wurden.

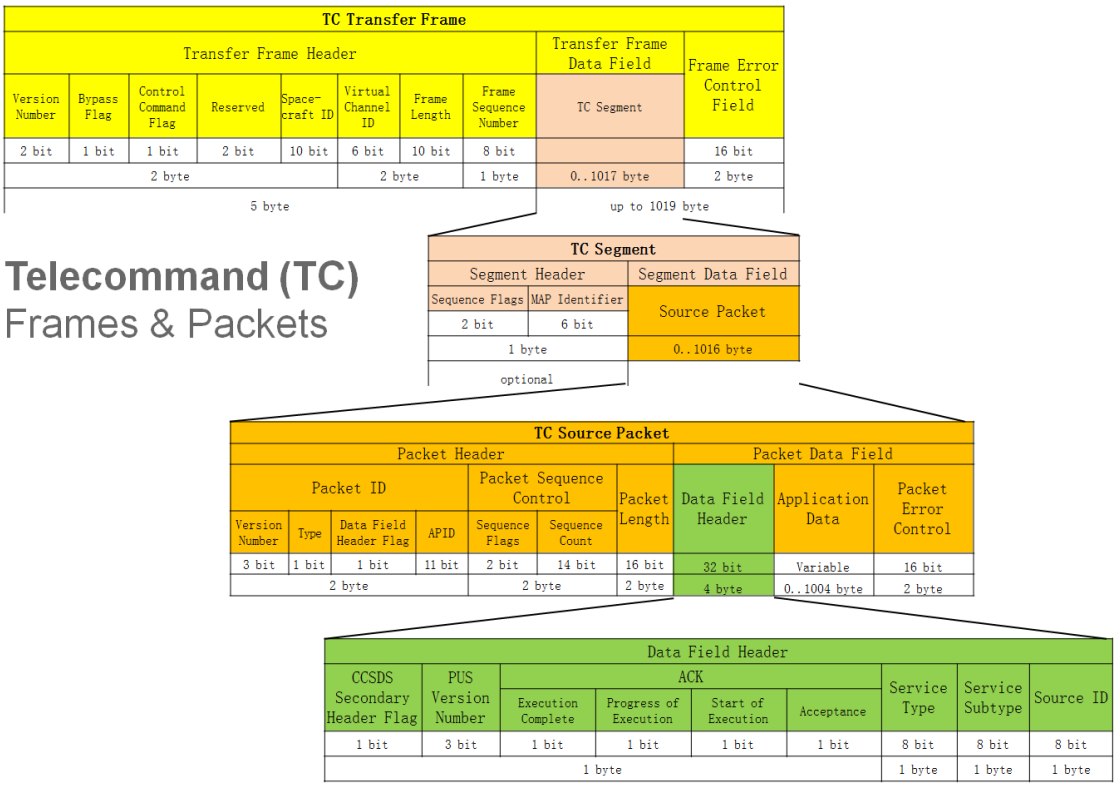


Abbildung 11: Mögliche Hierarchie bei der Übertragung von Telekommandos
Quelle: Interne Präsentation des DLR (nicht öffentlich zugänglich).

Abbildung 11 zeigt die mögliche Einbettung und Schichtung der Informationen von Telekommandos.

Telemetry Packets

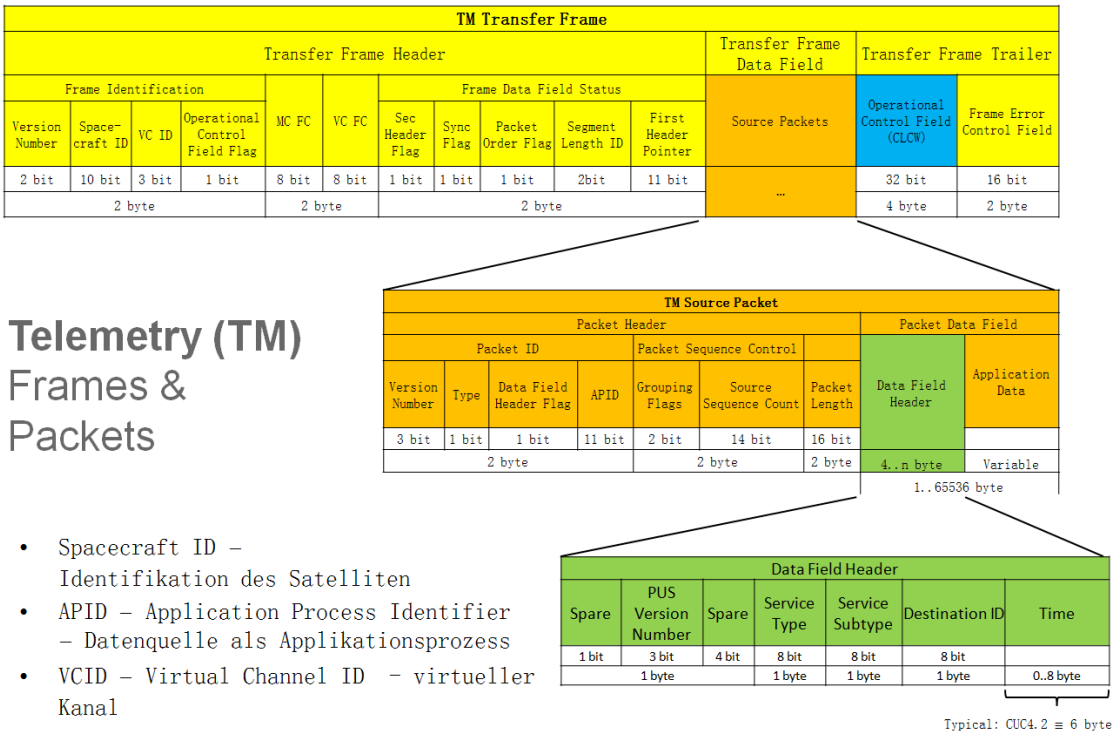
Wie ein Telekommandopaket ist ein Telemetriepaket ein spezielles Space Packet (packet type = 0) mit der Aufgabe, Nutzdaten zu übertragen und Fernmessungen zu ermöglichen. Zu diesen Nutzdaten gehören die bereits erwähnten Acknowledgements oder Ergenbnisse einer wissenschaftlichen Messeinrichtung.

TM packet PUS version number	spacecraft time reference status	message type ID		message type counter	destination ID	time	spare
		service type ID	message subtype ID				
enumerated (4 bits)	enumerated (4 bits)	enumerated (8 bits)	enumerated (8 bits)	unsigned integer (16 bits)	enumerated (16 bits)	absolute time	fixed-size bit-string
							optional

Abbildung 12: Secondary Header eines Telemetriepakets
Quelle: [SS16], S. 439.

Abbildung 12 zeigt den Secondary Header eines Telemetriepaketes. Wie bei einem Telekommando besteht hier die Unterteilung nach Service und Subservice. Dazu gibt es einen weiteren Zähler *message type counter*, der innerhalb einzelner Messreihen eine weitere und genauere (für jeden Service Subtype spezifisch und mit einem zwei Bit größeren Zahlenraum) Sequenznummer angibt.

Abbildung 13 beschreibt - ähnlich zu Abbildung 11 - die Einbettung von Telemetrie in ein



Telemetry (TM)
Frames & Packets

- Spacecraft ID – Identifikation des Satelliten
- APID – Application Process Identifier – Datenquelle als Applikationsprozess
- VCID – Virtual Channel ID – virtueller Kanal

Abbildung 13: Mögliche Hierarchie bei der Übertragung von Telemetrie
Quelle: Interne Präsentation des DLR (nicht öffentlich zugänglich).

weiteres Paket: dem TM Transfer Frame. Zusätzlich ist die Ausgestaltung der Zeitinformation in der Form CUC4.2 (CCSDS Unsegmented Time Code (CUC) mit vier Byte Genauigkeit für die ganzen Sekunden und zwei Byte Genauigkeit für die Sekundenbruchteile) innerhalb des Data Field Header erkennbar.
Analog zu TC Source Packets werden innerhalb dieser Arbeit UDP-Pakete genutzt, um TM Source Pakets für eine Datenübertragung zu kapseln.

2.4 Standard-Services

Im Folgenden werden die in [SS16] definierten Standard-Services beschrieben, die in vielen Missionen Anwendung finden und daher vom ECSS festgelegt wurden.

Service Type	Bezeichnung	Beschreibung
ST 01	Reqest Verification	Sendet Acknowledgements zum Status von Empfang und Ausführung.
ST 02	Device Access	Erlaubt Zugriff auf On-Board-Geräte.
ST 03	Housekeeping	Dient zur Abfrage und Übertragung von zu überwachenden Betriebsparametern.
ST 04	Parameter Statistics Reporting	Ermöglicht es, Zusammenfassungen (Mittelwert, Minimum, Maximum, Standardabweichung) einzelner Parameter anzufordern.
ST 05	Event Reporting	Erlaubt das Abfragen bestimmter Daten, die nicht von anderen Services betrachtet werden, etwa die Ergebnisse von Testläufen.
ST 06	Memory Management	Dient zur Speicherverwaltung, um etwa Dumps von RAM oder dauerhaften Speichern anzufertigen oder den Speicher mit definierten Inhalten zu beschreiben.
ST 07	Reserved	

ST 08	Function Management	Ermöglicht die Kontrolle missionsspezifischer Funktionen, etwa zur Kontrolle eines Nutzlasteninstrumentes. Dient als Alternative zu missionsspezifischen Services und zur Bewahrung von Abwärtskompatibilität.
ST09	Time Management	Dient dem Abgleich der Zeiten zwischen Ground Segment und Space Segment.
ST 10	Reserved	
ST 11	Time-Based Scheduling	Ermöglicht die zeitgesteuerte Ausführung von Telekommandos.
ST 12	On-Board-Monitoring	Erlaubt das Überwachen von Parametern und Auslösen von Events, wenn einer dieser Parameter außerhalb eines definierten Bereiches liegt.
ST 13	Large Packet Transfer	Dient der Übertragung von Paketen, deren Größe außerhalb der Grenzen liegen, die für Telemetrie- und Telekommandopakete gilt, indem sie auf mehrere kleinere Pakete verteilt werden.
ST 14	Real-Time Forwarding Control	Ermöglicht die Übertragung ausgewählter Daten innerhalb eines festgelegten Zeitfensters (Echtzeit) unter Benutzung eines speziellen Telemetriekanals.
ST 15	On-Board Storage and Retrieval	Erlaubt das Ablegen von Paketen in Packet Stores sowie dessen Management und den Abruf der darin enthaltenen Informationen.
ST 16	Reserved	
ST 17	Test	Dient dem Ausführen von Testroutinen und dem Abrufen der entsprechenden Resultate.

ST 18	On-Board Control Procedure	Ermöglicht die Ausführung und den Austausch von Prozeduren, die nicht in der Missionssoftware gespeichert sind (vgl.[SS10]).
ST 19	Event-Action	Erlaubt die Ausführung von Aktionen beim Auftreten spezifischer Events.
ST 20	On-Board Parameter Management	Dient der Zuordnung von Parametern (identifiziert anhand ihrer ID) und dem Speicherbereich, in dem diese Parameter gespeichert sind.
ST 21	Request Sequencing	Ermöglicht die Abfolge von Anfragen, zwischen denen eine definierte Zeitspanne liegt.
ST 22	Position-Based Scheduling	Erlaubt die positionsspezifische (etwa bei Erreichen des erwünschten Orbits) Ausführung von Befehlen.
ST 23	File Management	Dient der Verwaltung eines On-Board-Dateisystems.

2.5 OUTPOST

Die „Open modUlar sofTware PlatfOrm for SpacecrafT“ (OUTPOST) dient als Softwarebibliothek zur Implementierung von Raumfahrtanwendungen. Sie basiert auf der Entwicklung der Softwareplattform zur Nutzung des *Compact On-Board Computers* `libcobc`, einer bezüglich Prozessorleistung, Speichergröße und Tauglichkeitsstufe skalierbaren Hardwareplattform. Sie stellt zum Einen eine Abstraktion von der Hardware sowie vom Betriebssystem bereit und bietet zum Anderen Softwarekomponenten, etwa zum Zeitmanagement oder der systeminternen Kommunikation. Sie dient somit als Basis der Missions-Software-Entwicklung (vgl. [Dan]), indem missionsspezifische Anwendungen und Services hinzugefügt werden. Zentrale Bestandteile werden open-source bereitgestellt und als OUTPOST-CORE[8] bezeichnet. Weitere Bestandteile, etwa zur Kommunikation entsprechend dem Packet Utilization Standard unterliegen der Exportkontrolle und sind closed-source. Sie sind daher ausschließlich DLR-intern nutzbar und im Paket OUTPOST-SATELLITE enthalten.

[8] `https://github.com/DLR-RY/outpost-core`

2.5.1 OUTPOST-CORE

OUTPOST-CORE enthält die frei verfügbaren Komponenten von OUTPOST:

- **utils**
  Diese Komponente wird als „useful bits and pieces“ ([Gre]) bezeichnet und enthält verschiedene vielseitig einsetzbare Komponenten, darunter CRC-Funktionen, Listenimplementierungen, Serialisierungsfunktionen und Methoden zum bitweisen Zugriff auf Daten.

- **time**
  Hier sind Komponenten zum Zeitmanagement vorhanden, etwa Konvertierungen zwischen UNIX-Zeit und GPS-Zeit.

- **rtos**
  Diese Zwischenschicht ermöglicht die Benutzung von OUTPOST auf den Betriebssystemen RTEMS, FreeRTOS sowie allen POSIX-kompatiblen Systemen.

- **hal**
  Mithilfe dieser Komponente wird eine Abstraktionsschicht für Kommunikationsschnittstellen bereitgestellt.

- **smpc**
  Durch den „Simple Message Passing Channel“ können innerhalb eines Adressraumes Botschaften nach dem publish/subscribe-Modell verteilt werden.

- **l3test**
  Um als Teil eines Unit-Tests ein Lua-Skript laufen zu lassen, kann diese Komponente verwendet werden.

2.5.2 OUTPOST-Satellite

Im Gegensatz zu OUTPOST-Core ist dieser Bestandteil nicht open-source und ist daher nicht auf Github veröffentlicht. Er enthält die folgenden Komponenten:

- **spp**
  Diese Komponente ist eine Implementierung des Space Packet Protocolls der CCSDS.

- **pus**
  Dieses Modul beinhaltet eine partielle Implementierung des Packet Utilization Standards sowie der darin beschriebenen Standard-Services.

- **l3test**
  Diese Komponente kann verwendet werden, um zu Testzwecken durch ein Lua-Skript SPP- und PUS-Pakete zu generieren.

- **log**
  Um Logging-Nachrichten innerhalb einer begrenzten Bandbreite versenden zu können, wird dieses Modul verwendet.

- **cdh**
  Dieses Modul bietet Unterstützung bei häufigen Aufgaben des Command & Data Handlings.

2.5.3 Service-Interface

Von OUTPOST wird eine Schnittstelle angeboten, die genutzt werden kann, um missionsspezifische Services anzulegen. Hierzu muss entsprechend dem in Unterunterabschnitt 2.3.1 beschriebenen Service-Konzept zunächst eine Application angelegt werden, der ein Service übergeben wird. Die Application sorgt in ihrem (geerbten) Konstruktor dafür, dass eine Subscription an den entsprechenden Telecommand-Dispatcher übergeben wird, sodass eingehende Telekommandos an die einzelnen Services weitergereicht werden.
Im Serviceinterface ist festgelegt, dass ein Service die Methoden `executeCommand` und `sendDataToDownload` implementiert. Der Methode `executeCommand` wird vom Dispatcher ein Telekommando übergeben, welches ausgeführt werden soll. Da bereits nach APID und Service Type aufgeteilt wurde, bleibt noch die Aufteilung nach Service Subtype, was beispielsweise durch eine `switch-case`-Verzweigung möglich ist.

```
bool Service::executeCommand(const outpost::pus::TelecommandReader& command){

  switch (command.getServiceSubType()) {

    case 1: {
      //Send Acknowledgement
      if (!telecommandIsValid(command))
        telecommandVerification().rejected(command);
      else {
        telecommandVerification().accepted(command);

        bool success = true;

        //do something, e.g. movement
        if (!movementWasSuccessful())
```

```
          success = false;

        //Send Acknowledgment: Execution in progress
        if (success)
          telecommandVerification().executionInProgress(command);
        else
          telecommandVerification().executionProgressFailure(command)

        //do something, e.g. measurement
        uint8_t data[8];
        if (!measurementWasSuccessful(data))
          success = false;

        //Send Data to GUI
        sendDataToDownload(mServiceType, 128, 1, data, 8);

        //Send Acknowledgement: Execution completed
        if (success)
          telecommandVerification().executionCompleted(command);
        else
          telecommandVerification().executionCompletionFailed(command);
      }
    }
  ...
  }
}
```

Listing 1: Beispielimplementierung der Methode `executeCommand`

Listing 1 zeigt einen Ausschnitt der Methode `executeCommand`, in der ein Telekommando ausgeführt wird. Es wird an dieser Stelle nach Service Subtype unterschieden.
Zu Beginn der Ausführung eines Telekommandos nach Separierung anhand des Service Subtypes wird dem CC mitgeteilt, ob das Telekommando akzeptiert wurde. Anschließend folgt die eigentliche Ausführung des Befehls sowie Acknowledgements, welche die momentane Ausführung und Beendigung des Befehls anzeigen.
Es wird ein Telemetriepaket an das CC zurückgesendet, welches Messdaten enthält.
Erkennbar ist, dass das Telekommando nur ausgeführt wird, wenn es gültig ist (und beispielsweise die ApplicationData eine bestimmte Form hat). Es folgen eine Bewegung sowie eine Messung, die auch ausgeführt wird, wenn die Bewegung nicht erfolgreich war. Tatsächliche Missionsszenarien können sich hier unterscheiden. Das zurückgesendete Telemetriepaket hat die APID der beinhaltenden Application, den im Programmcode in der Variablen `mServiceType` gespeicherten Service Type, den Service Subtype 128 sowie 8 Bytes an

Nutzdaten.

2.6 Controller Area Network (CAN)

Das Controller Area Network (CAN) ist eine Spezifikation für die Datenübertragung sowohl auf physikalischer Ebene (Mithilfe welcher physikalischen Größe, etwa einem Spannungspegel, werden Daten kodiert?) als auch auf Protokollebene (Wie werden Daten verpackt? Wieviele Bits sind für ein Datenwort nötig und wie wird es durch Start- und Stoppbits von anderen Paketen abgegrenzt?). CAN wurde 1983 von Bosch für den Einsatz in Kraftfahrzeugen entwickelt und 1986 vorgestellt (vgl. [Gmb16]). Aufgrund der Einfachheit und gleichzeitig Robustheit von CAN wird es inzwischen auch in anderen Bereichen, wie der Industrieautomation sowie Luft- und Raumfahrt, eingesetzt (vgl. [Tro13]). Es ermöglicht auf kurzen Strecken bis 40m maximal 1 MBaud (vgl. [SM98]).

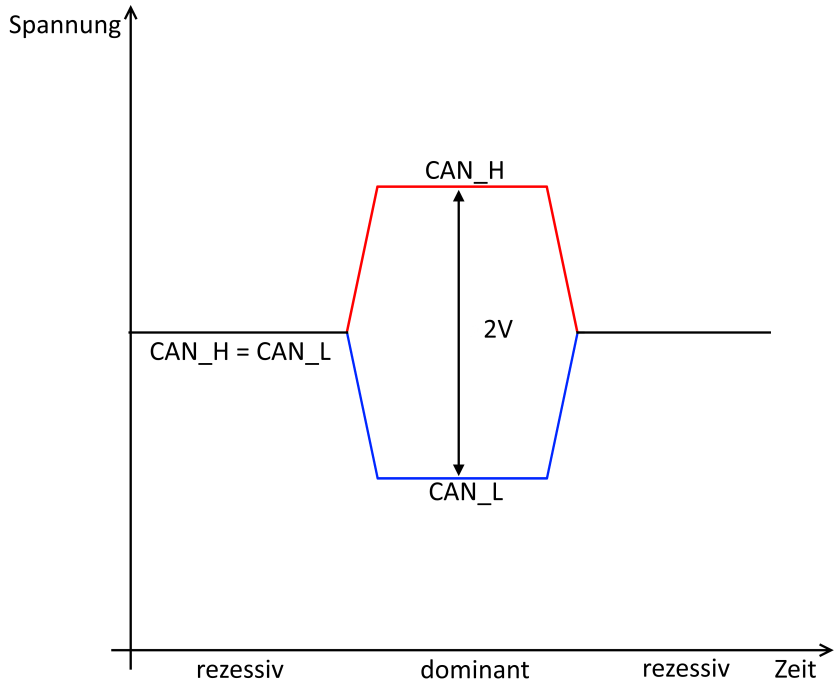


Abbildung 14: Abbildung der CAN-Signale HIGH und LOW auf die physikalische Größe Spannung(sdifferenz)
Quelle: Eigenes Bild, in Anlehnung an [Plu13].

Leitungscode

Auf der physikalischen Seite werden Dateneinheiten durch eine Spannungsdifferenz zwischen

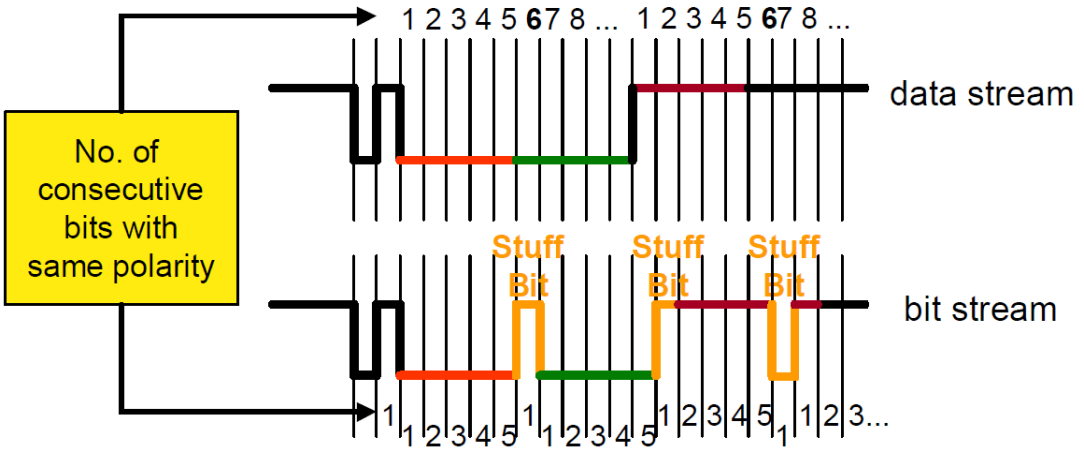


Abbildung 15: Bit-Stuffing bei aufeinanderfolgenden gleichen Pegeln
Quelle: [SM98], S. 33.

zwei Leitungen kodiert. Abbildung 14 zeigt die Kodierung eines HIGH- sowie eines LOW-Pegels. Beim HIGH-Pegel (logisch 0, vgl. [Gmb16]) beträgt die Spannungsdifferenz zwischen den Leitungen CAN_H und CAN_L 2V, bei einem LOW-Pegel (logisch 1) ist die Spannungsdifferenz annähernd 0 V. Aufgrund der Tatsache, dass beide Leitungen durch einen Widerstand verbunden sind, haben beide Leitungen ohne ein anliegendes Signal (annähernd) die gleiche Spannung. Dieser Zustand wird (aufgrund der nicht angelegten Spannung) als rezessiver Zustand bezeichnet. Entsprechend ist beim Anlegen einer Spannung durch einen Busteilnehmer vom dominanten Zustand die Rede, da ein dominanter Pegel eines Busteilnehmers den rezessiven Pegel „überschreibt“ (vgl. [Gmb16], [SM98]).

Taktrückgewinnung, Bit-Stuffing

Die Datenpakete werden seriell auf zwei Leitungen übertragen. Da es keine spezielle Leitung für den Takt gibt, muss dieser aus den übertragenen Signalpegeln rekonstruiert werden. Dies gelingt durch Rekonstruktion aus den übertragenen Signalpegeln. Die CAN-Spezifikation verbietet Übertragungen von mehr als fünf zusammenhängenden gleichen Signalpegeln. Zu diesem Zweck werden so genannte Stuff Bits (siehe Abbildung 15) eingefügt, welche die Anzahl der übertragenen Signalpegel (und damit auch die Übertragungsdauer) abhängig vom Nachrichteninhalt erhöht (vgl. [SM98]). Diese Stuff Bits müssen vom CAN-Controller wieder herausgerechnet werden.

Robustheit

Das CAN-Interface gilt als robuste Schnittstelle zur Datenübertragung. Dies liegt zum Einen

an der Fehlererkennung per CRC und zum Anderen an der Fähigkeit der Busteilnehmer, eigene Fehler zu erkennen und sich selbst in einen Fehlerzustand zu setzen. Dies kann dazu führen, dass ein Teilnehmer nicht mehr an der Kommunikation auf dem Bus teilnehmen darf. Fehler eines Busteilnehmers kann dieser selbst erkennen, indem er Error Frames anderer Busteilnehmer (zur Anzeige, dass ein empfangenes Datenpaket fehlerhaft ist) empfängt und entsprechend interpretiert. Ein Acknowledgement auf ein korrekt empfangenes Datenpaket wird dadurch umgesetzt, dass ein bestimmter Signalpegel nach dem CRC vom sendenden Teilnehmer rezessiv belassen wird. Zieht mindestens ein Busteilnehmer dieses Signal auf dominant, weiß der sendende Busteilnehmer über die (im Sinne des Protokolls) wahrscheinlich korrekte Übertragung (vgl. [Gmb16]). Prinzipbedingt kann der Fehlerfall eintreten, dass ein Busteilnehmer den Bus dauerhaft auf HIGH zieht. Dies führt dazu, dass der Bus für keinen Busteilnehmer mehr nutzbar ist.

Priorisierung

Eine Priorisierung der Nachrichten auf dem Bus geschieht, indem ein sendender Busteilnehmer beim Senden gleichzeitig auch den Signalpegel erfasst. Senden zwei Controller einen HIGH-Pegel, können sie dieses jedoch nicht erkennen (was jedoch nicht relevant ist, da beide Nachrichten bis zu diesem Zeitpunkt gleich sind). Sendet ein Teilnehmer einen rezessiven Pegel, liest aber bei der Auswertung des tatsächlich vorhandenen Signals einen dominanten, weiß er, dass zu diesem Zeitpunkt eine Nachricht mit höherer Priorität auf den Bus geschrieben wird und bricht seine Übertragung ab, sodass sie - nachdem das höher priorisierte Paket fertig gesendet wurde - zu einem späteren Zeitpunkt erneut gestartet wird. Aufgrund der Tatsache, dass ein Bit mit dem Wert 1 durch einen LOW-Pegel übertragen wird, erhält der Teilnehmer mit der kleineren CAN-ID die höhere Priorität (wobei die CAN-ID 0 der höchsten Priorität entspricht) (vgl. [Gmb18b]).

Frame-Typen

Das CAN-Bus-Protokoll bietet verschiedene Frame-Typen, die sich in ihrer Funktion und ihrer Adresslänge unterscheiden. Standard-Frames besitzen eine 11-bittige Adresse, Extended Frames weisen mithilfe eines gesetzten Bits „Extended Identifier Bit“ auf eine Adresslänge von 29 Bit hin (vgl. [Gmb18a]).

Ein Daten-Frame enthält 0-8 Bytes Daten und dient der Datenübertragung zwischen CAN-Knoten. Mithilfe eines Remote Frames (Das Bit „Remote Transmission Request“ ist gesetzt und das CAN-Frame enthält keine Nutzdaten.) kann ein Busteilnehmer einen anderen dazu auffordern, Daten zu senden. Ein Error-Frame dient dazu, andere Busteilnehmer über einen entdeckten Fehler bei der Datenübertragung zu informieren. Je nach Zustand des fehlermel-

denden Knotens können sechs dominante Bits (im Zustand „error-active) gesendet werden[9]. Senden mehrere Busteilnehmer ein Error-Frame, können bis zu 12 dominante Bits hintereinander auf den Bus geschrieben werden. Im Zustand „error-passive“ werden mehrere rezessive Bits auf den Bus geschrieben. Ein Overload Frame gleicht einem Active-Error-Frame, wird jedoch nicht während der Übertragung eines Frames gesendet, sondern zwischen zweien. Es dient der Anzeige, dass weitere ankommende Daten nicht verarbeitet werden können. Eine weitere ankommende Nachricht kann um die Zeitspanne von maximal zwei Overload Frames verzögert werden (vgl. [SM98]).

[9] Aufgrund des Bitstuffings kann diese Sequenz bei einer normalen Übertragung nicht vorkommen.

3 Protokoll

Das *Protokoll zur Service-Discovery für Netze heterogener Sensoreinheiten* stellt den Hauptteil dieser Arbeit dar. In diesem Abschnitt werden die Anforderungen an das Protokoll sowie die sich daraus ergebende Spezifikation dargestellt. Ebenso wird eine Erweiterung zur Nutzung von Aktoren vorgestellt. Um das Protokoll zu evaluieren, wurde eine Beispielanwendung implementiert. Deren Komponenten werden in Abschnitt 4: GUI, Abschnitt 5: Remote Unit, Abschnitt 6: Sensorschnittstelle und Abschnitt 7: CAN-I^{2}C-Bridge beschrieben. Die Ergebnisse der Evaluation finden sich in Abschnitt 8.

3.1 Anforderungen an das Protokoll

Im Folgenden wird dargelegt, welche Anforderungen an das Protokoll gestellt werden, um die gewünschte Service Discovery zu ermöglichen.

3.1.1 Zugrundeliegende Problemstellung

Nach der ROBEX-Mission wurden Ideen gesammelt, wie das Projekt fortzuführen wäre. Die von mir verfasste Ideensammlung hierzu findet sich in Unterabschnitt 12.2: Robex *Reloaded*. Einer der hier vorgestellten Vorschläge war die Entwicklung einer Service-Discovery zur Bestimmung der Fähigkeiten einer Remote Unit und die damit verbundene dynamische Anpassung der Anzeige innerhalb der Benutzeroberfläche. Diese erscheint sinnvoll, weil bei einer Erweiterung der Komponenten um Funktionen - was innerhalb der ROBEX-Mission mehrfach passierte - eine entsprechende Implementierung zweifach vorgenommen werden musste. Zum Einen musste die entsprechende Funktionalität in der Remote Unit selbst verankert werden und zum Anderen musste daraufhin die GUI angepasst werden, um entsprechend neue Anzeigeelemente sowie ggf. die Logik zur Umwandlung der Telemetrie in lesbare Sensorwerte zu implementieren.

3.1.2 Entwicklung der Anforderungsspezifikation

Aus der eben genannten Problemstellung ergab sich die Idee, dieses nur noch auf einer Seite (Remote Unit) durchzuführen, sodass die GUI die Anzeigeelemente selbst generiert. Hierzu ist es erforderlich, dass die Remote Unit die GUI über angeschlossene Sensoren informiert. Um dieses zu ermöglichen, muss das Protokoll folgende Eigenschaften ermöglichen:

- Es soll innerhalb jeder Messiteration feststellen, welche Sensoren vorhanden sind.

- Eine Änderung der Messkonfiguration (angeschlossene Sensoren) soll sich nach Empfang eines Telemetriepaketes in einer entsprechenden Anzeige wiederspiegeln.
- Gemessene Werte sollen korrekt angezeigt und den messenden Sensoren zugeordnet werden können.

3.1.3 Erweiterung des Service Discovery-Protokolls um Aktoren

Das ursprünglich als Service-Discovery-Protokoll für heterogene **Sensor**netzwerke konzipierte Protokoll bietet Möglichkeiten zur Erweiterung. Neben den im Ausblick in Unterabschnitt 11.1: Verallgemeinerte Abfrage und Unterabschnitt 11.2: Übertragungsskript genannten Möglichkeiten kann das Service-Discovery-Protokoll auch dafür genutzt werden, eine Service Discovery angeschlossener Aktoren durchzuführen, diese in der GUI anzuzeigen und somit Bedienelemente anzubieten.

3.2 Datenstrukturen

Mithilfe des Protokolls wird innerhalb jeder Messiteration übertragen, welche Sensoren angeschlossen sind. Dies erfolgt, indem jedem Sensormesswert innerhalb eines Telemetriepaketes die Information über den Sensortyp vorangestellt wird. Eine Auswertung in der GUI kann somit zum Einen das richtige Anzeigeelement für den Sensormesswert feststellen und zum Anderen die Daten hierin präsentieren.

Sensor Type ID	Sensor Data

Tabelle 2: Datenstruktur mit Sensortyp und -daten

Handelt es sich bei dem angeschlossenen Gerät nicht um einen Sensor, sondern um einen Aktor, ist - wie in Tabelle 3 beschrieben - nur die Übertragung des Aktortyps notwendig.

Actor Type ID

Tabelle 3: Datenstruktur mit Aktortyp

Weiterhin wurde eine Erweiterung des Protokolls vorgenommen, um die Problemstellung zu umgehen, dass nicht mehrere Sensoren eines Typs gleichzeitig genutzt werden können. Hierzu

wird die in Abschnitt 7 beschriebene CAN-I^{2}C-Bridge eingesetzt, die es erforderlich macht, Daten, wie in Tabelle 4 beschrieben, zu übertragen.

Sensor Type ID	Bridge ID	Sensor Data

Tabelle 4: Datenstruktur bei einem per CAN-I^{2}C-Bridge angeschlossenen Sensor

Neben der *Sensor Type ID*, welche Auskunft darüber gibt, um welchen per CAN-I^{2}C-Bridge angeschlossenen Sensor es sich handelt, wird die ID der CAN-I^{2}C-Bridge übertragen, die es ermöglicht, gleichartige Sensoren an CAN-I^{2}C-Bridges (mit unterschiedlichen, vom Benutzer festgelegten IDs) zu unterscheiden.
Analog zur Datenstruktur aus Tabelle 3 wird bei einem angeschlossenen Aktor nur dessen Typ sowie die CAN-I^{2}C-Bridge-spezifische ID übertragen, was in Tabelle 5 dargestellt ist.

Actor Type ID	Bridge ID

Tabelle 5: Datenstruktur bei einem per CAN-I^{2}C-Bridge angeschlossenen Aktor

Die in Tabelle 2, Tabelle 3, Tabelle 4 und Tabelle 5 beschriebenen Datenstrukturen werden als Service Discovery Entries bezeichnet und zusammengefasst in Tabelle 6 dargestellt. Ein von der Remote Unit gesendetes Telemetriepaket hat also den in Tabelle 7 beschriebenen Aufbau, bestehend aus Telemetrieheader und beliebig vielen Service Discovery Entries gefundener Sensoren und Aktoren, welche per I^{2}C oder CAN angeschlossen sind.

Type ID	Bridge ID	Sensor Data
	optional	optional

Tabelle 6: Aufbau eines Service Discovery Entries der Service Discovery

Telemetry Header	Service Discovery Entry
	repeated n times

Tabelle 7: Aufbau eines Telemetriepaketes der Service Discovery

Tabelle 7 zeigt den Aufbau eines Paketes, welches von der Remote Unit an die GUI übertragen wird und Informationen über Sensor- und Aktortypen, ggf. CAN-I^{2}C-Bridge-IDs und ggf. Sensorwerte enthält. Die Informationen über die verwendeten Sensor- / Aktortypen werden in jedem Paket mitgesendet. Dies stellt zwar einen gewissen Overhead dar, umgeht jedoch Probleme möglicher Paketverluste oder die Maßnahmen zur Verhinderung desselben. Die Zahl der tatsächlich nutzbaren Sensoren ist implementierungsspezifisch durch die maximale Größe eines Telemetriepaketes begrenzt. Dieser Aspekt wird im Ausblick im Unterabschnitt 11.4: Skalierbarkeit diskutiert.

3.3 Ablauf des Protokolls

Nachdem beschrieben wurde, wie die Datenstrukturen zur Identifizierung von Sensoren, Aktoren und CAN-I^{2}C-Bridges aussehen, wird im Folgenden herausgestellt, wie die Datenübertragung abläuft, wenn Geräte an der Remote Unit angeschlossen sind.

Wie in Abbildung 16 erkennbar ist, wird von der GUI ein Telekommando gesendet, welches den Sensor-Thread startet. Dieser durchläuft in regelmäßigen Abständen (nach einem Durchlauf beginnt eine Wartezeit von einer Sekunde) alle I^{2}C-Adressen auf der Suche nach Sensoren, welche im Falle eines Fundes anhand ihrer I^{2}C-Adresse identifiziert und abgefragt werden.
Nach der Abfrage aller Sensorwerte werden diese zusammen mit der Information über die gefundenen Sensortypen in einem Telemetriepaket an die GUI übertragen.

Abbildung 17 zeigt die Kommunikation zwischen GUI, Remote Unit, CAN-I^{2}C-Bridge und Sensoren. Nach dem Start des Sensor-Threads wird eine Nachricht an alle CAN-Teilnehmer mit ID `0xFF` und leerem Nutzdatenfeld gesendet, welche signalisiert, dass die Remote Unit Daten von allen angeschlossenen Sensoren (bzw. den zwischengeschalteten CAN-I^{2}C-Bridges) erwartet. Diese wird (wie alle CAN-Nachrichten) als Broadcast auf den Bus geschrieben und von allen verbundenen CAN-I^{2}C-Bridges empfangen. Die CAN-I^{2}C-Bridges finden anhand der I^{2}C-Adresse des angeschlossenen Gerätes heraus, um welchen Typ es sich handelt (Ablauf siehe Listing 6 bzw. Listing 4). Bei Sensoren wird nun der Sensorwert ermittelt. Für Aktoren und Sensoren gibt es eindeutige Typen-Kennungen, die innerhalb dieser Master Thesis festgelegt wurden (siehe Unterabschnitt 12.3: Festgelegte CAN-ID-Offsets für Sensoren und Aktoren). Die CAN-ID der CAN-I^{2}C-Bridge ergibt sich aus Sensortyp-Kennung und der vom Benutzer einstellbaren CAN-I^{2}C-Bridge-ID (siehe Tabelle 8).

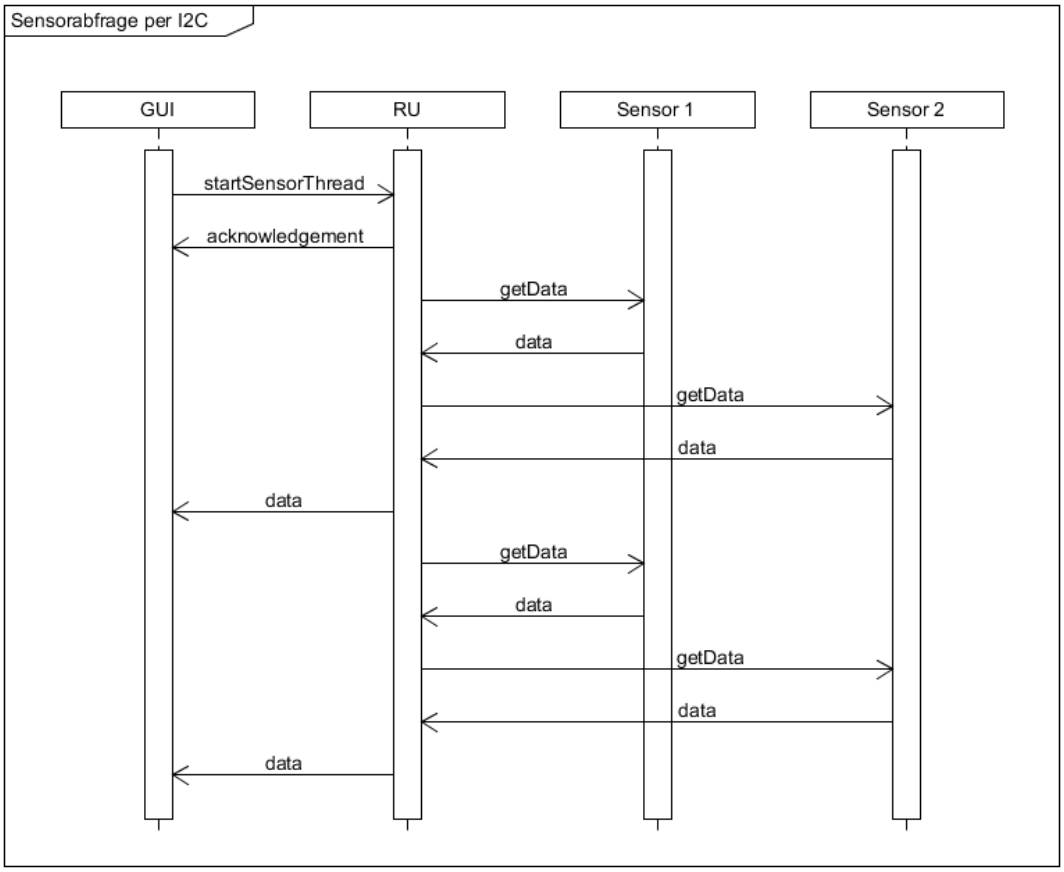


Abbildung 16: Start des Sensorthreads, Abfrage von mehreren I^2C-Sensoren

3.4 Service Discovery auf Netzebene

Die GUI soll mithilfe der Service Discovery auf Netzebene alle Remote Units innerhalb der Domäne *lokales Subnetz* finden und eine entsprechende Anzeige (Widget) generieren. Hierzu wird festgelegt, dass alle gefundenen Remote Units genau diejenigen sind, die mit zwei Acknowledgements (*accepted* und *executionCompleted*) auf ein entsprechendes Anfragepaket („DiscoveryCommand“) antworten. Somit wird festgestellt, dass es sich um eine Remote Unit handelt, die die Fähigkeit zur Service Discovery besitzt.
Dieses wird umgesetzt, indem ein Anfragepaket per Broadcast an alle Netzgeräte im lokalen Subnetz gesendet wird und empfangende Remote Units mit einem Acknowledgement antworten. Ein Anfragepaket ist ein Telekommandopaket mit APID 1, dem Service Type 128 und Service Subtype 128. Abbildung 18 zeigt den Ablauf dieses Prozesses.
Nachdem die GUI Acknowledgements von den Remote Units erhalten hat, soll sie die gefun-

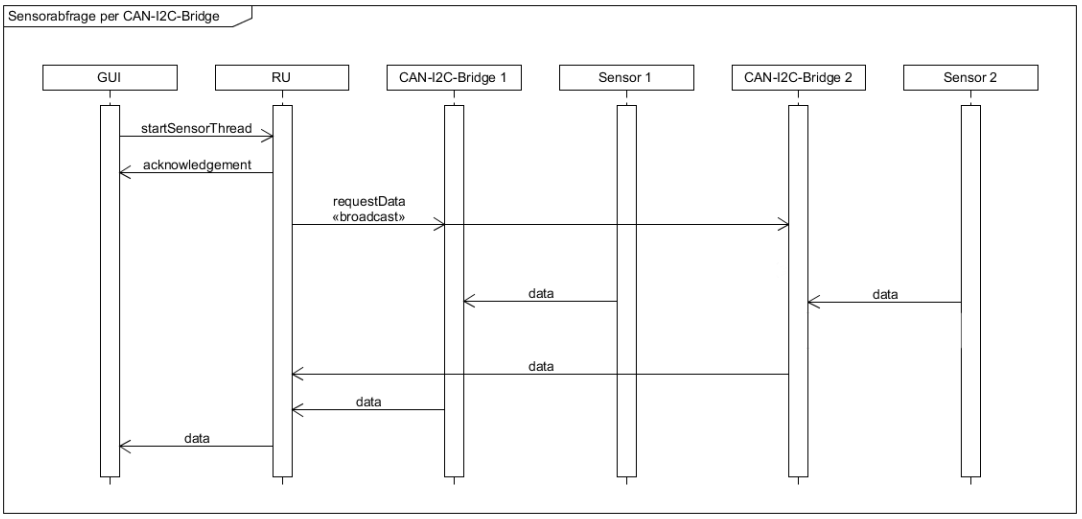


Abbildung 17: Ablauf der Abfrage von per CAN-I^2C-Bridge angeschlossenen Sensoren

dene Remote Unit in eine Liste gefundener Remote Units eintragen und ein entsprechendes Widget generieren.

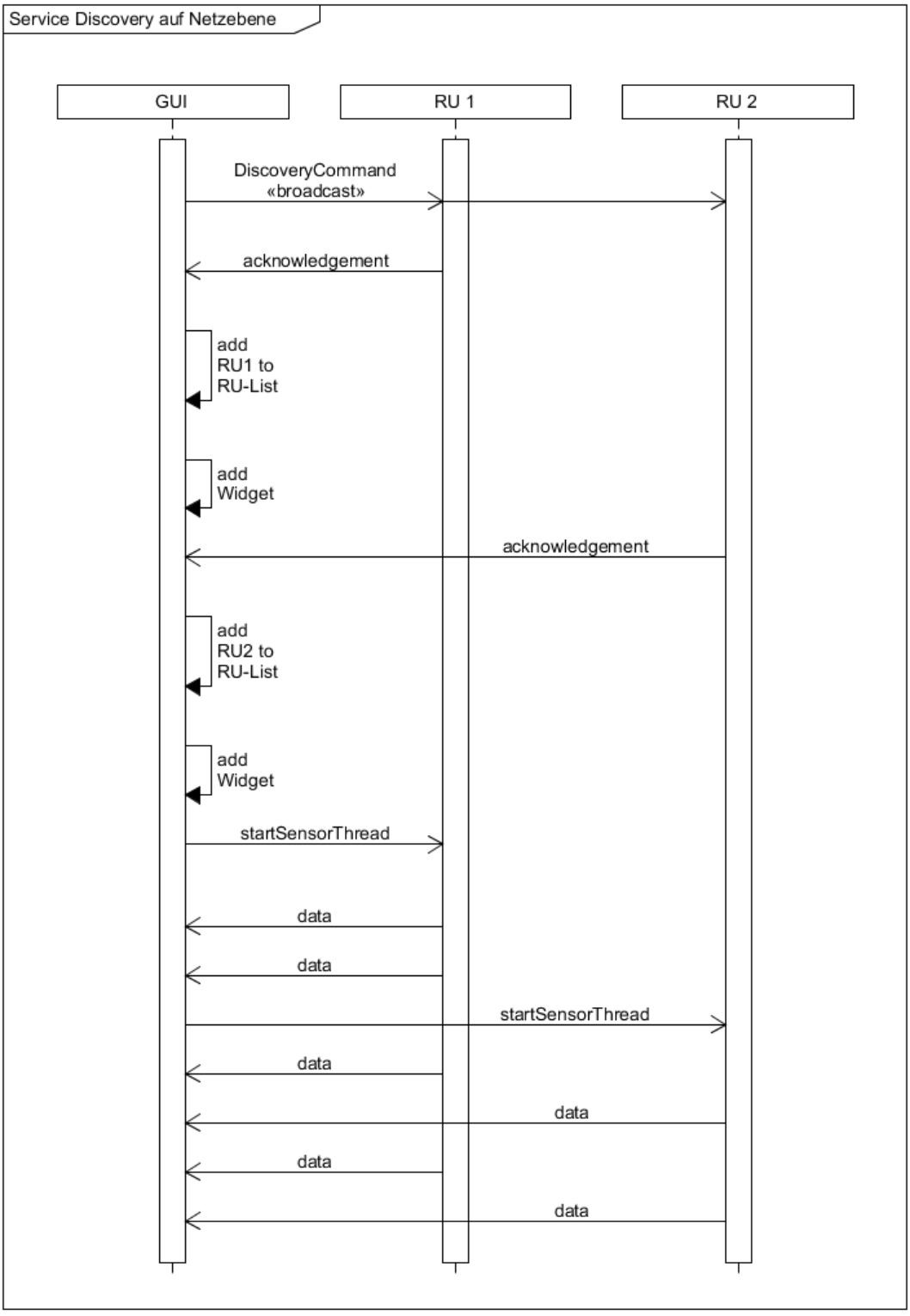


Abbildung 18: Service-Discovery auf Netzebene

4 GUI

Die GUI basiert auf der TM/TC-GUI des DLR. Sie dient der Kommandierung von Remote Units und der Anzeige von gemessenen Daten. Die grafische Oberfläche wurde mit dem Open-Source-Toolkit Qt 5 realisiert. Darüber hinaus wurde ihr Funktionsumfang erweitert, um der Zielsetzung dieser Master Thesis zu entsprechen.
Um die Paketgenerierung und -dekodierung zu gewährleisten, nutzt sie Funktionen der Bibliothek OUTPOST.

4.1 Grundlagen

In diesem Abschnitt werden die zur Implementierung bzw. Anpassung der GUI benötigten technischen Grundlagen dargelegt. Zunächst werden die genutzten Komponenten der Bibliothek OUTPOST und die TM/TC-GUI (auf der die GUI aufbaut, die in dieser Arbeit implementiert wurde) dargestellt.

4.1.1 OUTPOST

Die GUI wurde mithilfe der Bibliothek OUTPOST implementiert. Diese bietet verschiedene Schnittstellen und Tools, um den Entwickler zu unterstützen. Als besonders relevant sind hierbei die Klassen `outpost::pus::TelecommandReader` und `outpost::pus::TelecommandWriter` zu nennen, die Telekommandos kapseln. Für Telemetrie-Pakete gibt es analog die Klassen `outpost::pus::TelemetryReader` und `outpost::pus::TelemetryWriter`. Weiterhin bietet die Klasse `outpost::BoundedArray<T>` die Möglichkeit, Daten in einem Puffer zu speichern.

BoundedArray
Um das Problem des nichtdeterministischen Speichernutzungsverhaltens bei dynamischer Allozierung von Speicher während einer Mission, inbesondere wenn diese durch externe (gemessene oder gesteuerte) Ereignisse getriggert wird, zu umgehen, bietet es sich an, die Speicherbelegung bereits während der Programmierung festzulegen. Somit können von vornherein Aussagen über die Speichernutzung während der Mission getroffen werden. Um dennoch flexibel auf Speicher zugreifen zu können, bieten sich intuitiv Arrays einfacher Datentypen, etwa `uint8_t`, an. Da der Zugriff hierauf jedoch nicht besonders komfortabel ist und nicht sichergestellt werden kann, dass nicht über Arraygrenzen hinaus auf den Speicher zugegriffen wird, bietet die Klasse `outpost::BoundedArray` einen Wrapper an, um den Zugriff zu vereinfachen und nicht erlaubte Zugriffe zu verhindern. Intern nutzt BoundedArray allozierte Speicherbereiche, jedoch wird der Zugriff hierauf gekapselt.

Telecommand- / TelemetryReader und -Writer

Wie in ihren Namen bereits angedeutet, dienen die Klassen `outpost::pus::TelecommandReader` und `outpost::pus::TelecommandWriter` dem Auslesen und Einfügen von Daten in Telekommandos. Die Klasse `outpost::pus::TelemetryReader` bietet analog hierzu verschiedene Methoden an, um beispielsweise APID, Service Type, Sequenznummer usw. von Telemetriepaketen auszulesen. Für diese Daten bietet wiederum die Klasse `outpost::pus::TelemetryWriter` eine Schnittstelle, um sie zu setzen. Als Datenstruktur zum Zugriff auf Daten nutzen beide Klassen ein BoundedArray[10].

4.1.2 TM/TC-GUI

Die TM/TC-GUI (siehe Abbildung 19) ist eine vom DLR entwickelte grafische Benutzerschnittstelle auf Basis von Qt 5 und OUTPOST. Sie soll es dem Anwender ermöglichen, Remote Units zu kommandieren und Telemetriedaten zu visualisieren. Hierzu kann sie missionsspezifisch angepasst werden und stellt verschiedene Beispielumsetzungen zur Kommunikation bereit: In verschiedenen Tabs (genauer: *dockWidgets*, die vom Hauptfenster abkoppelbar sind) werden Anzeigen bereitgestellt, die Sensordaten einer RU anzeigen. Weiterhin werden beispielhafte Telekommandos bereitgestellt.
Die TM/TC-GUI stellt darüber hinaus die Möglichkeit bereit, Telekommandosequenzen zu verschicken oder - soweit vorhanden - in den internen Scheduler einer RU zu schreiben, Telekommandosequenzen per Klick auf einem Button zu senden oder zentrale Konfigurationsmerkmale (etwa die Intervalle des Erstellens eines neuen Log-Files) zu verwalten[11].

Die TM/TC-GUI stellt keine Schnittstellen[12] zur Kommunikation mit Remote Units bereit. Diese müssen vom Entwickler missionsspezifisch implementiert werden. Beispielhaft kann die Implementierung eines TCP-Transceivers genannt werden, der innerhalb der ROBEX-Mission verwendet wurde, um Telekommandos und Telemetrie mithilfe eines (W)LANs zu übertragen. Die TM/TC-GUI stellt weiterhin Widgets zur Verfügung, die genutzt werden können, um eine Mission zu kontrollieren. Hierzu gehört das *Universal TC Widget*, welches die Aufgabe hat, Telekommandos zu definieren, indem in Textfeldern und aus Auswahllisten etwa APID,

[10] Dies gilt für die in dieser Arbeit verwendete Version von OUTPOST. Aus verschiedenen Gründen dient in der aktuellen Version die Klasse `outpost::Slice` als Datenstruktur zum Datenzugriff.

[11] Die drei zuletzt genannten Funktionalitäten sowie die Nutzung von dockWidgets (statt nicht frei positionierbarer Tabs) wurden von mir im Kontext des Projektes ROBEX missionsspezifisch implementiert und im Anschluss an das Projekt in den Hauptentwicklungszweig der TM/TC-GUI zurückportiert.

[12] Eine Ausnahme bildet eine Referenzimplementierung einer seriellen Schnittstelle, über die per UART angeschlossenene Geräte angesprochen werden können.

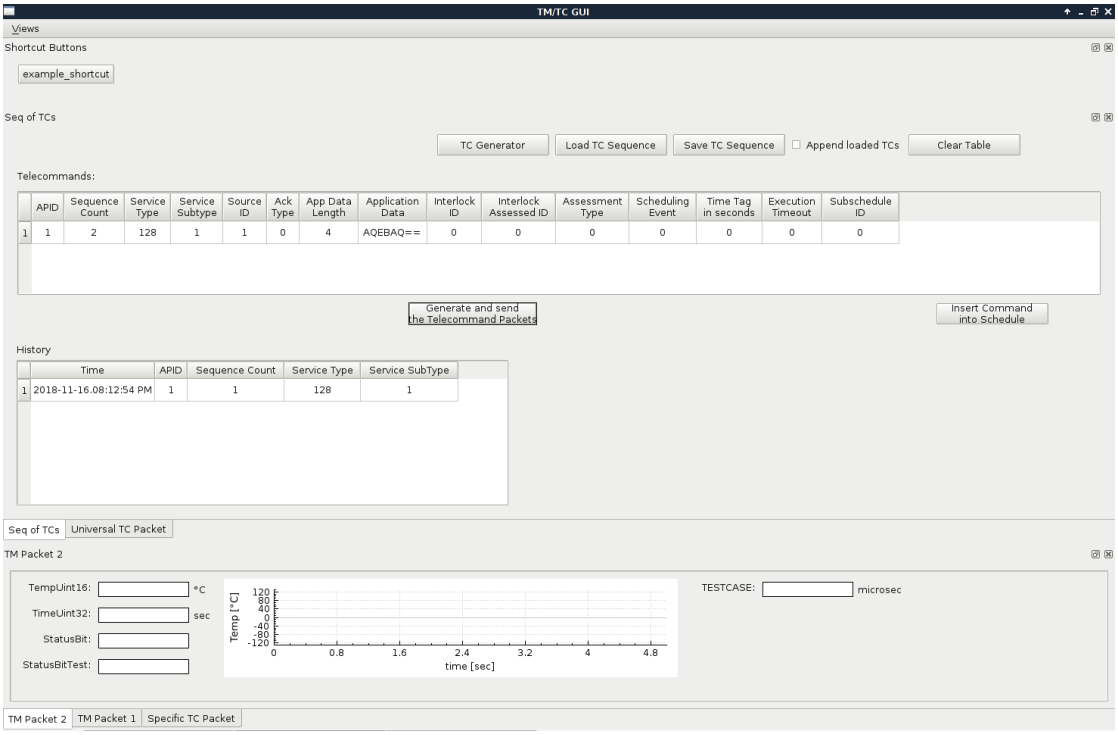


Abbildung 19: TM/TC-GUI

Service Type usw. festgelegt werden können. Dieses Telekommando kann durch einen Druck auf einen „Send"-Button an die angeschlossene bzw. verbundene Einheit gesendet werden. Ein weiteres Widget „Send Sequences of Telecommands" erlaubt die Generierung von Telekommandosequenzen, indem nacheinander einzelne Telekommandos generiert und in einer Liste gespeichert werden können. Diese werden dann durch einen Klick auf einen Button entweder direkt (nacheinander) gesendet oder in den internen Scheduler der Remote Unit geschrieben[13]. Zur Verifikation des Empfangs sowie der Ausführung von Telekommandos gibt es ein (in das Universal TC Widget eingebettetes) Anzeigeelement zur Darstellung von eingehenden Acknowledgements.

4.1.2.1 Dispatcher-Matcher-Pattern

Um eingehende Telemetriepakete an die verarbeitenden Klassen bzw. deren Instanzen weiterzuverteilen, wird das (innerhalb des Source-Codes sogenannte) Dispatcher-Matcher-Pattern als Entwurfsmuster verwendet. Hierbei verteilt ein (etwa an einen TCP-Server angeschlosse-

[13] Hierzu werden die Telekommandos als Application Data zusammen mit weiteren Informationen (etwa über die Abhängigkeit der Ausführung eines Befehls von der korrekten oder nicht erfolgten Ausführung eines vorangegangenen Befehls) in ein spezielles Scheduling-Telekommando eingebettet.

ner) Dispatcher Telemetriepakete an alle bei ihm registrierten Matcher. Hierzu wird Anzeigeelementen (z.B. DockWidgets) eine Referenz auf den Dispatcher übergeben. Das Anzeigeelement selbst kümmert sich um die Instanziierung des Matchers, welche für die Aktualisierung der angezeigten Daten zuständig ist. Dieser Matcher wird dem Dispatcher durch Aufruf der Methode

```
addMatcher(QSharedPointer<ListenerInterface> matcher);
```

übergeben. Kommt nun ein Telemetriepaket an, welches dem Dispatcher übergeben wird, geht dieser die Liste seiner registrierten Matcher durch und ruft darauf die Methode

```
doMatching(outpost::pus::TelemetryReader mTelemetryReader)
```

auf, um jedem Matcher das eingehende Telemetriepaket zu übergeben. Innerhalb dieser Methode kann eine weitere Prüfung stattfinden, ob Pakete dort weiterverarbeitet werden sollen (etwa zur Anzeige einer Gruppe bestimmter Betriebsparameter mit dem selben Service Type) oder ob dies für Pakete gilt (beispielsweise bei einem Telemetrielogger).

4.2 Implementierung

Im Folgenden wird beschrieben, wie das User-Interface zur Nutzung des Service-Discovery-Protokolls umgesetzt wurde.

4.2.1 UDP-Verbindung

Um einen Datenaustausch zwischen GUI und Remote Units zu ermöglichen, werden die Daten (Telekommandos und Telemetrie) per UDP übertragen. Die GUI bietet hierfür die Klasse `Sender`, welche Pakete entgegennimmt und sie an eine bestimmte Zieladresse weiterleitet sowie die Klasse `Receiver`, die Pakete empfängt und an den ConnectionManager weiterleitet. Durch die Verwendung von UDP ist es (im Gegensatz zum im Projekt ROBEX verwendeten TCP) möglich, Broadcasts zu senden. Dies ist für die Service Discovery auf Netzebene erforderlich, da nur so innerhalb kurzer Zeit (mit nur einer einzigen Nachricht) alle sich im festgelegten Subnetz befindlichen Remote Units angesprochen werden können.
UDP bietet im Gegensatz zu TCP keine Transportkontrolle, d.h. es kann nicht festgestellt werden, wenn Pakete nicht ihr Ziel erreichen. Bei Telekommandos wird jedoch durch die angeforderten Acknowledgements (in der Anwendungsschicht realisiert durch die OUTPOST-Library) sichtbar, wenn ein Telekommando (nicht) empfangen wurde.

4.2.1.1 Sender

Die Klasse `net::Sender` stellt zwei Methoden bereit, um UDP-Pakete zu versenden. `Sender::sendData(const uint8_t* buffer, uint16_t length)` ermittelt mithilfe von `getifaddrs` die Broadcastadresse des Subnetzes und ruft hiermit `Sender::sendData(const uint8_t* buffer, uint16_t length, **uint32_t destIp**)` auf, um das Paket zu verschicken. Diese Methode legt einen Socket mit dem Protokoll UDP an und setzt mithilfe von `setsockopt(mSocket, SOL_SOCKET, SO_BROADCAST, ...)` die Option für einen möglichen Broadcast. Ein Struct des Typs `sockaddr_in` wird angelegt und hierin Ziel-IP und Zielport 32108 festgelegt. Ein Aufruf von

```
int sentBytes = sendto(mSocket, buffer, length, 0, (struct sockaddr *) &
    destAddr, sizeof(destAddr));
```

sendet das Paket an das Zielsystem (Remote Unit).

4.2.1.2 Receiver

Der Receiver empfängt UDP-Pakete und leitet sie an die weiterverarbeitende Klasse ConnectionManager weiter.
Dazu wird ein Thread gestartet, der einen UDP-Socket anlegt und an den eigehenden Port 32109 bindet und mit

```
recvfrom(sockfd, buf, BUFSIZE,0, (struct sockaddr*) &clientaddr, &clientlen);
```

ankommende Pakete entgegennimmt. Aus der Struktur `clientaddr` (vom Typ `sockaddr_in`) wird die IP der Remote Unit ermittelt und zusammen mit den angekommenen Daten an den ConnectionManager übergeben. Um eine Trennung der Threads des Receivers sowie der grafischen Oberfläche zu erreichen, wurde mithilfe des Signal-Slot-Konzeptes die (die Daten) weitergebende Methode angebunden, indem

```
connect(this, SIGNAL(readyRead()), this, SLOT(receiveData()));
```

im Kostruktor aufgerufen wird. Das Signal `readyRead` wird aufgerufen, nachdem die eingehenden Daten in ein Objekt vom Typ `outpost::pus::TelemetryReader` umgewandelt wurden. Die Methode `receiveData` ruft den ConnectionManager mit dem Telemtriepaket auf.

4.2.1.3 Connection-Manager

Da es innerhalb dieser Master Thesis vorgesehen ist, eingehende Datenpakete von verschiedenen Remote Units getrennt weiterzuverarbeiten, dient der Connection-Manager als Zwi-

schenschicht zwischen Receiver und Connection-Wrapper. Bei einem eingehenden Paket wird in einer Liste ein Connection-Wrapper gesucht, bei dem die Sender-IP hinterlegt ist. Ist dies nicht der Fall, wird ein neuer Connection-Wrapper angelegt. Das eigehende Paket wird nun an den zuständigen Connection-Wrapper weitergeleitet.

4.2.1.4 Connection-Wrapper

Der Connection-Wrapper hat die Aufgabe, die zu einer (Quasi-)Verbindung zugehörigen Daten zu kapseln. Hierzu gehört die IP der Remote Unit, der zur Remote Unit gehörende Dispatcher sowie das Widget, welches die empfangenen Daten anzeigt. Eingehende Pakete werden an den zur Remote Unit gehörenden Dispatcher weitergeleitet, der wiederum für die Weiterverteilung zuständig ist. Der Connection-Wrapper legt auch das Widget für die Remote Unit an und heftet es an das Hauptfenster.

4.2.1.5 Main-Dispatcher

Der Main-Dispatcher hat die Aufgabe, *alle* eingehenden Telemetriepakete weiterzuverteilen. Dies dient der Möglichkeit, innerhalb der GUI in einem einzigen Fenster alle Acknowledgements von allen Remote Units gemeinsam anzuzeigen und dem Telemetrielogger alle eingehenden Pakete zu übergeben. Er bietet die öffentlichen Methoden `addMatcher` und `handlePacket`. `addMatcher` trägt einen Matcher in eine Liste ein, `handlePacket` verteilt eingehende Telemetriepakete (sowie die IP-Adresse) an alle Matcher aus dieser Liste.

4.2.1.6 Remote-Unit-Dispatcher

Ebenso wie der Main-Dispatcher bietet der Remote-Unit-Matcher Schnittstellen, um Matcher einzutragen und Pakete an eingetragene Matcher weiterzuverteilen. Seine Funktionsweise ist dieselbe, jedoch mit dem Unterschied, dass ihm nur Pakete einer bestimmten Remote Unit übergeben und diese nur an den Matcher des zur Remote Unit gehörenden Widgets weitergeleitet werden. Da eine eindeutige Zuordnung der Telemetriepakete zu den Remote Units bzw. den zugehörigen Matchern und Widgets bereits im Connection-Manager geschieht, ist es an dieser Stelle nicht notwendig, die IP-Adresse der Remote Unit als Parameter mit zu übertragen.

4.2.2 Remote-Unit-Matcher

Der Remote-Unit-Matcher hat die Aufgabe, von den RUs ankommende Telemetriepakete entgegenzunehmen und zu verarbeiten. Für diesen Zweck selektiert er Pakete mit dem Service

Type 128 und dem Service Subtype 151 (Telemetriepakete vom Sensor-Thread). In diesen Telemetriepaketen sind zum Einen die Sensortypen sowie ggf. CAN-I^{2}C-Bridge-IDs und zum Anderen die entsprechenden Messwerte codiert und gespeichert. Für Aktoren finden sich Informationen zu deren Kommandierung (APID, Service Type und Service Subtype). Die ausgelesenen Daten werden in mit dem Widget geteilten Puffern (`DataModel`, `SensorModel`, `IdModel` und `ActorModel`)realisiert als `std::vector<uint8_t>`) gespeichert. Bei neu ankommenden Daten sowie sich verändernden Sensortypen wird das Remote-Unit-Widget (siehe Unterunterabschnitt 4.2.4) informiert.
Die Verarbeitung der Telemetriepakete verläuft wie folgt:
Das Informieren des Remote-Unit-Widgets funktioniert mithilfe eines QT-SIGNALS, welches mit einem QT-SLOT im Remote-Unit-Widget verbunden ist. Wird das Signal *newTypes* vom Matcher gesetzt, führt dies zur Ausführung der Methode `updateTypes` im Remote-Unit-Widget. Analog führt ein gesetztes Signal *newData* zur Ausführung der Methode `updateData`. Eine Erläuterung hierzu findet sich in Unterunterabschnitt 4.2.4.

4.2.3 Main-Window

Das Main-Window beinhaltet die anzuzeigenden Widgets sowie eine Menüleiste, in der deren Sichtbarkeit eingestellt werden kann. Neben den Remote-Unit-spezifischen Widgets enthält es auch eines, mit dem Telekommandos verschickt werden können („Universal TC Packet“), eines, das ankommende Acknowledgements mit dazugehöriger IP anzeigt („Verification“) und einen Telemetrielogger („TM-Logger“).

Abbildung 20 zeigt das Main-Window mit dem Widget für drei Remote Units (rechts oben, mittig, unten rechts), dem Universal-TC-Packet-Widget (linke Hälfte) mit Eingabemöglichkeiten der Parameter des zu sendenden Telekommandos (und einer History bereits gesendeter Telekommandos), ein Widget zur Anzeige der von der Remote Unit gesendeten Acknowledgements (rechts mittig) sowie den Telemetrielogger (oben mittig).

4.2.4 Remote-Unit-Widget

Das Remote-Unit-Widget dient der Anzeige der von der Remote Unit empfangenen Daten sowie der Kontrolle der daran angeschlossenen Aktoren. Es teilt sich mit dem Matcher mehrere gemeinsame Puffer für anzuzeigende Sensortypen, IDs (von an CAN-I^{2}C-Bridges angeschlossenen Sensoren), gemessene Daten und Steuerinformationen für Aktoren. Mithilfe des Signal-Slot-Konzeptes von QT wird dem Widget vom Matcher mitgeteilt, dass neue Daten oder Sensoren vorhanden sind.

Algorithm 1: doMatching Verarbeitet ein Telemetriepaket

```
   Input: Telemetry Packet t
1  if t.serviceType == 128 AND t.serviceSubType == 151 then
2      for i ← 0 to t.getAppDataLength do
3          DataModel ← ∅
4          TypeModel ← ∅
5          IdModel ← ∅
6          ActorModel ← ∅
7          b ← t.AppData[i]
8          Write b into TypeModel
9          i ← i + 1
10         switch b do
11             case Sensortyp::TEMPERATURE do
12                 Read 1 Bytes of Data at Position i
13                 Write Data into DataModel
14                 i ← i + 1
15             case Sensortyp::LUMINOSITY do
16                 Read 4 Bytes of Data at Position i
17                 Write Data into DataModel
18                 i ← i + 4
19             case Sensortyp::CAN_LUMINOSITY do
20                 Read 1 Byte at position i
21                 Write CAN-I²C-Bridge-ID to IdModel
22                 Read 4 Bytes of Data at Position i + 1
23                 Write Data into DataModel
24                 i ← i + 5
25             case Sensortyp::CAN_LCD do
26                 Read 1 Byte at position i
27                 Write CAN-I²C-Bridge-ID to IdModel
28                 Read 4 Bytes of Data at position i + 1
29                 Write Data into ActorModel
30                 i ← i + 5
31             ...
32     if TypeModel has changed then
33         Inform widget about new Sensor-Types
34     Inform widget about new Sensor-Data
```

Abbildung 21 zeigt das Klassendiagramm des Widgets mit den wichtigsten Attributen und Methoden. Innerhalb der ersten vier Vektoren werden Anzeigeelemente zwischengespeichert,

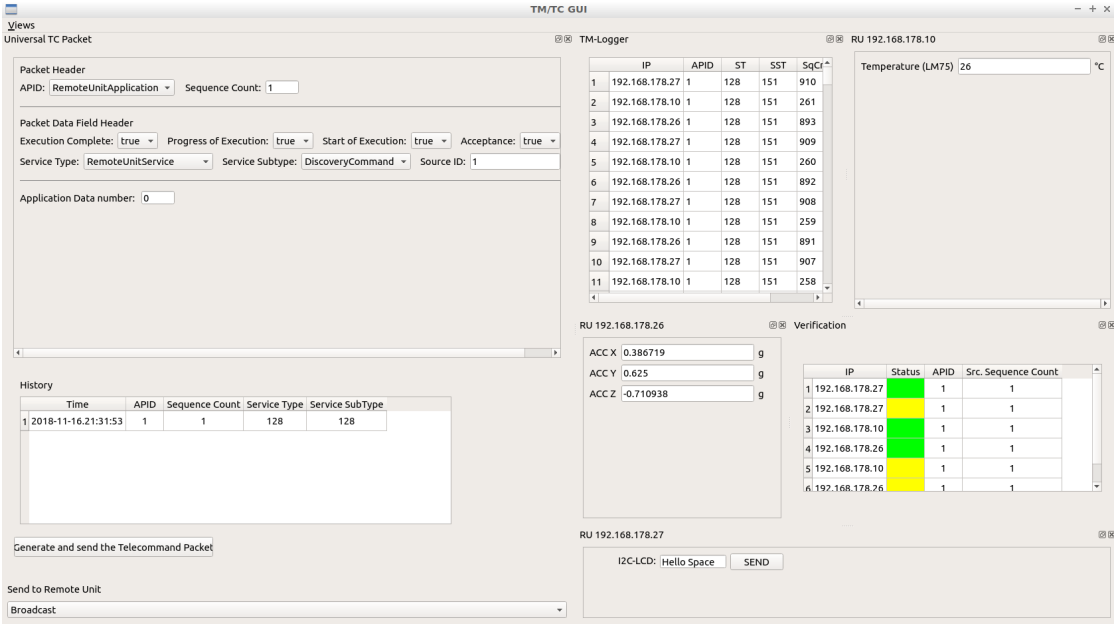


Abbildung 20: Main-Window mit Widgets

«class» RemoteUnitWidget
std::vector<QLineEdit*> mLineEditVector std::vector<QLabel*> mLabelVector std::vector<TextField*> mTextFieldsVector std::vector<PwmSlider*> mPwmSliderVector std::vector<uint8_t> dataModelVector std::vector<uint8_t> typeModelVector std::vector<uint8_t> idModelVector std::vector<uint8_t> actorDataModelVector
void updateData() void updateTypes() void addDataField(std::string description, std::string unit) void addTextField(uint16_t apid, uint16_t serviceType, uint16_t serviceSubType, std::string TextLabel) void addPwmSlider(uint16_t apid, uint16_t serviceType, uint16_t serviceSubType) void flushViewElements()

Abbildung 21: Klassendiagramm des Remote-Unit-Widget

um deren Daten zu aktualisieren oder sie entfernen zu können. Die angesprochenen Buffer sind als Vektoren mit `uint_8`-Werten realisiert.
Der Matcher ruft bei empfangenen Daten (über eine Signal-Slot-Verbindung) die Methode `updateData()` auf. Deren Ablauf ist in Listing 2 beschrieben.

Algorithm 2: RemoteUnitWidget::updateData()

```
1  for each Byte b in typeModelVector do
2      switch b do
3          case Sensortyp::TEMPERATURE do
4              Write one Byte from DataModel to temp
5              Set next LineEdit's Text to temp
6          case SensorType::LUMINOSITY do
7              Write next four Bytes from DataModel into uint32_t-value lux
8              Set next LineEdit's Text to lux
9          case SensorType::ACCELEROMETER do
10             Write two Bytes from DataModel into uint16_t-value accX
11             Set next LineEdit's Text to accX
12             Write two Bytes from DataModel into uint16_t-value accY
13             Set next LineEdit's Text to accY
14             Write two Bytes from DataModel into uint16_t-value accZ
15             Set next LineEdit's Text to accZ
16         ...
```

Beispielhaft sind hier drei Sensortypen gezeigt. Für den ersten wird der Messwert in einem einzigen Byte gespeichert, beim nächsten umfassen die Daten vier Bytes und beim letzten werden die Daten in drei Anzeigen geschrieben, die jeweils zwei Bytes umfassen. Anzumerken ist, dass die Reihenfolgen der Sensor- / Aktordaten in den jeweiligen Puffern gleich sind. Daher kann für jede Anzeige bestimmt werden, welche und wieviele Bytes des DataModels für sie bestimmt sind.

Ändern sich die angeschlossenen Sensortypen, wird durch den Remote-Unit-Matcher die Methode `updateTypes()` (siehe Listing 3) aufgerufen. Diese löscht zunächst alle Anzeigeelemente und fügt die Sensoren (ggf. erneut) zum Widget hinzu, indem es `addDataField` mit den Parametern *Beschriftung* und *Einheit* aufruft.

Weiterhin werden Anzeigen für die Aktoren in das Widget eingefügt. Mithilfe der Methoden `addTextField` und `addPwmSlider` werden Anzeigen zur Steuerung von Aktoren eingefügt. `addTextField` fügt ein Textfeld und einen „Send“-Button ein. Mit diesem Button wird ein Telekommando verschickt, welches den eingegebenen Text enthält. Damit dies korrekt adressiert werden kann, werden der Methode `addTextField` als Parameter die entsprechende APID, der Service Type und der Service Subtype übergeben. Analog findet das Hinzufügen eines Sliders statt, der einen per PWM gesteuerten Motor kontrolliert. Hierbei wird nach jeder Manipulation des Sliders ein neues Telekommando mit Stellwert verschickt.

Algorithm 3: RemoteUnitWidget::updateTypes()

```
1  flushViewElements()
2  for each Byte b in TypeModel do
3      switch b do
4          case Sensortyp::TEMPERATURE do
5              addDataField("Temperature (LM75)", "°C")
6          case SensorType::CAN_TEMPERATURE do
7              read id from IdModel
8              addDataField("Temperature (LM75, ID "+ id + ")", "°C")
9          case SensorType::CAN_LCD do
10             get APID, Service Type and Service Subtype from ActorModel
11             get id from IdModel
12             addTextField(apid, serviceType, serviceSubType, "LCD (ID "+ id + ")")
13         ...
```

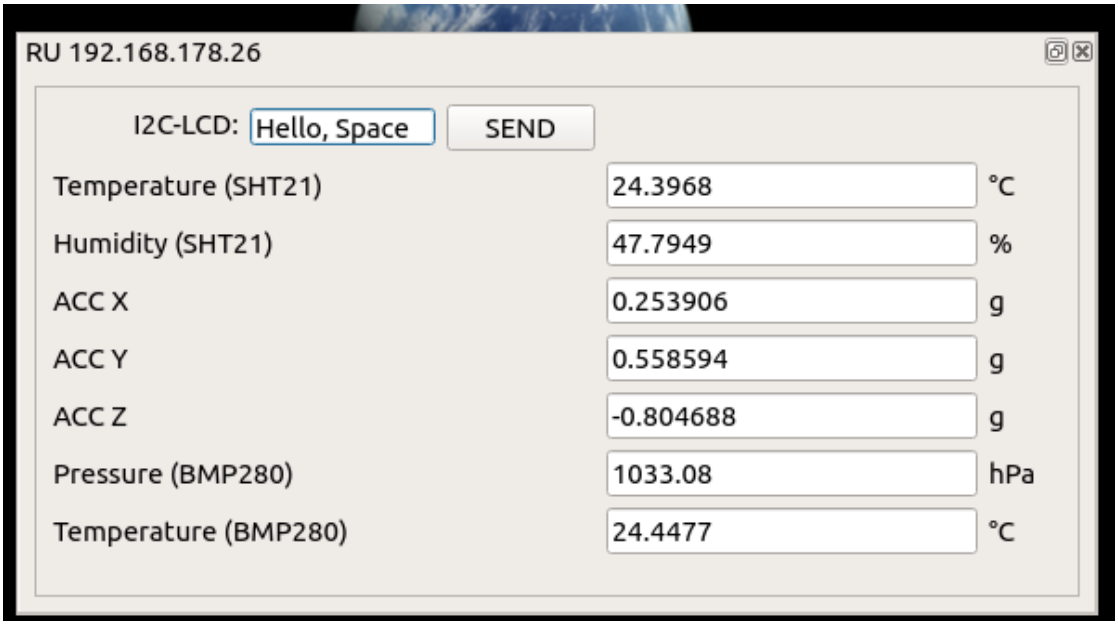


Abbildung 22: Abgelöstes Remote-Unit-Widget mit Anzeigeelementen für Sensoren und Aktoren

Abbildung 22 zeigt das Remote-Unit-Widget. Es zeigt in seinem Titel die IP der Remote Unit an. Weiterhin enthält es Daten von einem Aktor (LCD) und drei Sensoren (Accelerometer, Luftfeuchtigkeits- und Temperatursensor sowie Luftdruck- und Temperatursensor).

4.2.5 Anzeigeelemente

Zur Anzeige von Telemetriedaten sowie zur Bedienung von Aktoren wurden generische Anzeigeelemente entworfen und implementiert, die im Folgenden vorgestellt werden.

Data-Field

Das Data-Field dient der Anzeige von numerischen Sensormesswerten. Es kombiniert ein Textfeld, in dem der gemessene Wert angezeigt wird, mit zwei Labels. Das links vom Textfeld angezeigte Label enthält die Bezeichnung des Sensors (sowie im Fall der Nutzung der CAN-I^{2}C-Bridge deren ID). Das rechts vom Textfeld angezeigte Label gibt die Einheit des Messwertes an.

Abbildung 23: Data-Field: Temperaturanzeige mit ID und Einheit

Abbildung 23 zeigt ein Data-Field zur Temperaturanzeige. Erkennbar ist weiterhin der Sensortyp, die ID der CAN-I^{2}C-Bridge sowie die Einheit °C.
Zum Hinzufügen eines Data-Fields wird die (für alle Sensoren, die numerische Werte liefern) generische Methode `addDataField(std::string description, std::string unit)`. verwendet

Text-Field

Ein Text-Field dient der Eingabe und dem Absenden von Zeichenketten, wie sie beispielsweise auf einem LCD ausgegeben werden können. Es ist in Abbildung 24 zu erkennen.

Abbildung 24: Text-Field mit Label, Textfeld und Send-Button

Die Methode `addTextField(uint16_t apid, uint16_t serviceType, uint16_t serviceSubType, std::string pTextLabel)` fügt ein Textfeld mit einem Send-Button hinzu. Wird dieser angeklickt, sendet das Text-Field ein Telekommando an die als Parameter angegebene Adressierung aus APID, Service Type und Service Subtype,

dessen Application Data dem eingegebenen Text entspricht. Mithilfe eines regulären Ausdrucks wird verhindert, dass ungültige (nicht auf dem LCD anzeigbare) Zeichen eingegeben werden können. Ebenso ist die Länge der Eingabe auf 32 Zeichen beschränkt (Größe des in dieser Arbeit verwendeten Displays mit $2 \cdot 16$ Zeichen).

PWM-Slider

Der PWM-Slider dient der Kontrolle eines per PWM angeschlossenen Servomotors. Wie in Abbildung 25 zu sehen ist, wird ein Slider angezeigt. Wird dieser bewegt, dreht sich der Servomotor entsprechend. Der linke bzw. rechte Rand repäsentieren hierbei die maximale Auslenkung des Servomotors.

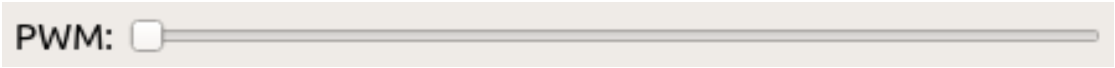


Abbildung 25: Slider zur Kontrolle von PWM-gesteuerten Servomotoren

4.2.6 Universal-TC-Packet-Widget

Das Universal-TC-Packet-Widget dient zum Senden eines Telekommandos. Wie in Abbildung 26 erkennbar, sind diverse Parameter des Telekommandos einstellbar, darunter APID, Service Type und Service Subtype, welche Acknowledgments angefordert werden sowie welche Application Data das Telekommando enthalten soll. Darunter ist eine History bereits gesendeter Telekommandos zu erkennen. Unten ist die aufgeklappte Liste mit den IPs der gefundenen Remote Units sowie dem Eintrag für einen Broadcast zu erkennen.

4.2.7 Verification-Widget

Das Verification-Widget hat die Aufgabe, eingehende Acknowledgements (Telemetriepakete vom Service Type 1, siehe Unterabschnitt 2.4: Standard-Services) zusammen mit der IP der absendenden Remote Unit anzuzeigen. Es ist an das Feld „Verification“ auf dem Universal TC Packet Widget aus der vom DLR stammenden Fassung der TM/TC-GUI angelehnt, wurde jedoch angepasst und zeigt nun zusätzlich die Spalte „IP“ an. Weiterhin wurde es abgelöst und wird als eigenständiges Widget eingesetzt.

Im in Abbildung 27 gezeigten Verification-Widget sind eingehende Acknowledgements zu erkennen. Sie enthalten dieselbe APID und Sequenznummer des versandten Telekommandos (dieses 2-Tupel identifiziert also eindeutig ein versandtes Telekommando). Der gelbe Kasten

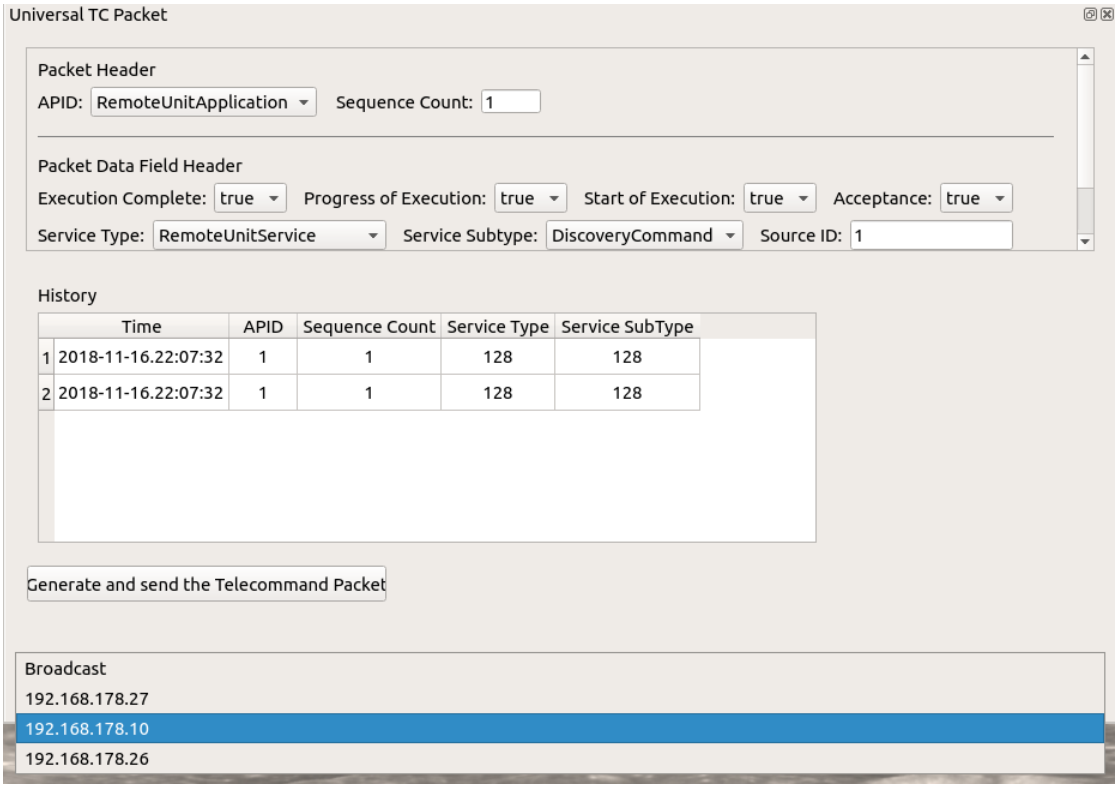


Abbildung 26: Universal-TC-Packet-Widget

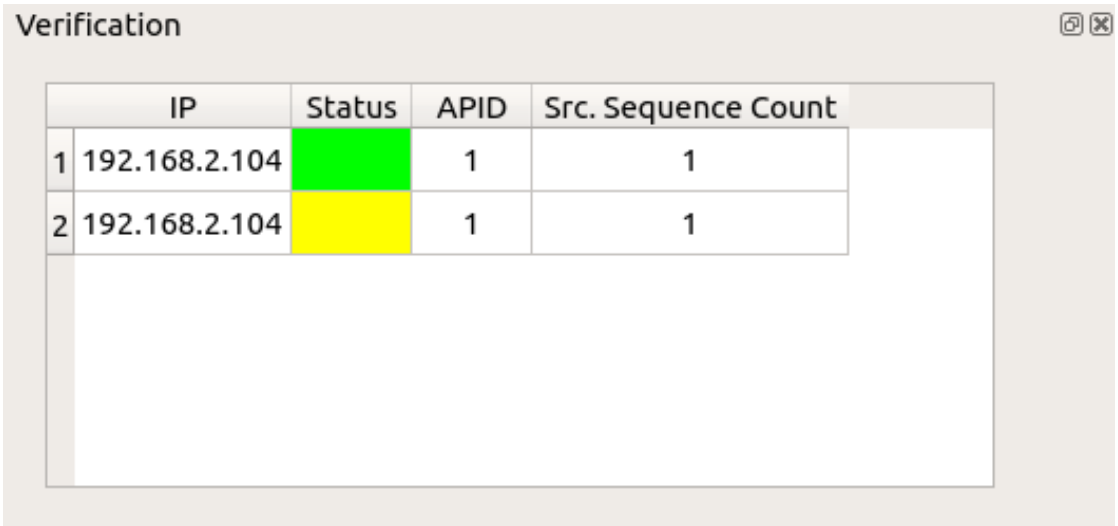


Abbildung 27: Verification-Widget

unter „Status“ quittiert den erfolgreichen Empfang des Telekommandos und der grüne die abgeschlossene Ausführung.

4.2.8 Der Telemetrielogger

Um das Protokoll zur Service-Discovery für Netze heterogener Sensoreinheiten zu evaluieren, wurde ein Telemetrielogger implementiert. Dieser ermöglicht es, Telemetriepakete zusammen mit der absendenden IP und der Application Data in einer CSV-Datei[14] zu speichern, sodass objektiv zwischen erwartetem Ergebnis und tatsächlich gemessenen Daten verglichen werden kann. Ankommende Telemetriepakete werden darüber hinaus in einer Tabelle dargestellt.

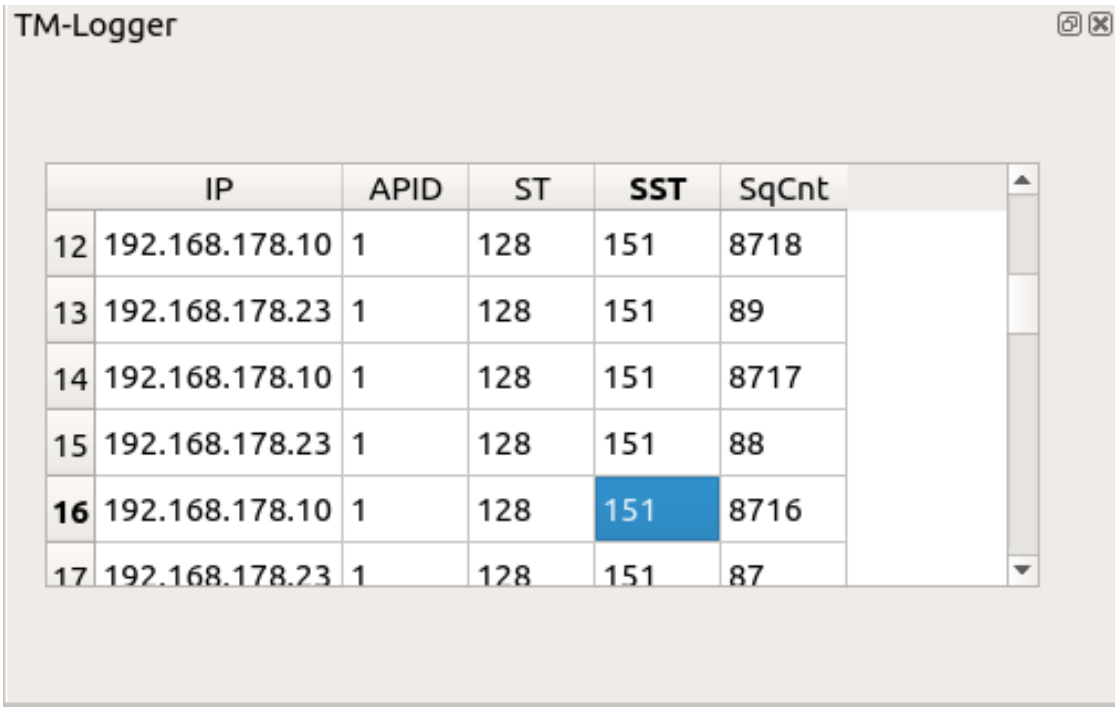


Abbildung 28: Telemetrielogger mit Tabellenansicht

Abbildung 28 zeigt das Widget mit empfangenen Telemetriepaketen von zwei Remote Units. Die Abkürzungen ST, SST und SqCnt bedeuten Service Type, Service Subtype und Sequence Count.

[14] Diese Datei wird beim Start der GUI im Unterordner /history angelegt und folgt dem Namensschema tm_history_YYYY-MM-DD_HH-MM-SS.csv .

5 Remote Unit

Dieses Kapitel behandelt die Implementierung der Remote Unit. Hierhin führend werden dabei die Motivation zum Bau einer eigenen Remote Unit sowie Details zum Vorbild *ROBEX-Remote Unit* und die technischen Grundlagen, insbesondere im Hinblick auf die verwendete Hardware (Raspberry Pi) und den softwaretechnischen Unterbau in Form der OUTPOST-Library, dargestellt.
Die Remote Unit dient dem Anschluss von Sensoren sowie Aktoren. Sie kann von einem Nutzer platziert werden und benötigt eine Stromversorgung (5V per Micro-USB) sowie eine Netzwerkverbindung; im Falle des verwendeten Raspberry Pi (1) geschieht dies mittels Ethernet. Nachdem sie mit Sensoren und Aktoren bestückt ist, kann sie ferngesteuert Messprozesse durchführen und die ermittelten Daten an das Command Center / die GUI senden, welche die Daten anzeigt und in einer Log-Datei speichert. Abbildung 33 zeigt die Remote Unit mit CAN-Controller (rechts, über dem Raspberry Pi) sowie fünf Sub-D-Buchsen zum Anschluss von Sensoren oder CAN-I^2C-Bridges. Über diese Anschlüsse werden sowohl Versorgungsspannungen (3,3V und 5V sowie Masse), die CAN-Leitungen CAN_H und CAN_L sowie die I^2C-Leitungen SDA und SCL herausgeführt. Ein zweiter CAN-Controller mit einem Arduino Nano (oben rechts) sorgt dafür, dass für alle ausgehenden Nachrichten ein Acknowledgement gesendet wird. Gleichzeitig dient dieser als CAN-Reader zum Debugging.

5.1 Motivation

Während des Projektes ROBEX (Robotische Exploration unter Extremen Bedingungen), einer simulierten Mondmission auf dem Etna (Sizilien, Italien), leistete ich einen Beitrag zum Projekt und machte hierbei weitreichende Erfahrungen bezüglich des Aufbaus einer Mission sowie der Herausforderungen bei deren Umsetzung. Da sich die Mission bei meinem Einstieg bereits in einer späten Phase befand, konnte ich nur technisches Wissen und programmiertechnische Umsetzungsfähigkeiten einbringen, erkannte jedoch an verschiedenen Stellen Möglichkeiten zur Verbesserung der Missionsarchitektur. Diese werden im Anhang innerhalb der Ideensammlung „Robex Reloaded“ (siehe Unterabschnitt 12.2) beschrieben, in der ich verschiedene Vorschläge zur Fortführung und Weiterentwicklung festhielt. Ein Aspekt betraf hierbei die Konstruktion einer Remote Unit (die in der ROBEX-Mission entwickelte Variante ist in Abbildung 29 zu sehen) mit Off-the-Shelf-Teilen. Dies meint, handelsübliche und vor allem günstige Komponenten zu verwenden, um das Missionsteilziel seismischer Messungen auf dem Vulkan mit deutlich geringerem finanzellen Aufwand zu erfüllen. Diese Motivation ergab sich aufgrund der Tatsache, dass der verwendete Microcontroller COBC (Compact On-Board Computer) - auch aufgrund der notwendigen Zertifizierung zum Einsatz

im Weltraum - verhältnismäßig teuer ist. Als Möglichkeiten wurden hier beispielhaft der ESP8266 angeführt, wobei die Liste in dieser Arbeit noch um den Raspberry Pi, ein ebenfalls mit einem Preis von ca. 35€ kostengünstiger Einplatinencomputer, ergänzt wird. Letzterer ist aufgrund seiner hardwaretechnischen Möglichkeiten in Form vieler frei nutzbarer I/O-Ports sowie seiner ausreichend hohen Rechenleistung (in Bezug auf die erwarteten Anforderungen durch die in dieser Arbeit vorgestellte Service Discovery) dem ESP8266 vorzuziehen. Ein Einsatz des ESP8266 wird im Ausblick (siehe Unterabschnitt 11.3: Andere Hardware-Basis) diskutiert werden.

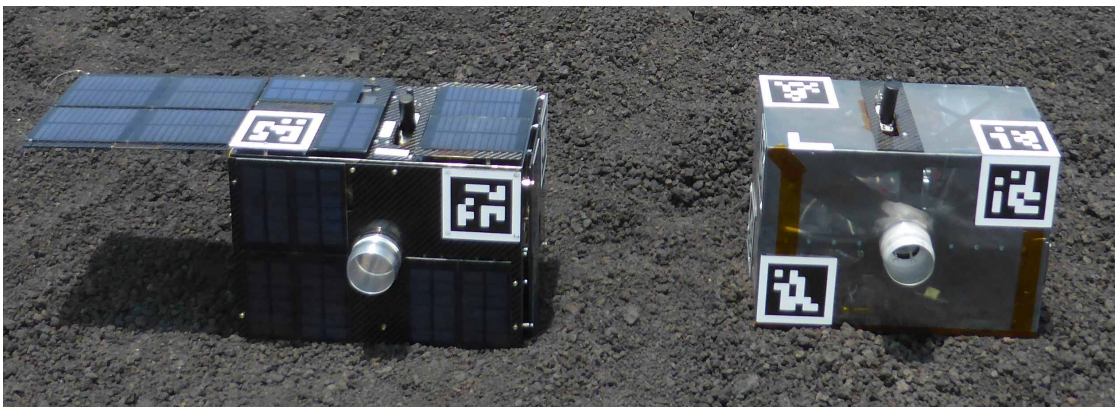


Abbildung 29: Zwei Remote Units der ROBEX-Mission
Quelle: Lars Witte, DLR.

5.2 Technische Grundlagen

In diesem Abschnitt werden technische Grundlagen zur verwendeten Harware dargelegt. Fokussiert werden hierbei Details der Architektur des Raspberry Pi im Hinblick auf das verwendete SoC sowie die Auslegung der I/O-Pins. Ebenfalls vorgestellt werden das Betriebssystem *Raspbian*, das verwendete CAN-Modul sowie die verwendeten Komponenten der Bibliothek OUTPOST.

5.2.1 Raspberry Pi

Der Raspberry Pi (1) (abgebildet auf Abbildung 31) ist ein Einplatinenrechner mit einem SoC BCM2835 von Broadcom sowie 512MByte LPDDR-RAM. Das SoC beinhaltet einen ARM1176JZFS Einkern-Prozessor mit ARMv6-Architektur sowie einer Videocore IV-GPU. Er verfügt über insgesamt 26 teils frei benutzbare I/O-Pins (siehe Abbildung 30) zur Kommunikation mit weiteren Komponenten, darunter per SPI, I^2C und UART. Mit 700 MHz Takt-

frequenz bietet er ausreichend Rechenleistung, um die Demonstration des Service-Discovery-Protokolls zu ermöglichen. Er bietet eine 100 Mbps-Ethernet-Schnittstelle, zwei USB 2.0-Ports, einen Monitor-Anschluss per HDMI oder Composite sowie einen Audioausgang. Neben diesen Schnittstellen bieten die auf der Platine vorhandenen ZIF-Sockel die Möglichkeit, ein LCD sowie eine Kamera zu betreiben. Für Datenspeicher ist ein SD-Kartenslot auf der Rückseite der Platine vorgesehen, auf denen auch das zu startende Betriebssystem gespeichert ist. Abbildung 30 zeigt die GPIO-Leiste des Raspberry Pi.

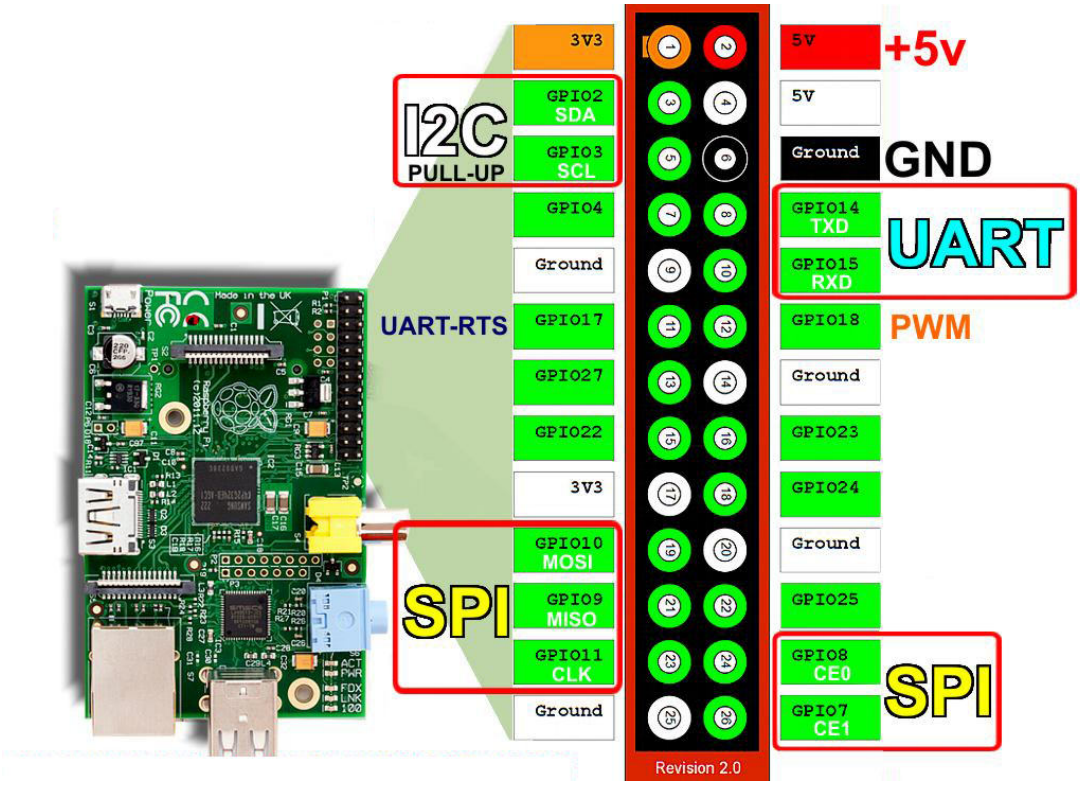


Abbildung 30: Pinout des Raspberry Pi
Quelle: [Zie13].

5.2.2 Raspbian

Raspbian ist eine auf Debian basierende Linux-Distribution. Hierfür sind aufgrund der großen Community diverse Werkzeuge vorhanden, die zur Anpassung an eigene Anwendungsbereiche dienen. Darunter gehören die GNU Compiler Collection (GCC) zum Kompilieren von u.a. C- und C++-Code, die Standard-Library (g)libc und der Paketmanager apt. Weiterhin bietet es die Möglichkeit, angeschlossene Peripherie einzubinden, indem diese mithilfe von Device-

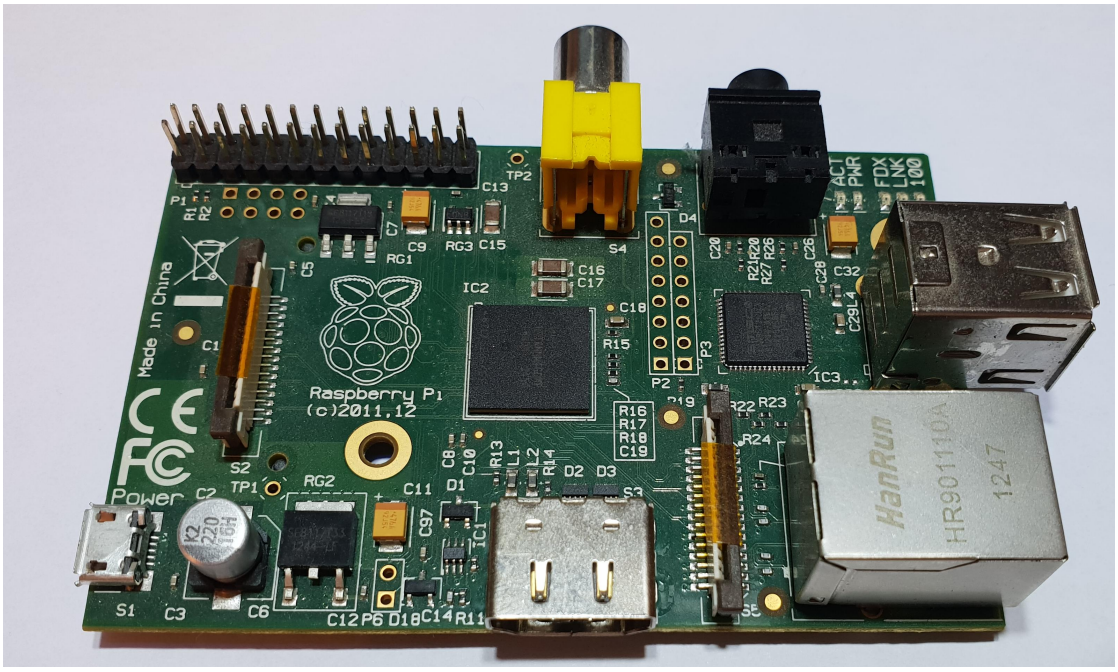


Abbildung 31: Ein Raspberry Pi (1, Revision 2, Modell B)
Quelle: Eigenes Bild.

Tree-Overlays in den Baum verfügbarer Hardwarekomponenten eingebunden werden.
Raspbian wurde als Betriebssystem für die Remote Unit ausgewählt, da es zum Einen kostenfrei ist und durch eine große Community unterstützt wird und zum Anderen POSIX-kompatibel ist. Der Autor dieser Arbeit verfügt über Erfahrungen im Bereich der Programmierung unter Linux, was die Entscheidung in diese Richtung begünstigt.

5.2.3 OUTPOST

Im Folgenden werden die Grundlagen erklärt, die benötigt werden, um mithilfe von OUTPOST eine eigene Anwendung zu implementieren. Wie bereits erwähnt, ist OUTPOST eine Library des Deutschen Zentrums für Luft- und Raumfahrt (DLR e.V.) zum Betrieb von Weltraumapplikationen. Hierbei ist zu erwähnen, dass mit OUTPOST auch Anwendungen entwickelt werden können, die nicht im Weltraum genutzt werden sollen. Diese Möglichkeit wurde bereits beim Projekt ROBEX (siehe Unterabschnitt 5.1) gezeigt. Innerhalb dieser Arbeit wird OUTPOST ebenfalls für eine Nicht-Weltraum-Anwendung genutzt. Hierfür spricht, dass das Missionsdesign einer „ausgesetzten“ Remote Unit, welche nicht durch einen Techniker oder Wissenschaftler gewartet werden kann, dem eines in den Weltraum gebrachten Satelliten ähnelt. Es wird daher die Möglichkeit der Nutzung der Bibliothek über

ihren ursprünglichen Anwendungszweck hinaus demonstriert.
Um mithilfe von OUTPOST einen Service zu implementieren, muss - entsprechend des vorgesehenen Service-Konzeptes - zunächst eine Application instanziiert werden, der dann eine Menge von Services zugeordnet werden. Hierbei ist zu erwähnen, dass Applications mithilfe einer APID (Application Process ID) und Services anhand ihres Service Types identifiziert werden. Weiterhin ist es notwendig, ein Telekommando-Dispatcher-Topic zur Verteilung eingehender Telekommandos an die entsprechenden Applications einzurichten und diesem eine Referenz auf einen Clock-Service zu übergeben, der etwa die Zeitstempelung übernimmt.
Die Verteilung von eingehenden Telekommandos ist in der Klasse `outpost::pus::Application` festgeschrieben, die wiederum Superklasse selbst implementierter Applications ist. Im Service-Konzept von OUTPOST ist festgelegt, dass bis zu 2047[15] Applications angelegt werden können, die wiederum Services beinhalten. Geht ein Telekommando ein, wird die Methode `Application::executeCommand(TelecommandReader telecommand)` aufgerufen, die zunächst innerhalb einer Liste registrierter Services nach dem passenden (anhand des Service Types identifizierten) sucht, um diesem das Telekommando zur Abarbeitung zu übergeben. Findet sich kein passender Service, übernimmt die Application selbst die Abarbeitung des Telekommandos.
Da OUTPOST selbst keine Klassen zur Kommunikation (unterhalb der Anwendungsschicht) anbietet, müssen diese vom Anwender implementiert werden. Ein Interface definiert hierbei, mithilfe welcher Methode ein eingehendes Telekommando-Paket, beispielsweise von einem UDP-Server, an den Telecommand-Dispatcher weiterverteilt wird. Auch für ausgehende (Telemetrie-)Pakete steht ein Interface bereit, sodass diese versendet werden können.
OUTPOST bietet somit einen technischen Unterbau zur Abstraktion von Hardware und Betriebssystem, interne und externe Kommunikationsschnittstellen sowie einige der innerhalb des Packet Utilization Standards beschriebene Standard Services.

5.2.4 CAN-Transceiver

Der CAN-Transceiver dient der Kommunikation mit per CAN-Interface angeschlossenen Geräten. Ein Modul mit einem MCP2515 als Controller- und einem TJA1050 als Transceiver-Chip ist auf Abbildung 32 zu erkennen.
Der MCP2515 ist ein CAN-Protokoll-Controller mit SPI-Interface. Er ermöglicht eine Datenübertragung mit bis zu 1 MBaud, kann dabei pro Datenpaket 0-8 Byte versenden und un-

[15] Bei 11 Bit APID-Länge und einer reservierten APID.

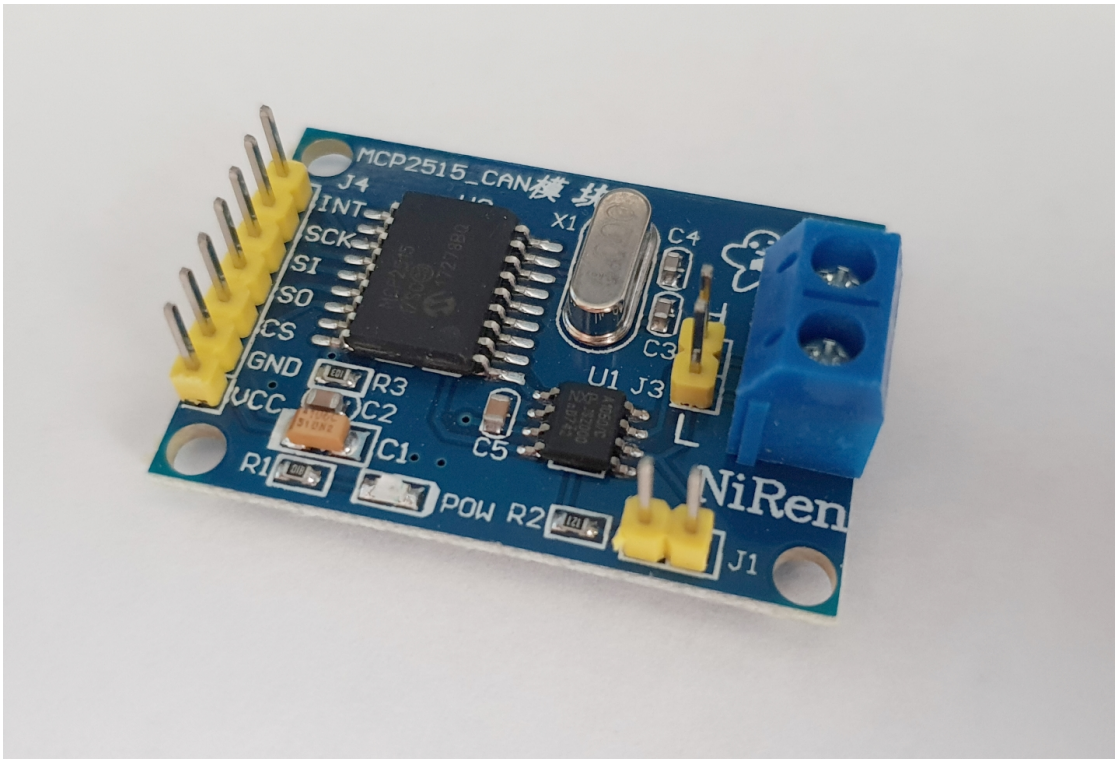


Abbildung 32: CAN-Modul mit MCP2515 und TJA1050
Quelle: Eigenes Bild.

terstützt sowohl Standard-Frames mit 11-bittiger als auch Extended Frames mit 29-bittiger ID. Er besitzt einen Datenpuffer für zwei eingehende und drei ausgehende Frames. Per SPI ist eine Kommunikation bei bis zu 10 MHz Takt möglich. Ein Interrupt-Pin ermöglicht das automatisierte Abfragen durch einen SPI-Master, wenn ein Datenpaket angekommen ist. Es wird eine Spannungsversorgung von 2,7 - 5,5V vorausgesetzt, wobei betriebstypisch 5 mA Strom fließen ([Mic07]).
Der CAN-Transceiver TJA1050 dient nur der Konvertierung des Ausgangssignals des MCP2515 auf einen CAN-Signalpegel. Er benötigt eine Versorgungsspannung zwischen 4,75 und 5,25V und ermöglicht die Umsetzung von CAN-Signalen bei bis zu 1 MBaud ([Sem03]).

5.3 Implementierung

Im Folgenden wird dargestellt, wie die Remote Unit technisch - auf Hard- und auf Softwareebene - aufgebaut ist.

5.3.1 Mechanischer Aufbau

Auf der Hardwareseite wurde die Remote Unit (siehe Abbildung 33) umgesetzt, indem ein Raspberry Pi sowie das CAN-Modul auf einem stabilen Holzbrett befestigt und Anschlüsse für die Sensoren platziert wurden. Hierfür werden, wie im Bild zu erkennen ist, Sub-D-Buchsen genutzt. Dieser Aufbau ermöglicht auch stabile Konnektoren zum Anschluss der Sensoren/Aktoren und CAN-I^2C-Bridges, indem die Sub-D-Buchsen mithilfe von Aluminiumrohren und Schrauben befestigt sind. Über diese werden sowohl Versorgungsspannungen (5V und 3,3V sowie Masse) als auch die Interfaces CAN und I^2C bereitgestellt. Ein zweites CAN-Modul mit einem Arduino Nano ist oben rechts zu erkennen. Um der Remote Unit mechanische Stabilität zu verleihen, wird ein ca. 2cm dickes Brett aus Pressholz als Basis für den Aufbau verwendet. Die Stromversorgung geschieht mithilfe eines 5V-Netzteils (ursprünglich zum Laden von Smartphones verwendet) und einem USB-Kabel mit Micro-USB-Konnektor, welches in der Abbildung fehlt.

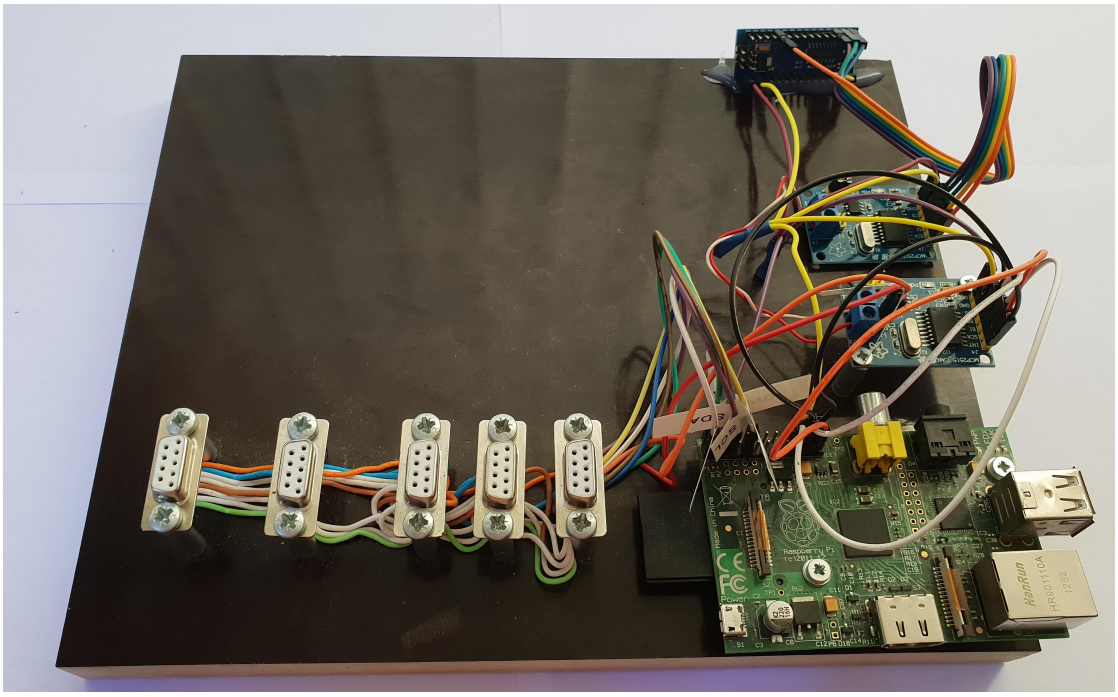


Abbildung 33: Remote Unit in einem prototypischen Aufbau
Quelle: Eigenes Bild.

5.3.2 CAN-Modul

Das in Abbildung 33 oberhalb des Raspberry Pi zu erkennende CAN-Modul wird - vorgegeben durch das in Abbildung 30 gezeigte Pinout des Raspberry Pi - per SPI mit diesem verbunden und ebenfalls über dessen GPIO-Pins mit Spannung versorgt. Es ist ebenfalls mit Schrauben und Aluminiumrohren an der Grundplatte befestigt. Der Pin *CS* des CAN-Transceivers wird mit Pin *CE0* (GPIO8) des Raspberry Pi verbunden. Abbildung 34 zeigt den Schaltplan, um

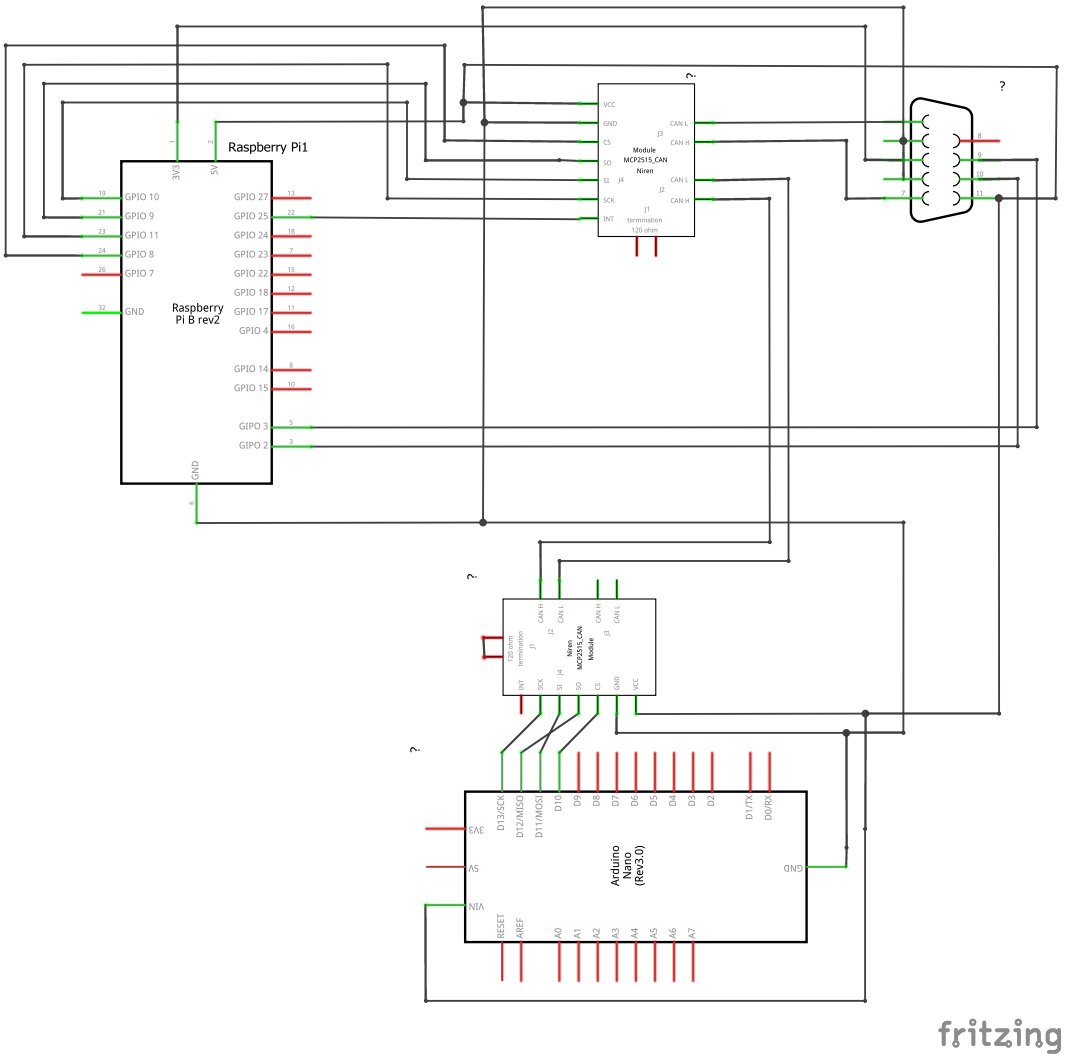


Abbildung 34: Anschluss des CAN-Moduls an den Raspberry Pi

das CAN-Modul mit dem Raspberry Pi zu verbinden.
Das zweite CAN-Modul mit angeschlossenem Arduino Nano wird benötigt, damit gesendete

CAN-Pakete ein Acknowledgement erhalten und das sendende CAN-Modul nicht in einen Fehlerzustand übergeht.

5.3.3 Software

Das System wurde so eingerichtet, dass die CAN- und I^2C-Schnittstellen softwareseitig zur Verfügung stehen. Weiterhin wurde ein Programm implementiert, welches drei OUTPOST-Services (Service-Discovery sowie Ansteuerung von LCDs und PWM-kontrollierten Servomotoren per CAN) anbietet, die Schnittstellen CAN und Ethernet mittels eines Posix-Sockets anbindet und die I^2C-Schnittstelle mithilfe der Bibliothek WiringPi nutzbar macht.
Abbildung 35 zeigt das Blockdiagramm der für die Remote Unit entwickelten Software. Als

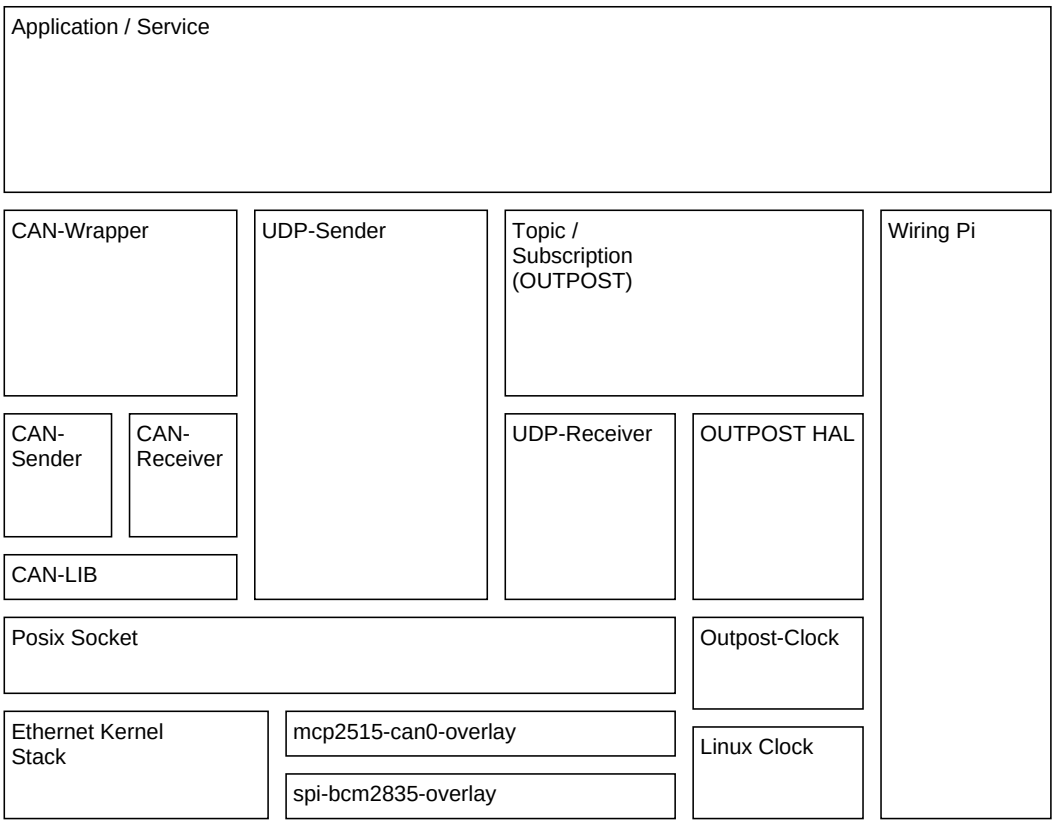


Abbildung 35: Blockdiagramm der Remote Unit-Software

entscheidendes Element ist der Remote-Unit-Service innerhalb des Blockes „Application / Service“ zu nennen. Er betreibt den Sensor-Thread, der Daten von per I^2C oder CAN angeschlossenen Sensoren und Aktoren (unter Benutzung der Library WiringPi) sammelt und

mithilfe der Klasse `net::Sender` zur GUI sendet. Er instanziiert einen CAN-Wrapper, der Abfragen per CAN erlaubt und empfangene Daten in das Telemetriepaket mit Sensor- / Aktordaten schreibt. Der Service selbst nimmt Telekommandopakete entgegen, die durch die Klasse `net::Receiver` empfangen wurden und mithilfe eines Topic/Subscriber-Patterns (implementiert durch die OUTPOST-Library) an die einzelnen Services weiterleitet. Der Remote-Unit-Service implementiert (zusammen mit dem darin instanziierten CAN-Wrapper) die Funktionalität des Service-Discovery-Protokolls.

Allgemeines zur Programmierung

Die Software der Remote Unit wurde in C++ geschrieben. Sie startet mehrere Threads, etwa zur Service Discovery oder zum Empfang von Datenpaketen per CAN oder UDP. Die in der `main.cpp` angelegte Variable *running* trägt Information darüber, ob die Software weiterhin laufen soll. Somit werden die Threads über eine Beendigung des Programms informiert. Ein Signal-Handler setzt die zum Programmstart auf den Wert 1 gesetzte Variable auf den Wert 0, sobald die Tastenkombination STRG-C gedrückt wird, sodass sich das Programm beendet.

UDP-Verbindung

Die UDP-Verbindung ermöglicht den Datenaustausch zwischen GUI und Remote Unit. Mithilfe der Klassen `net::Receiver` und `net::Sender` werden ankommende Telekommandos entgegengenommen und Telemetriedaten an die GUI gesendet. Die UDP-Verbindung ist so ausgelegt, dass sie Verbindungen per IPv4 behandeln kann.

UDP-Receiver

Die Klasse `net::Receiver` arbeitet ähnlich wie die entsprechende Klasse in der GUI, unterscheidet sich jedoch darin, dass für dieses Missionsszenario angenommen wird, dass nur eine GUI zur Zeit genutzt wird (umgekehrt ist es für die GUI wichtig, Daten von mehreren Remote Units empfangen zu können). Genutzt wird ein POSIX-Socket, der UDP-Pakete empfängt, die auf Port 32108 ankommen. Es wird ein Thread gestartet, der mithilfe von

```
n = recvfrom(sockfd, buf, BUFSIZE, 0, (struct sockaddr *) &clientaddr, &
    clientlen);
*mCcIpAdress = clientaddr.sin_addr.s_addr;
```

Daten empfängt, in einem Puffer zwischenspeichert, die Absender-IP-Adresse ermittelt und das Paket durch

```
outpost::BoundedArray<uint8_t> boundedArray(reinterpret_cast<unsigned char*>(
    buf), len);
outpost::pus::TelecommandWriter* telecommandWriter = new outpost::pus::
    TelecommandWriter(boundedArray);
```

```
outpost::pus::TelecommandReader tcReader(telecommandWriter->getPointer());
```

in ein Objekt vom Typ `outpost::pus::TelecommandReader` umwandelt und mithilfe von

```
outpost::pus::telecommandDistribution.publish(tcReader);
```

an die entsprechenden Services verteilt. Dem Receiver wird weiterhin eine Referenz auf eine `uint32_t`-Variable übergeben, die nach dem Empfang die IP-Adresse des UDP-Pakete sendenden Computers (typischerweise derjenige, auf dem die GUI ausgeführt wird) speichert, sodass diese für andere Klassen zur Verfügung steht.

UDP-Sender

Die Klasse `net::Sender` hat die Aufgabe, Telemetriepakete an die GUI zu senden. Deren IP-Adresse wird dem Sender per Referenz auf eine (vom Receiver gesetzte) Variable übergeben. Der Socket wird beim Erstellen des Objektes angelegt. Die Methode `void net::Sender::sendData(const outpost::pus::TelemetryHeader* telemetryHeaderData)` dient dazu, Telemetriepakete zu versenden. Hierbei wird durch

```
uint8_t buffer[outpost::spp::parameter::maximumTelemetryPacketLength];
outpost::Serialize* serializedPacked = new outpost::Serialize(buffer);
telemetryHeaderData->serializeWithApplicationData(*serializedPacked);
```

ein Puffer angelegt, in den die Daten des Telemetriepaketes kopiert werden, sodass dieser anschließend mithilfe von

```
ssize_t bytesWritten = sendto(mSocket, buffer,
        telemetryHeaderData->getLength(), 0,
        (struct sockaddr *) &destIpAddr, sizeof(destIpAddr));
```

an die GUI verschickt wird.

Die CAN-Verbindung

Der CAN-Transceiver ist wie in Unterunterabschnitt 5.3.2 beschrieben per SPI mit dem Raspberry Pi verbunden. Es gibt verschiedene Möglichkeiten, den CAN-Transceiver zur Kommunikation mit dem Raspberry Pi zu verwenden. Im Folgenden wird dargestellt, wie der CAN-Transceiver als Netzwerkinterface eingebunden werden kann, sodass eine Kommunikation per Socket möglich ist. Hierzu müssen innerhalb der Datei „/boot/config.txt“[16] Device-Tree-Overlays eingerichtet werden, welche die SPI-Schnittstelle für die Kommunikation zum CAN-Transceiver bereitstellen und den CAN-Transceiver als Netzwerggerät

[16] Um die Datei „/boot/config.txt“ zu editieren, werden Root-Rechte benötigt.

einbinden. Die Zeile

```
dtoverlay=spi-bcm2835-overlay
```

ermöglicht den Zugriff auf die SPI-Schnittstelle und

```
dtoverlay=mcp2515-can0,oscillator=8000000,interrupt=25
```

bindet den CAN-Transceiver als Netzwerkinterface ein. Dem Betriebssystem ist nun bekannt, dass dieser mit einem 8 MHz-Quarz ausgestattet ist (ebenfalls gebräuchlich ist die Variante mit einem 16 MHz-Quarz) und die Interruptleitung am GPIO-Pin 25 bedient, wenn eingehende Daten abgeholt werden sollen. Nach einem Neustart kann der CAN-Transceiver als Netzwerkinterface hinzugefügt, indem innerhalb der Shell folgender Befehl eingegeben wird:

```
sudo ip link set can0 up type can bitrate 500000
```

Dieses startet eine Netzwerkverbindung des mit `can0` bezeichneten CAN-Transceivers (Die Bezeichnung entspricht dabei der aus der confix.txt), welcher auf eine Bitrate von 500kbps eingestellt wird. Aufgrund der kurzen Leitungslänge von ca. 0,5m ist diese möglich; bei längeren Leitungswegen müsste die Bitrate an dieser Stelle (und in der CAN-I^2C-Bridge) verringert werden. Die Netzwerk-Verbindung wird hierbei bis zu einem Neustart bereitgestellt. Entweder wird der eben genannte Befehl hiernach erneut eingegeben (oder in einem Boot-Skript ausgeführt) oder das Herstellen der Netzwerkverbindung wird durch einen Eintrag in der Datei „/etc/network/interfaces“ automatisiert. Hierzu muss diese Datei um die Zeilen

```
auto can0
iface can0 can static
        bitrate 500000
```

ergänzt werden.

CAN-Wrapper

Die Kommunikation der Remote-Unit-Software per CAN-Interface funktioniert, indem mithilfe eines Sockets Daten gesendet und empfangen werden. Ebenso wie bei der UDP-Schnittstelle wird die Funktionalität zum Senden und zum Empfangen in zwei separaten Klassen `net::CanReceiver` und `net::CanSender` gekapselt. Der CAN-Receiver erstellt zum Empfangen einen eigenen Thread.
Da die Abfrage der per CAN angeschlossenen Sensoren (und Aktoren) nach dem Schema

1. Sende Anfragepaket
2. Empfange und verarbeite Antworten

abläuft, wird die Ausführung dieser Schritte mithilfe der Klasse `wrapper::CanWrapper` gekapselt. Hier befindet sich die Methode `void CanWrapper::pollCan(outpost::Serialize& pSerialize)`, die eine Anfrage mithilfe des CAN-Senders schickt und anschließend eingehende CAN-Pakete[17] entgegennimmt und deren Daten in ein Objekt des Typs `outpost::Serialize` schreibt. Die Nutzung dieses Datentyps bietet sich an, da hieraus mit wenig Aufwand die Application-Data des ausgehenden Telemetriepaketes generiert werden kann und eine Schnittstelle zum Eintragen etwa von `double`- und `uint_X`-Werten vorhanden ist.

Mithilfe einer for-Schleife werden alle eingegangenen Pakete verarbeitet, indem der innerhalb der CAN-ID kodierte Sensortyp bestimmt, die ebenfalls dort kodierte CAN-I^{2}C-Bridge-ID ausgelesen und diese Daten zusammen mit dem gemessenen Sensorwert in das Telemetriepaket geschrieben werden.

CAN-Receiver

Der CanReceiver startet einen Thread, der per CAN eingehende Pakete entgegennimmt, die vom CanWrapper abgefragt werden. Hierzu wird mit

```
int s = socket(PF_CAN, SOCK_RAW, CAN_RAW);
```

ein Socket angelegt, der alle CAN-Pakete entgegennimmt. Innerhalb einer Schleife werden mithilfe des Befehls

```
int retval = recvmmsg(s, msgs, maxInterfaces, 0, &timeout);
```

innerhalb eines Timeouts von 50 ms bis zu 256 Pakete empfangen und in einer Datenstruktur abgelegt. Diese wird anschließend ausgelesen und innerhalb einer Schachtelung von Vektoren (`std::vector<std::vector<uint8_t>>`) gespeichert, sodass die CAN-Pakete vom CAN-Wrapper weiterverarbeitet werden können.

CAN-Sender

Der CAN-Sender hat die Aufgabe, CAN-Nachrichten zu senden, wenn die Methode `CanSender::sendPacket(uint32_t id, uint8_t* data, uint8_t length)` aufgerufen wird. Hierzu wird ebenfalls ein Socket angelegt (durch dieselbe Instruktion wie im CAN-Receiver), auf den mithilfe von

```
write(s, &frame, required_mtu)
```

geschrieben werden kann.

[17] Tatsächlich wird per loopback auch das gesendete Anfragepaket empfangen. Dieses wird ignoriert.

Service und Application

Ein OUTPOST-Service muss von einer Application beinhaltet werden. Diese Application wurde implementiert, indem eine Klasse RemoteUnitApplication implementiert wurde, die innerhalb ihres Konstruktors ihre Super-Klasse aufruft und somit für die Verteilung eingehener Telekommandos registriert wird.

```
RemoteUnitApplication::RemoteUnitApplication(uint16_t apid, uint8_t
    serviceType,
    outpost::time::Clock& clockIn, const char* applicationName) :
    outpost::pus::Application(apid, clockIn, "RemoteUnitApplication"),
        mRemoteUnitService(
        serviceType, this), mCanLcdService(CANLCDST, this), mCanPwmService(
        CANPWMST, this), mGenericCanActorService(
        GENERIC_CAN_ACTR_SERVICE, this)
```

Über die Initialisierung im Konstruktor werden auch die Services `RemoteUnitService`[18], `CanPwmService` und `CanLcdService` instanziiert. Der RemoteUnitService stellt die eigentliche Funktionalität des Service-Discovery-Protokolls her. CanPwmService und CanLcdService dienen der Ansteuerung von per CAN-I^{2}C-Bridge angeschlossenen LC-Displays und PWM-gesteuerten Servomotoren; der GenericActorService sendet beliebige CAN-Nachrichten an CAN-I^{2}C-Bridges, beispielsweise um die Echtzeituhr zu stellen.

Remote-Unit-Service

Die Klasse `service::RemoteUnitService` dient der Service-Discovery, indem angeschlossene Sensoren und Aktoren erkannt, ggf. ausgelesen und die GUI hierüber per Telemetriepaket informiert wird. Ebenfalls wird eine Anfrage zur Service-Discovery auf Netzebene („DiscoveryCommand“) durch das Senden von zwei Acknowledgements beantwortet. Ähnlich wie die Application führt auch der Remote-Unit-Service einen Aufruf seines Super-Konstruktors aus. Hierüber ist auch definiert, dass der Service die Methode `executeCommand` implementieren muss, in der die Abarbeitung eines Telekommandos stattfindet. Innerhalb derselben findet sich eine switch-case-Verzweigung, die Telekommandos anhand ihres Service (Sub-)Types separiert.

```
RemoteUnitService::RemoteUnitService(uint8_t serviceType,
    outpost::pus::Application* applicationIn) :
    outpost::pus::Service(serviceType, "Raspi Service", applicationIn),
        mServiceType(
        serviceType),  mThread(nullptr)
```

[18] Das vorangestellte „m“ dient der Kennzeichnung, dass es sich bei diesen Objekten um Members der Klasse handelt.

Ein eingehendes Telekommando wird entsprechend seines Service Types und Service Subtypes verarbeitet. Hierbei wird entweder eine Antwort auf ein (Netzwerk-) Discovery-Telekommando geschickt, der Sensor-Thread gestartet bzw. gestoppt oder ein Aktor angesprochen.

```
switch (command.getServiceSubType()){
  case 128: //just reply ACK to CC; for Network-level Service Discovery
  ...
  case 132: //stop Sensor Thread
  ...
  case 133: //print to I2C-LCD
  ...
  case 134: //Control I2C-Servo
  ...
  case 135: //start Sensor Thread
  ...
}
```

Sensor-Thread

Innerhalb des Remote Unit-Services wird mithilfe eines Telekommandos ein Thread gestartet, der in einer Schleife Sensoren und Aktoren am I^2C- sowie am CAN-Bus abfragt und Telemetriepakete generiert, die an die GUI gesendet werden. Mithilfe eines Telekommandos [128,135][19] wird ein Thread gestartet, der den CAN-Bus sowie den I^2C-Bus abfragt und Sensor-/Aktordaten einsammelt, welche daraufhin an die GUI gesendet werden.
Mithilfe des Telekommandos [128,133] wird auf ein LCD geschrieben, das per I^2C direkt an die Remote Unit angeschlossen ist. LCDs, die per CAN-I^2C-Bridge mit der ID `id` angeschlossen sind, werden mithilfe des Telekommandos [129,`id`] gesteuert. Analog dazu werden Servomotoren mithilfe der Kommandos [128,134] bzw. [130,`id`] angesteuert.

Zur Abfrage von Sensordaten werden innerhalb des Threads zunächst die CAN-Geräte abgefragt und anschließend alle I^2C-Adressen überprüft, ob ein Byte gelesen werden kann. Bei einem Erfolg (es konnte ein Byte gelesen werden) wird anhand der Adresse durch die Klasse `AdressTypeConverter` bestimmt, um welchen Sensortyp es sich handelt. Anschließend werden - sensorspezifisch - Daten ausgelesen und zusammen mit dem Sensortyp im Telemetriepaket verpackt. Dieser Ablauf ist in Listing 4 dargestellt.
Es ist zu erkennen, dass die Sensoren hier unterschiedlich behandelt werden. Während der Temperatursensor seine Daten durch das Auslesen des ersten Registers (0) preisgibt, werden für die anderen Sensoren Bibliotheken genutzt, die die Sensoren für die Messung durch Setzen

[19] Diese Schreibweise dient als Kurzform von Service Type 128, Service Subtype 135.

Algorithm 4: RemoteUnitService::runThread Thread zur Abfrage von Sensordaten

```
1  Read Data from CAN-Bus and write to telemetry Packet
2  for adress ← 0 to 127 do
3      int value = read Byte from I²C-Sensor with adress
4      uint8_t sensorType = AdressTypeConverter.getSensorTypeByAdress(adress)
5      switch sensorType do
6          case Sensortype::TEMPERATURE do
7              store Byte-Value SensorValue::TEMPERATURE to telemetry
8              store Byte-Value value to telemetry
9          case Sensortype::HUMIDITY_TEMPERATURE do
10             double temp = getTemperature(file) //file points to I²C-Device
11             double humi = getHumidity(file)
12             store Byte-Value SensorValue::TEMPERATURE2 to telemetry
13             store Double-Value temp to telemetry
14             store Byte-Value SensorValue::HUMIDITY to telemetry
15             store Double-Value humi to telemetry
16         case Sensortyp::ACCELEROMETER do
17             init ADXL345-Object acc
18             wait 100ms
19             uint16_t ax, ay, az
20             acc.read(ax, ay, az)
21             store Byte-Value SensorValue::ACCELEROMETER to telemetry
22             store uint16_t-Value ax to telemetry
23             store uint16_t-Value ay to telemetry
24             store uint16_t-Value az to telemetry
25         ...
26 send Telemetry Packet
```

spezieller Register initialisieren und wiederum spezifische Register auslesen, um an die Sensordaten zu gelangen.
Aufgrund der Annahme, dass Sensoren während des Betriebes getrennt und verbunden werden können, muss diese Initialisierung bei jedem Durchlauf erfolgen. Ein Speichern der beim letzten Durchgang vorhandenen Sensortypen und der Verzicht auf eine erneute Initialisierung ist nicht sinnvoll, da das Messintervall mit einer Sekunde groß genug ist, um einen Sensor kurz zu trennen und erneut zu verbinden. Eine vorher geschehene Initialisierung wäre somit hinfällig.

CAN-Services

Damit eingehende Daten an einen per CAN-I^{2}C-Bridge angeschlossenen Aktor übertragen werden können, müssen diese (wie in Unterabschnitt 7.4: CAN-Datenübertragungsprotokoll beschrieben) zunächst innerhalb eines oder mehrerer CAN-Pakete an die CAN-I^{2}C-Bridge übertragen werden. Für den CAN-PWM-Service (zur Ansteuerung des PWM-kontrollierten Servomotors) wird nur ein Byte mit der Information über die gewünschte Auslenkung übertragen. Damit Daten auf dem LCD angezeigt werden können, müssen diese auf mehrere CAN-Pakete verteilt werden. Dies ergibt sich aus der maximalen Nutzdatengröße eines CAN-Paketes von acht Bytes sowie einer (in dieser Arbeit verwendeten) Displaygröße von $2 \cdot 16$ Zeichen. Die Übertragung der in der CAN-I^{2}C-Bridge zwischengespeicherten Daten an das LCD wird mithilfe eines weiteren Paketes initiiert.
Wie in Listing 5 zu erkennen ist, wird beim Empfang eines Telekommandos ein Puffer angelegt, der anzuzeigende Text zwischengespeichert und anschließend innerhalb von vier Paketen an die CAN-I^{2}C-Bridge gesendet. Die CAN-IDs dieser Pakete wurden so gewählt, dass eindeutig gekennzeichnet ist, dass es sich um Pakete zur Anzeige auf einem LCD handelt (ID-Präfix 0x200000), welches der vier textenthaltenden Datenpakete gemeint ist (Infixe 0x**0**0000, 0x**1**0000, 0x**2**0000, 0x**3**0000) und für welche ID das Paket bestimmt ist (niederwertigstes Byte). Analog ergibt sich das Kommando zur Anzeige auf dem Display, jedoch mit Infix 0x**F**0000 und ohne Nutzdaten.

Algorithm 5: CanLcdService::executeCommand Verarbeitung eines Telekommandos zur Anzeige von Text auf einem per CAN-I^{2}C-Bridge angeschlossenen LCD

Input: Incoming Telemetry Packet tm

```
1 id ← 0x200000+tm.serviceSubType
2 char displayBuffer [32] ← {0}
3 memcpy(displayBuffer, tm.ApplicationData, tm.ApplicationDataLength)
4 canSender.sendPacket(id + 0x00000, &displayBuffer[0], 8)
5 canSender.sendPacket(id + 0x10000, &displayBuffer[8], 8)
6 canSender.sendPacket(id + 0x20000, &displayBuffer[16], 8)
7 canSender.sendPacket(id + 0x30000, &displayBuffer[24], 8)
8 canSender.sendPacket(id + 0xF0000, NULL, 0)
```

6 Sensorschnittstelle

In diesem Abschnitt werden die Kommunikation per I^2C sowie die verwendeten Sensoren und Aktoren beschrieben und der mechanische Aufbau der genutzten Sub-D-Schnittstelle dargestellt.

6.1 Grundlagen

Zunächst werden Grundlagen zur Kommunikation per I^2C sowie verwendete Sensoren und Aktoren vorgestellt.

6.1.1 I^2C

I^2C, kurz für IIC bzw. inter-integrated-circuit, beschreibt ein Kommunikationsprotokoll, welches für die in dieser Arbeit genutzten Sensoren verwendet wird. Der Vorteil von I^2C ist die einfache Handhabung durch die Nutzung von nur zwei Leitungen sowie die Kostengünstigkeit der verwendeten Sensoren.
I^2C verwendet zwei Leitungen, SDA für die Datenübertragung und SCL als Taktleitung. Es gibt einen Master auf dem Bus sowie mehrere mögliche Slaves. Der Master kommandiert hierbei die Slaves. Ein I^2-Gerät verfügt über eine Adresse, über die es angesprochen werden kann. Ein Bit der Adresse wird verwendet, damit der Master dem Slave mitteilen kann, ob es sich um eine Anforderung zum Lesen oder Schreiben handelt. Bei einer 8-Bit-Adressierung (es ist auch möglich, 10-Bit-Adressen zu verwenden) gibt es somit jeweils 128 Adressen für Lese- und Schreibzugriffe, von denen jedoch 16 für andere Zwecke verwendet werden, etwa um eine 10-Bit-Adressierung zu kennzeichnen (vgl. [rau]). Die Steuerung von I^2C-Geräten erfolg über das Modifizieren und Auslesen von Registereinträgen. Hierbei wird bei einem Schreibzugriff zunächst die Nummer des Registers übertragen, bevor dessen Sollwert gesendet wird. Ebenso erfolgt bei einem Lesezugriff die Angabe eines Registers, aus dem gelesen werden soll.

6.1.2 Sensoren

Um Sensordaten zu erhalten, werden die in dieser Arbeit genutzten Sensoren per I^2C angesprochen und abgefragt. Die Verwendung dieser Schnittstelle bietet kostengünstig die Möglichkeit, verschiedene Sensoren etwa zur Messung von Temperatur, Luftfeuchte usw. einzusetzen.

LM75A

Der LM75A ist ein digitaler Sensor mit I^2C-Schnittstelle, der Temperaturen mit einer Genauigkeit von 11 Bit (entspricht Messschritten von 0,125°C) misst. Dieses geschieht mit

einer Unsicherheit vom ± 2°C im Bereich zwischen -25°C und 100°C. Der Sensor ist in Abbildung 36 zu erkennen. Die Versorgungsspannung beträgt 2,8 - 5,5V (vgl. [NXP07]).

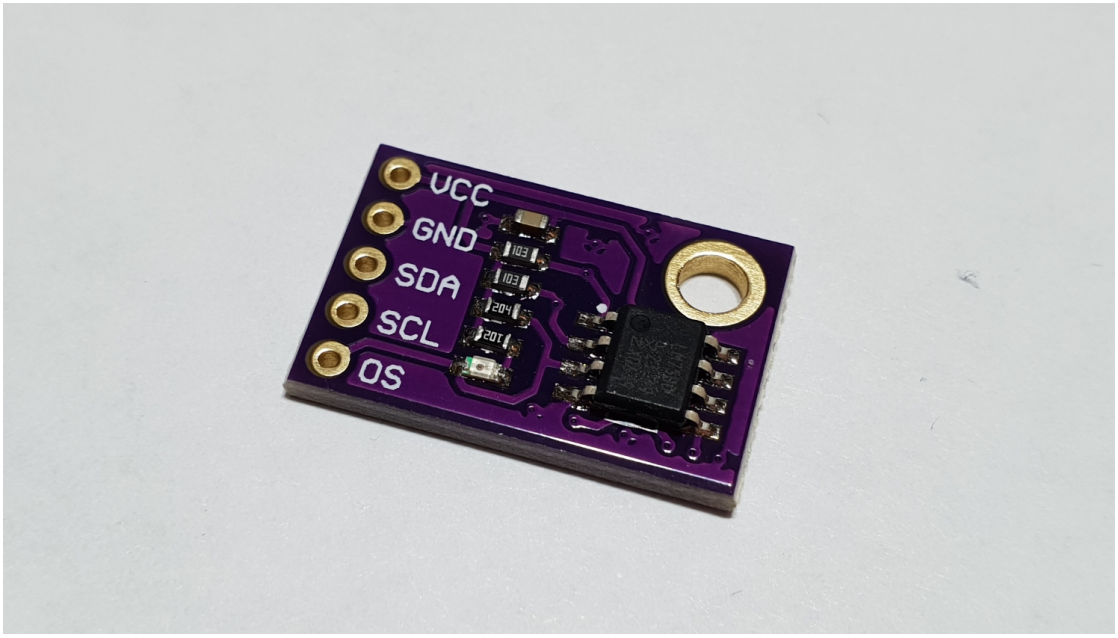


Abbildung 36: Temperatursensor LM75A
Quelle: Eigenes Bild.

ADXL345

Der ADXL345 ist ein Drei-Achsen-Accelerometer mit einem Messbereich von ± 16g. Seine Auflösung liegt bei 10 Bit und er benötigt bei einer Versorgungsspannung zwischen 2,0V und 3,6V 23 μA bei Messungen und 0,1 μA im Standby (vgl. [DEV14]). Der ADXL345 ist in Abbildung 37 abegebildet.

BMP280

Der BMP280 (vgl. [Sen18]) ist ein Sensor, um den Druck der Umgebungsluft zu messen. Wie in Abbildung 38 zu erkennen, besitzt er einen Metallkörper mit einer kleinen Öffnung, sodass sich Druck der Umgebungsluft messen lässt und der eigentliche Sensor geschützt bleibt.

Er kann Drücke im Bereich zwischen 300 hPa und 1100 hPa mit einer Genauigkeit von ±1,7 hPa messen und benötigt bei einem Messintervall von 1s und einer Versorgungsspannung von 1,71 V - 3,6 V 2,7 μA (Durchschnittsverbrauch; Stromspitzen von bis zu 1,12mA).

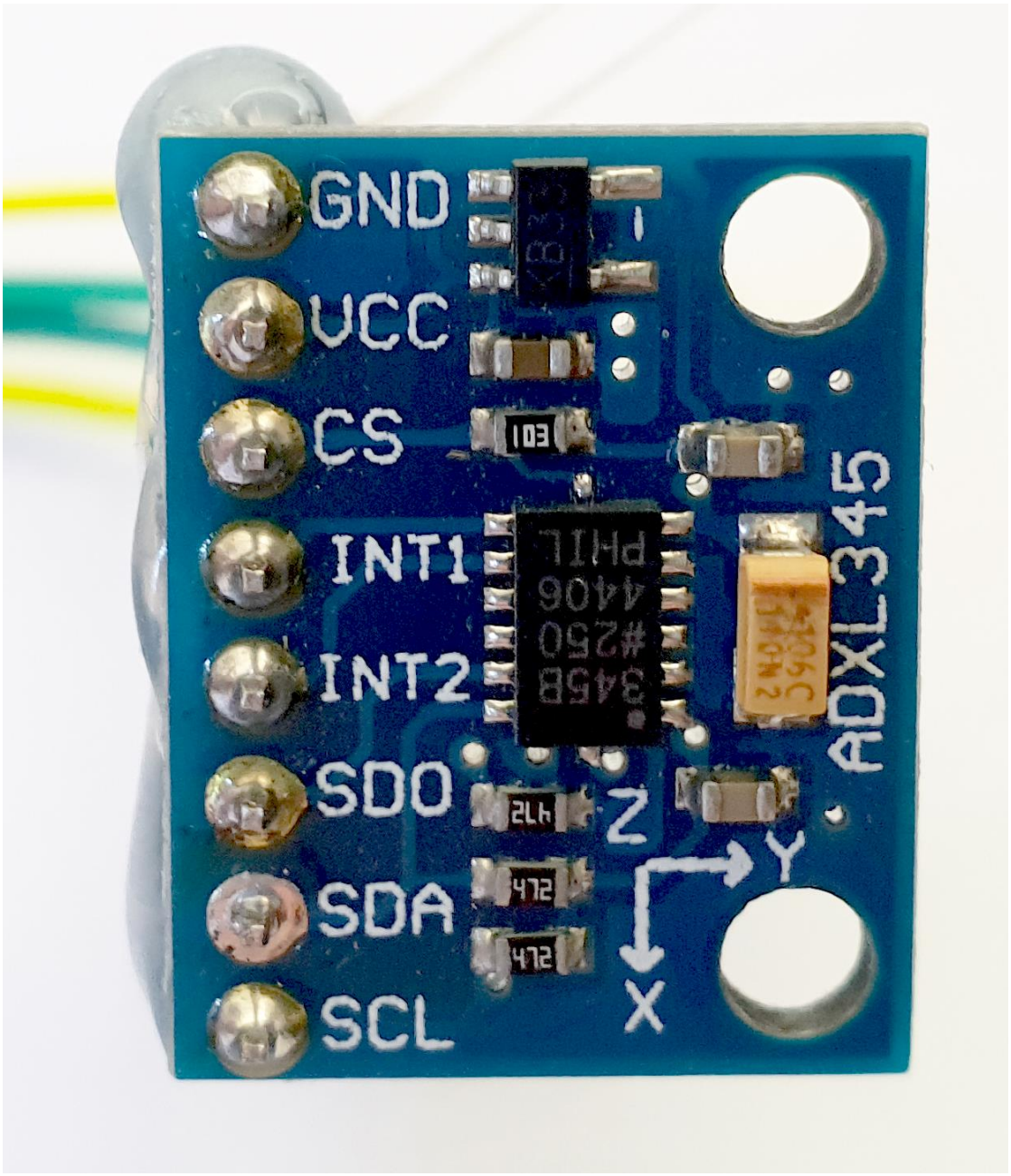


Abbildung 37: Drei-Achsen-Accelerometer ADXL345
Quelle: Eigenes Bild.

TSL2561

Der Sensor TSL2561 dient der Messung der Beleuchtungsstärke. Hierzu ist, wie in Abbildung 39 zu erkennen, ein IC verwendet worden, welcher die Messung durchführt, das Signal

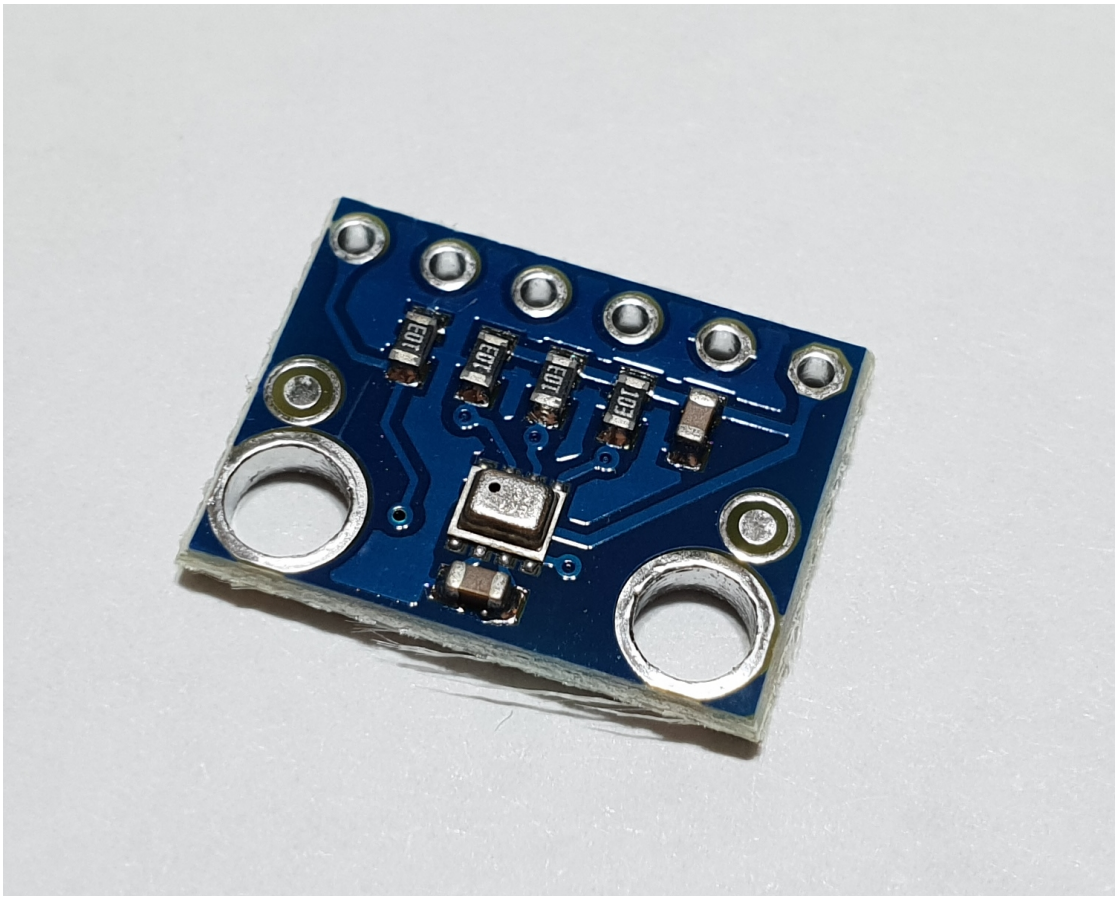


Abbildung 38: Drucksensor BMP280
Quelle: Eigenes Bild.

digitalisiert und ggf. verstärkt und als Messwert am I^2C-Bus zur Verfügung stellt. Er benötigt eine Versorgungsspannung zwischen 2,7 und 3.6V

Er besitzt zwei Sensoren: Einer misst die Helligkeit innerhalb des sichtbaren sowie infraroten Spektrums, der zweite misst nur infrarotes Licht. Empfindlichkeitsmaxima liegen bei ca. 650nm respektive 810nm. Zur Messung werden Messwerte mithilfe des integrierten Analog-Digital-Konverters innerhalb von Zeitspannen von 13,7 ms, 101 ms oder 402 ms integriert. Er besitzt die Möglichkeit, auf Kommando eine 16-fache Verstärkung einzuschalten, um bei geringen Beleuchtungsstärken Messungen durchzuführen (vgl. [ada18]).

Tiny RTC

Tiny RTC bietet die Funktionen einer Echtzeituhr (im eigentlichen Sinne ist dies also kein

Abbildung 39: Beleuchtungsstärkensensor TSL2561
Quelle: Eigenes Bild.

Sensor, wird jedoch im Folgenden so behandelt). Sie benutzt den IC DS1307 und verwendet eine Knopfzelle zur Sicherstellung der Zeitmessung ohne externe Stromversorgung.

Abbildung 40 zeigt die Tiny RTC. Nicht abgebildet ist die auf der Rückseite der Platine befindliche Knopfzelle. Sie benötigt eine Versorgungsspannung von 5V. Per I^2C kann die intern gespeicherte (und fortlaufende) Uhrzeit eingestellt und abgefragt werden. Die Auflösung der Echtzeituhr beträgt eine Sekunde (vgl. [Ele14]).

6.1.3 Aktoren

Im Folgenden werden die Aktorenkomponenten beschrieben, die innerhalb dieser Master Thesis genutzt werden.

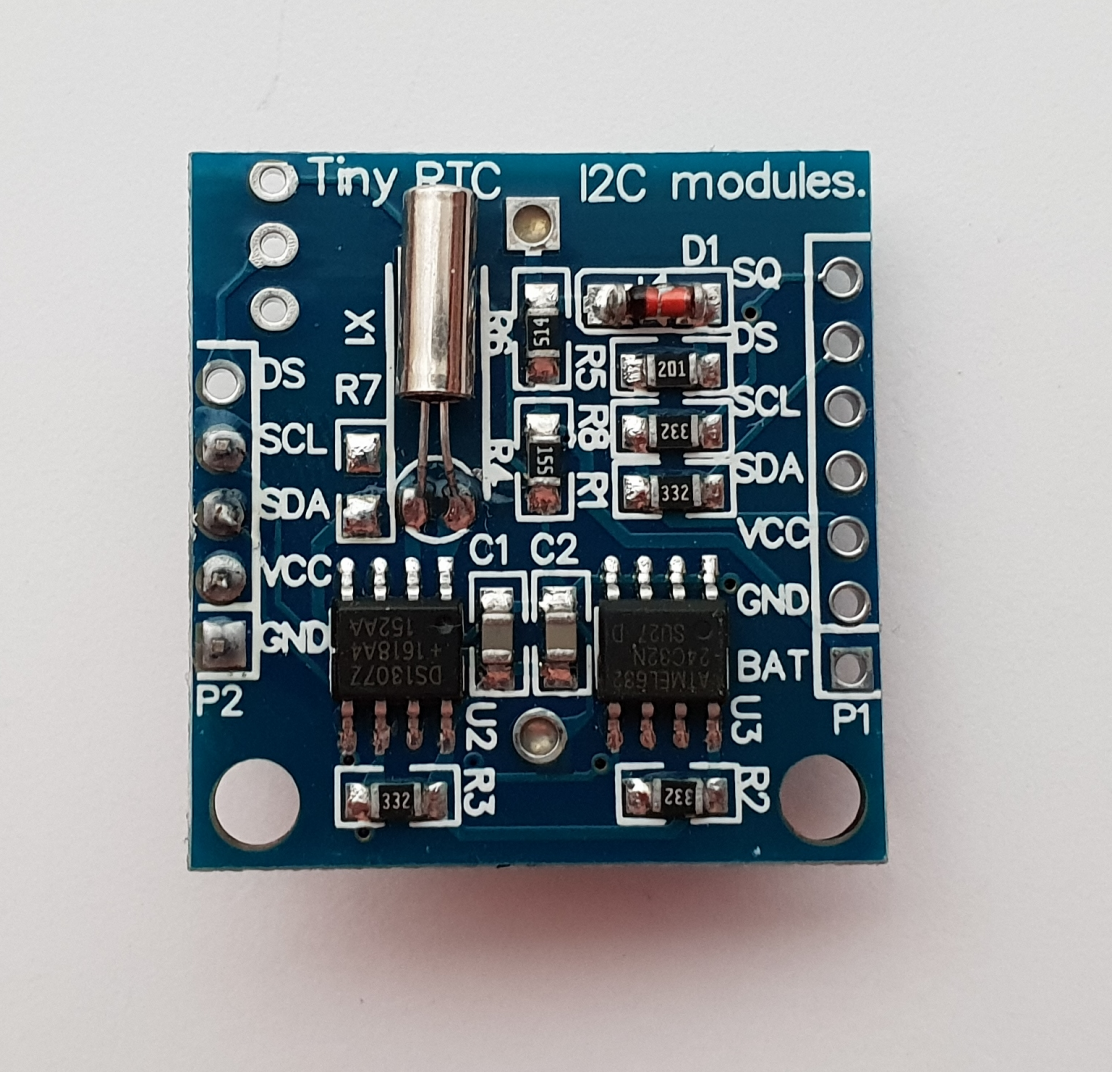


Abbildung 40: Echtzweit-Uhr Tiny RTC
Quelle: Eigenes Bild.

PCA9685

Der PCA9685 (vgl. [NXP15]) ist ein Schnittstellen-IC zur Generierung von PWM-Signalen. Er wird per I^2C (mit bis zu 1MHz) angesteuert und besitzt 16 Ausgänge, die sich einzeln oder gemeinsam mit einer Genauigkeit von 12 Bit (4096 Stufen) ansteuern lassen. Bei 5V und einer Nutzung als Stromquelle führt er 10 mA und als Stromsenke 25mA zu. Per PWM lassen sich hiermit beispielsweise LEDs dimmen, Servomotoren kontrollieren oder Lüfter steuern. Zur Speisung der Ausgangssignale steht ein dedizierter Anschluss für die Versorgungsspannung des angeschlossenen Aktors zur Verfügung, die unabhängig von der

Versorgungsspannung des ICs (2,3 - 5,5V) ist. Die Frequenz des PWM-Signals ist zwischen 24 und 1526Hz einstellbar, wobei diese für alle Ausgänge stets gleich ist. Abbildung 41 zeigt die Platine mit IC und Anschlüssen. Auf der rechten Seite sind entfernbare Widerstände erkennbar, mit denen sich 62 verschiedene I^2C-Adressen einstellen lassen (2^6 abzüglich zwei reservierter Adressen).

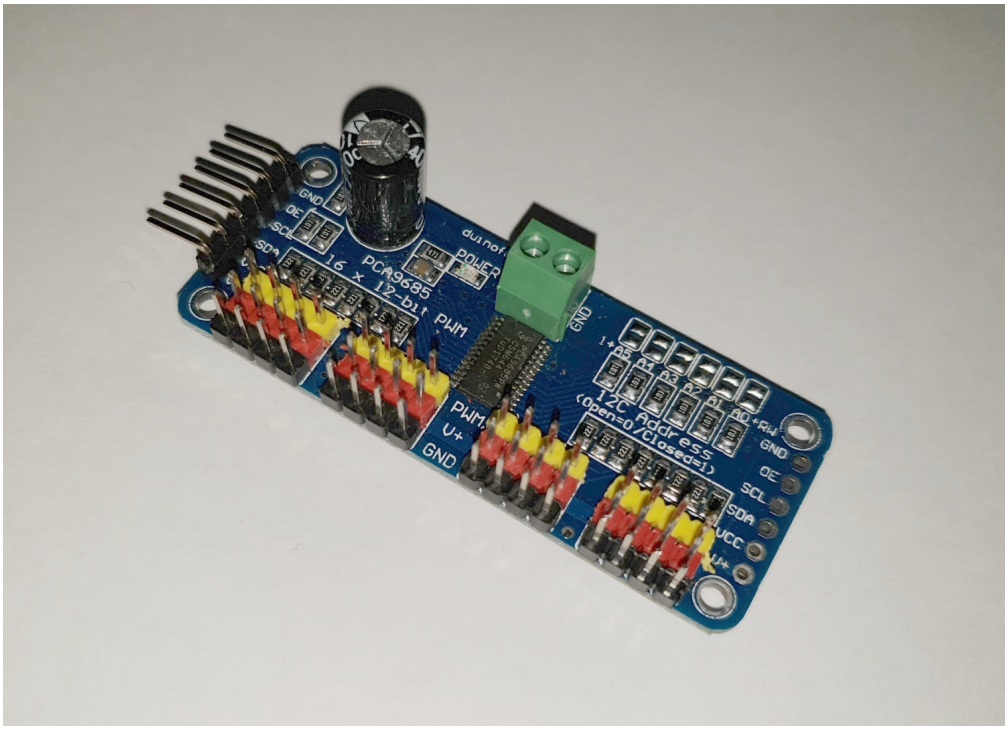


Abbildung 41: PCA9685 auf einer Platine mit Anschlüssen für 16 Servomotoren
Quelle: Eigenes Bild.

Servomotor SG90

Der SG90 (zu erkennen in Abbildung 42) ist ein per PWM gesteuerter Servomotor mit einem Gewicht von 14,7g, Abmessungen von 32 · 12 · 32 mm (L · B · H) und einem maximalen Drehmoment von 0,245Nm sowie einer Versorgungsspannung von 4,8 - 6V (vgl. [Unb14]). Er wird per Pulsweitenmodulation (PWM) angesteuert und kann sich um ca. 180° drehen (90° von der Mittelposition in jede Richtung)).

LCD

Das in dieser Arbeit verwendete LC-Display kann 16 × 2 Zeichen anzeigen und besitzt eine

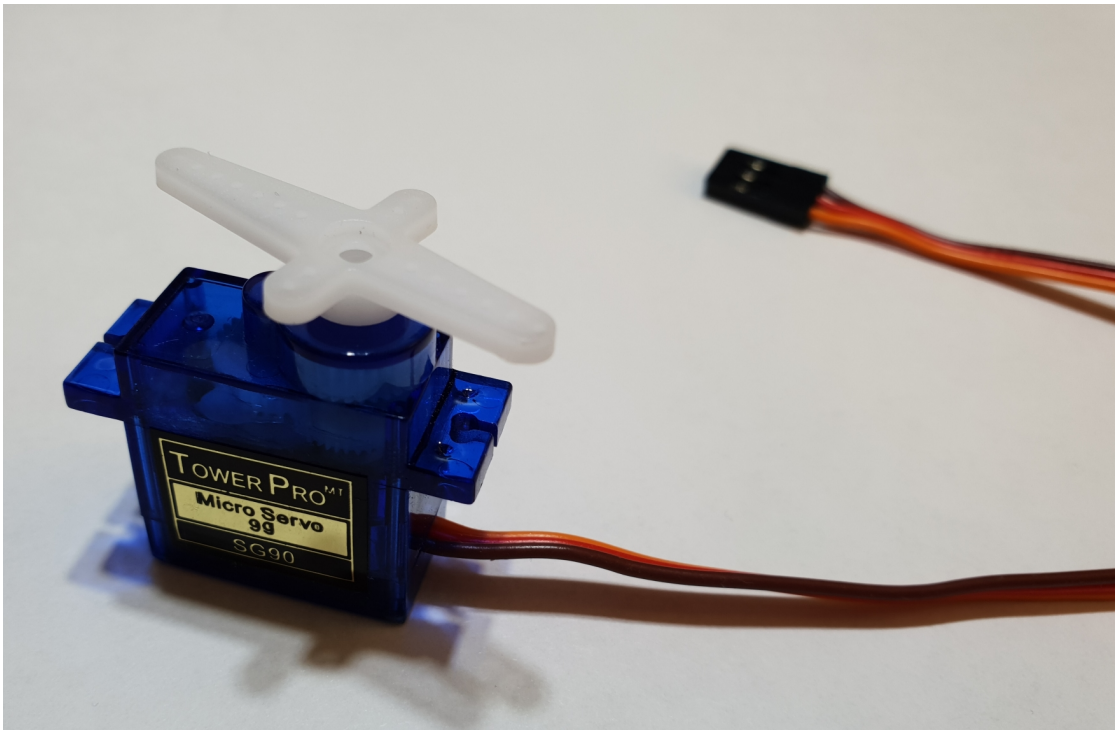


Abbildung 42: PWM-gesteuerter Servomotor SG90
Quelle: Eigenes Bild

Hintergrundbeleuchtung. Da zur Ansteuerung des Displays nur vier Datenbits pro Teilbefehl gesetzt werden müssen, bleiben bei Verwendung eines PCF8574 noch vier weitere Bits zur Steuerung. Diese werden genutzt, um das Schreiben in ein Register zu ermöglichen oder die LED der Hintergrundbeleuchtung zu schalten (vgl. [als16]).

Abbildung 43 zeigt das LCD von der Vorder- und Rückseite. Unten im Bild ist die angelötete Platine mit PCF8574 zu erkennen.

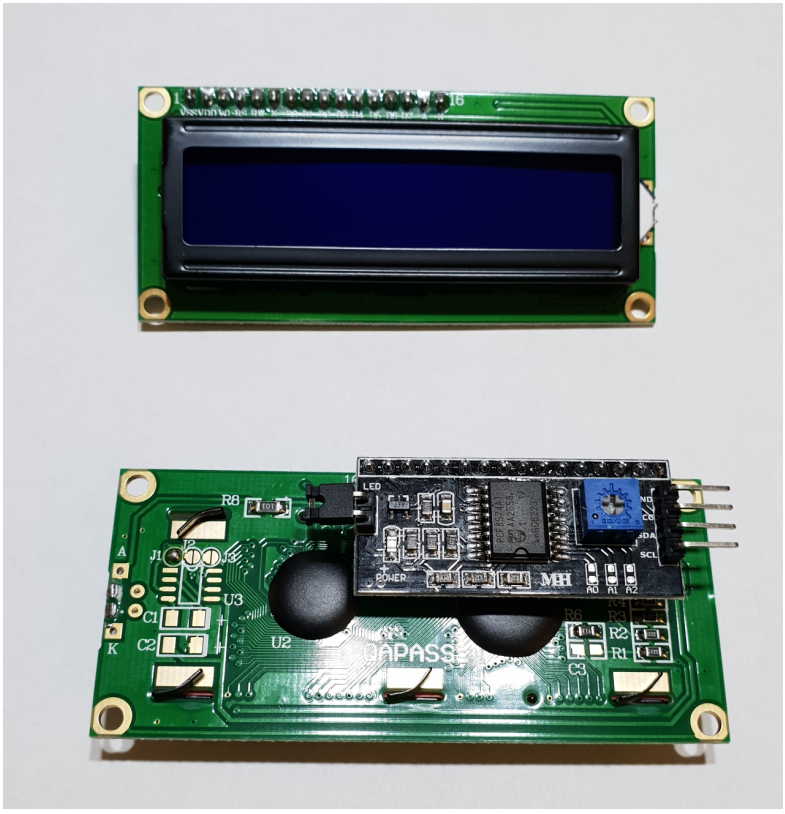


Abbildung 43: 16 ×2-LCD
Quelle: Eigenes Bild

6.2 Mechanische Sensorschnittstelle

Die in dieser Arbeit verwendeten Sensoren besitzen eine I^2C-Schnittstelle und benötigen eine Versorgungsspannung von 3,3V oder 5V. Um dieses bereitzustellen (und darüber hinaus Zugang zum CAN-Bus für die im Abschnitt 7 beschriebene CAN-I^2C-Bridge), verfügt jede RU über fünf Sub-D-Buchsen, über die eben genannte Leitungen herausgeführt werden. Es wurde an dieser Stelle eine Buchse (und kein Stecker) verwendet, um einen möglichen Kurzschluss durch offenliegende, spannungsführende Leitungen zu vermeiden. Abbildung 44 zeigt die Anschlussbelegung der Buchse (sowie spiegelverkehrt die des Steckers).

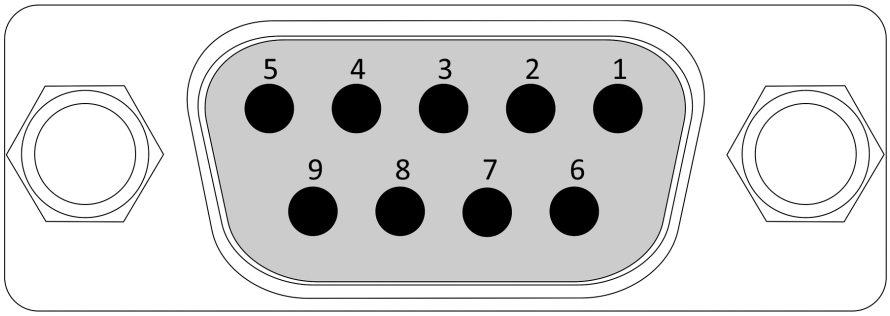


Abbildung 44: Sicht auf die Buchse des D-Substeckers
Quelle: [wul14], ergänzt um Nummerierung der Pins

Die in Abbildung 44 abgebildete Buchse verfügt über folgende Pinbelegung:

1: CAN Low

2: Masse

3: +3,3V

4: Masse

5: CAN High

6: Unbenutzt

7: SCL

8: SDA

9: +5V

Die Beschränkung auf acht Leitungen wurde festgelegt, um Ethernet-Kabel verwenden zu können.

7 CAN-I^{2}C-Bridge

In diesem Kapitel wird dargestellt, welche Aspekte die Nutzung einer CAN-I^{2}C-Bridge sinnvoll machen und wie diese implementiert ist.

7.1 Problemstellung

Wird versucht, wie in Abbildung 45 gezeigt, mehrere gleichartige Sensoren an die Remote Unit anzuschließen, ergeben sich Probleme aufgrund der nicht oder nur eingeschränkt einstellbaren Adresse des Gerätes und den daraus resultierenden Adresskonflikten. Es ist also nicht möglich, beliebig viele gleichartige Sensoren gleichzeitig zu betreiben.

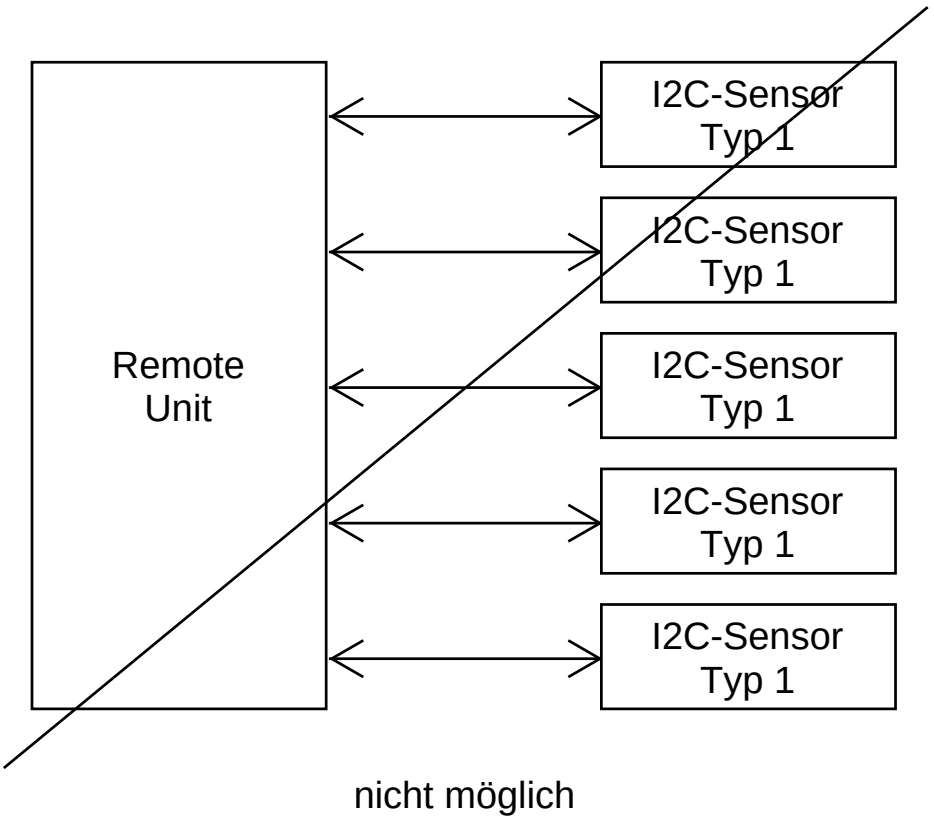


Abbildung 45: (Nichtmöglicher) Anschluss mehrerer gleichartiger Sensoren per I^{2}C

7.2 Lösungsansatz

Um das beschriebene Problem zu lösen, wurde eine Bridge zwischen Remote Unit und I^{2}C-Sensor entwickelt (siehe Abbildung 46). Diese soll von der tatsächlichen I^{2}C-Adresse abstrahieren und die Datenabfrage mithilfe einer einstellbaren ID ermöglichen. Zur Kommunikation mit der Remote Unit wurde das Controller Area Network ausgewählt, da es robust ist, entsprechende Transceivermodule günstig zu erwerben sind und sich die Kommunikationsschnittstelle mit geringem Aufwand in die Software der Remote Unit integrieren lässt. Darüber hinaus lässt CAN Broadcasts zu[20], sodass mithilfe eines einzigen gesendeten Paketes alle angeschlossenen Sensoren angesprochen und um Zusendung der Messdaten gebeten werden können. Die Bezeichnung „CAN-I^{2}C-Bridge“ ergibt sich aus den verwendeten Kommunikationsprotokollen.
Damit mehrere Sensoren eines Typs angeschlossen und unterschieden werden können, verfügt jede CAN-I^{2}C-Bridge über eine vom Benutzer (per DIP-Schalter) einstellbare ID, im Folgenden „Bridge-ID“ genannt.

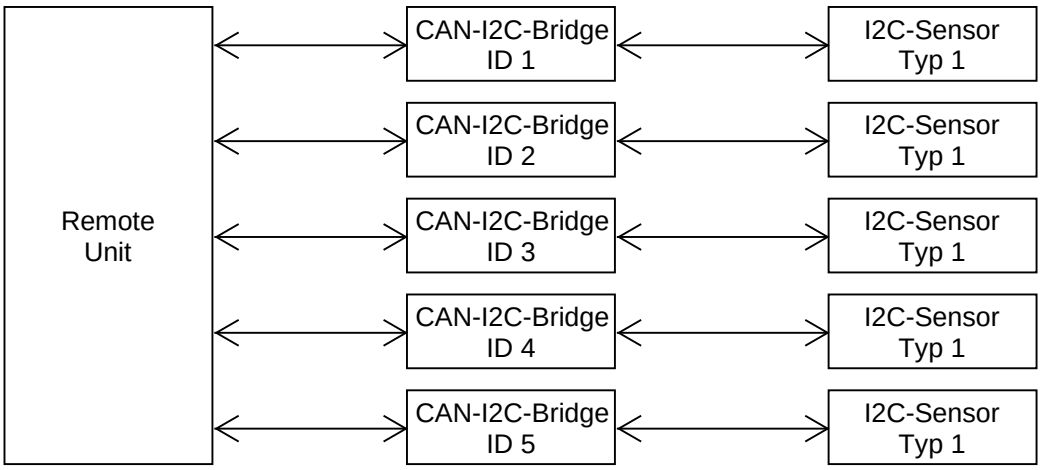


Abbildung 46: Anschluss mehrerer gleichartiger Sensoren per CAN-I^{2}C-Bridge

7.3 Grundlagen

In diesem Abschnitt werden die Grundlagen für die Konstruktion der CAN-I^{2}C-Bridge dargelegt. Hierbei wird auf den Arduino Nano sowie den I^{2}C-Port-Expander PCF8574 Bezug genommen. Das CAN-Board mit MCP2515 wurde bereits in Unterunterabschnitt 5.2.4 beschrieben.

[20] Genaugenommen sind alle CAN-Nachrichten Broadcasts.

7.3.1 Arduino Nano

Der Arduino Nano (Version 3.0) ist ein Mikrocontroller-Board, welches mit einem ATMEGA328P ausgestattet ist. Er besitzt insgesamt 14 digitale Ein- und Ausgänge (sechs bieten einen PWM-Ausgang an, der Logik-Level beträgt bei allen digitalen 5V) sowie acht analoge Eingänge. Mithilfe eines internen, standardmäßig deaktivierten Pull-Up-Widerstands von 20-50kΩ, können seine Pins bei Nichtanliegen einer externen Spannung auf den HIGH-Logikpegel von 5V gezogen werden. Zwei Pins können zur externen Interruptauslösung genutzt werden. Darüber hinaus stehen die Schnittstellen SPI und I^2C zur Verfügung. Mithilfe eines Soft-UARTS (Softwareseitig werden die Pegel ausgewählter I/O-Pins so gesetzt, dass diese als UART genutzt werden können) und des eingebauten UARTS (auch mit dem TTL-USB-IC verbunden) stehen mehrere UART-Schnittstellen zur seriellen Kommunikation bereit. Auf

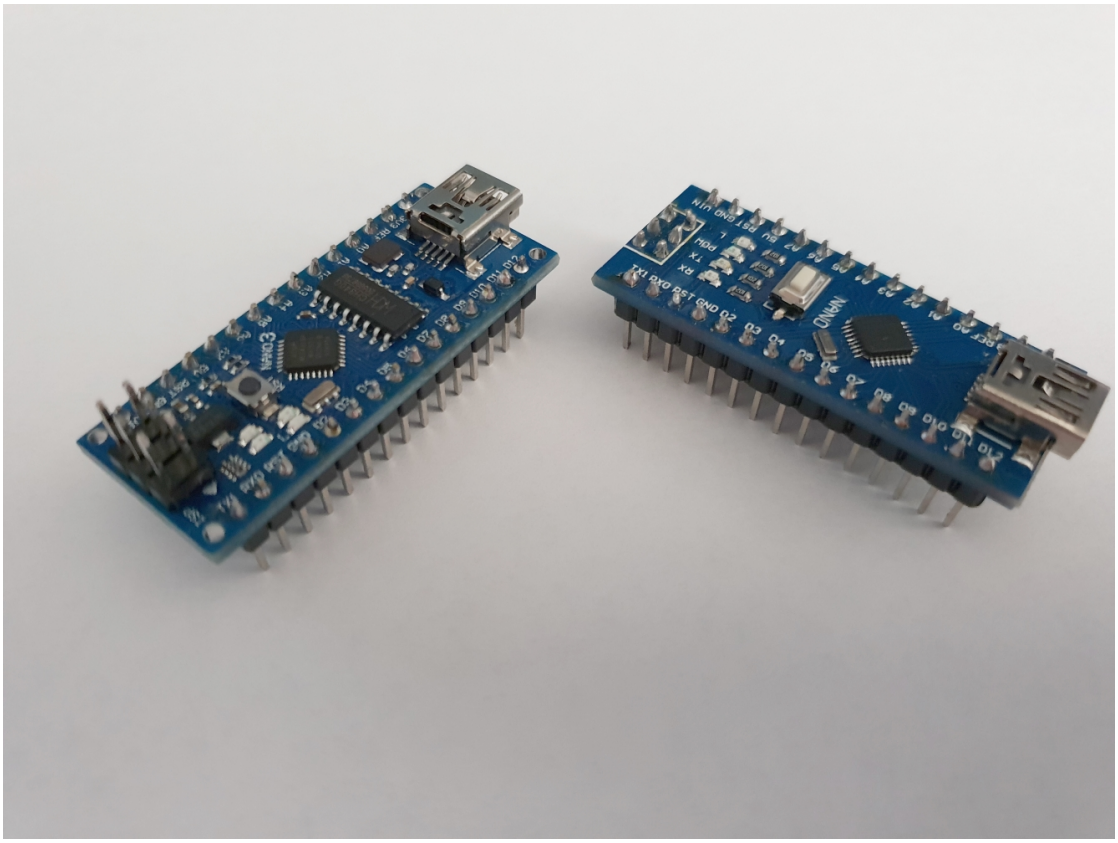


Abbildung 47: Arduino Nano, Version 3.0
Quelle: Eigenes Bild.

dem Board (abgebildet auf Abbildung 47) ist neben dem Mikrocontroller auch ein USB-Seriell-Wandler FT232RL vorhanden. Der USB-Port dient der Kommunikation zu einem PC (für ein

serielles Terminal oder zur Programmierung) und dient gleichzeitig der Stromversorgung mit 5V. Mithilfe seines Spannungsreglers ist es jedoch auch möglich, ihn mit einer Spannung von 6 - 20V zu betreiben. Seine Taktfrequenz beträgt 16 MHz, er besitzt 32kiB Flash-Speicher, 2 kiB SRAM und 1 kiB nichtflüchtigen EEPROM. Das gesamte Board hat eine Größe von 1,85 · 4,32 cm (vgl. [AG08], [New08]) und kostet im Onlinehandel ca. 3€.

7.3.2 PCF8574

Der PCF8574 ist ein Schnittstellen-IC, der per I^2C gesteuert werden kann und acht Ein- und Ausgänge besitzt, deren Pegel standardmäßig per Pull-up auf HIGH gehalten werden. Er verfügt über drei einstellbare Adresseingänge, was acht PCF8574 (des gleichen Subtyps) an einem I^2C-Bus ermöglicht. Es gibt zwei Subtypen, die sich jedoch nur in ihrer Basis-I^2C-Adresse unterscheiden. Der PCF8574 hat die Basisadresse 32 (0x20), der PCF8574 nutzt die Basisadresse 56 (0x38). Die untersten Bits der I^2C-Adresse sind schaltbar, indem die Versorgungsspannung oder Masse[21] angelegt werden. Zusätzlich zu den Subtypen wird der Gehäusetyp unterschieden und ein Suffix angehängt: P (oder N) entspricht DIP16, T entspricht SO16 und S entspricht SSOP20 (vgl. [NXP13]).

Abbildung 48 zeigt den PCF8574 in verschiedenen Varianten. Die ersten drei tragen die Typenbezeichnung PCF8574AT, der letzte PCF8574AN. Die dritte Variante ist so ausgeführt, dass sie direkt LCDs ansteuern kann (etwa das in dieser Arbeit verwendete 16 × 2-Display) und hat daher eine zum LCD passende Pinbelegung.

7.4 CAN-Datenübertragungsprotokoll

Um Sensordaten von der CAN-I^2C-Bridge an die Remote Unit zu übertragen, wurde ein Protokoll zur Übertragung auf dem CAN-Bus entworfen. Dieses sieht vor, dass Teile der CAN-ID genutzt werden, um den gefundenen Sensortyp zu identifizieren. Weiterhin findet sich hier auch die ID der CAN-I^2C-Bridge. Tabelle 8 zeigt den Aufbau eines Paketes mit Sensordaten. Ein Paket von der Remote Unit, um Daten anzufordern, ist ein CAN-Paket mit der ID 0xFF ohne weitere Nutzdaten. Der Paketaufbau zur Steuerung eines Aktors mithilfe der CAN-I^2C-Bridge ergibt sich aus Listing 6. Hierbei berechnet sich die CAN-ID aus dem Sensortyp bzw. dem daraus resultierenden Type-Offset sowie der ID der angesprochenen CAN-I^2C-Bridge.

[21] Eigene Versuche haben gezeigt, dass das Anlegen von Masse nicht notwendig ist, damit die entsprechenden Adressbits LOW sind.

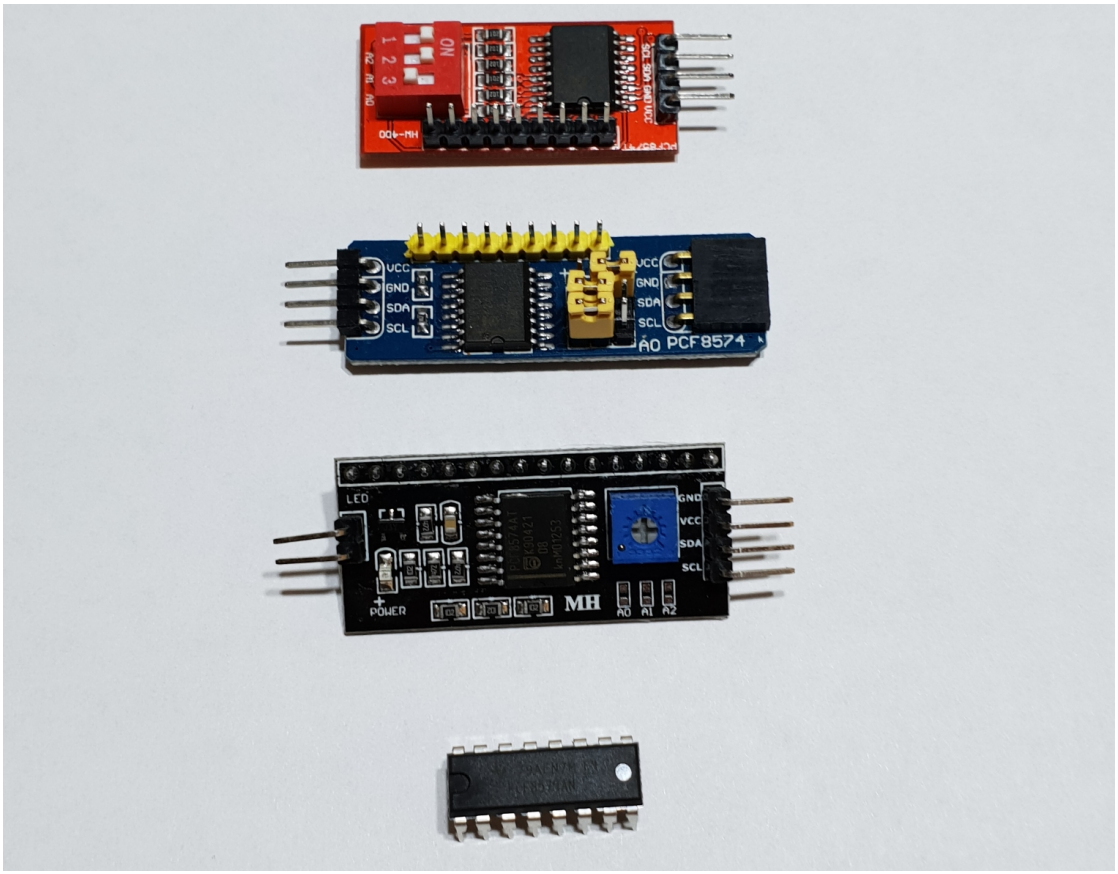


Abbildung 48: PCF8574 in verschiedenen Varianten
Quelle: Eigenes Bild

CAN-ID			DATA							
Bit 28-16	Bit 15-8	Bit 7-0	Data [7]	Data [6]	Data [5]	Data [4]	Data [3]	Data [2]	Data [1]	Data [0]
0...0	Type ID	Bridge ID	Sensor Data	Sensor Data optional	Sensor Data optional	Sensor Data optional	Sensor Data optional	Sensor Data optional	Sensor Data optional	Sensor Data optional

Tabelle 8: Aufbau eines CAN-Paketes der CAN-I^{2}C-Bridge mit Sensorwerten

7.5 Implementierung

Die CAN-I^{2}C-Bridge wurde umgesetzt, indem ein Mikrocontroller (Arduino Nano) mit einem CAN-Modul (zum Anschluss an die Remote Unit) verbunden wurde und ein PCF8574 zur Feststellung der eingestellten ID sowie der verwendete Sensor (am SUB-D-Stecker) per I^{2}C angebunden wurden. Die fertige CAN-I^{2}C-Bridge ist in Abbildung 50 zu erkennen.

Abbildung 49 zeigt den Schaltplan der CAN-I^{2}C-Bridge.

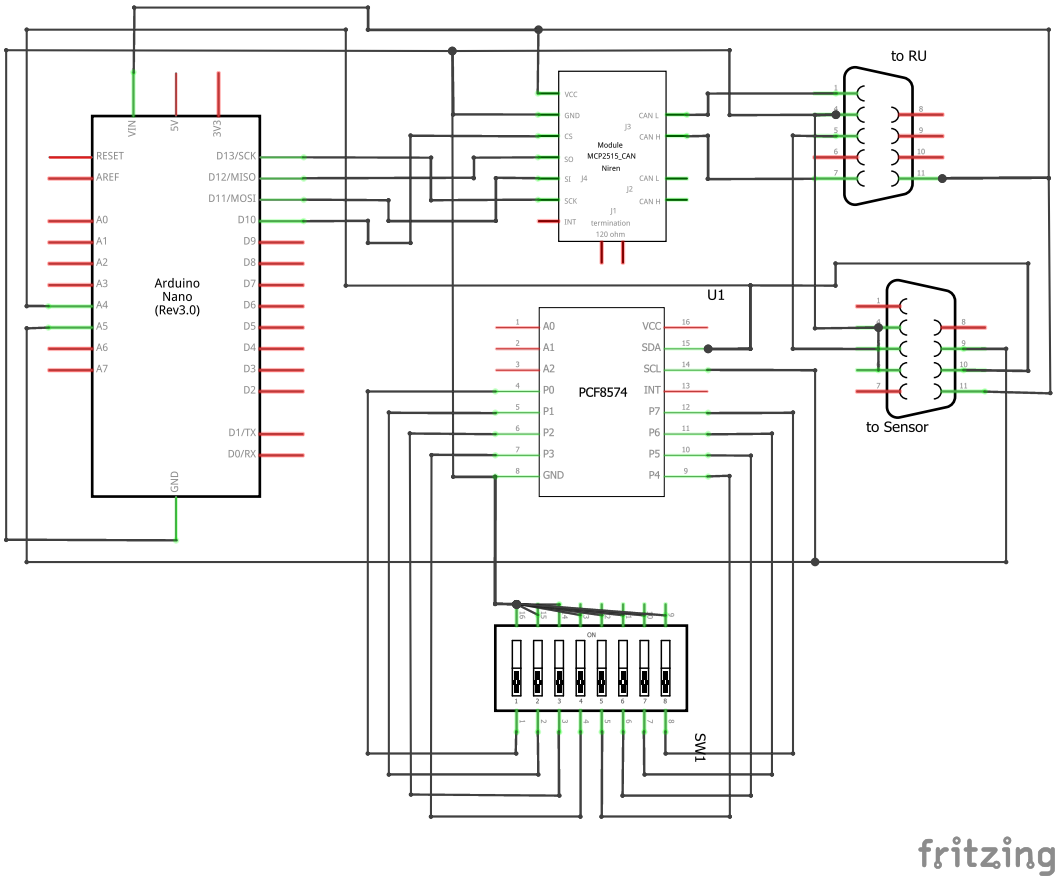


Abbildung 49: Schaltplan der CAN-I^{2}C-Bridge

7.5.1 Hardware

Zur Kommunikation mit der Remote Unit enthält die CAN-I^{2}C-Bridge ein CAN-Modul. Per I^{2}C ist neben dem Sensor- / Aktormodul ein PCF8574 angeschlossen. Dieser ermöglicht es, der CAN-I^{2}C-Bridge eine ID zu geben, die mithilfe von DIP-Schaltern vom Benutzer eingestellt werden kann. Diese ID unterscheidet mehrere CAN-Bridges mit demselben Sensortyp voneinander und wird auch innerhalb der GUI (siehe Abbildung 51) angezeigt.
Abbildung 50 zeigt die CAN-I^{2}C-Bridge, die aus einem Microcontroller (unten mittig, Arduino Nano bzw. ein entsprechender Nachbau), einem CAN-Interface-Board (links) mit MCP2515 und TJA1050 sowie einem PCF8574 mit angeschlossenen DIP-Schaltern sowie einem Stecker für den Anschluss an die RU und einer Buchse für den Anschluss eines I^{2}C-Sensors besteht.
Mithilfe des erkannten Sensortyps sowie einer achtbittigen, vom Benutzer per DIP-Schaltern festgelegten Sensor-ID, ergibt sich eine CAN-ID der Nachrichten mit gemessenen Sensorda-

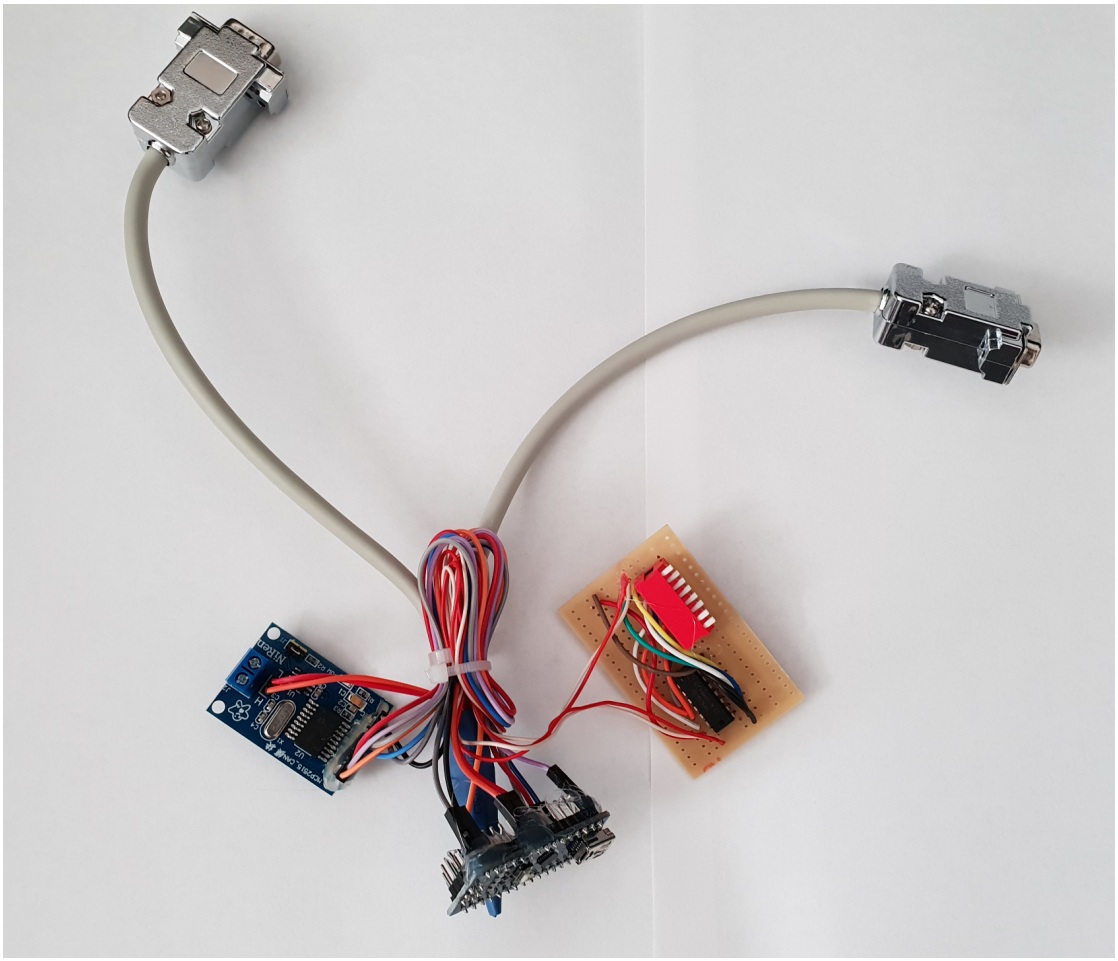


Abbildung 50: CAN-I^{2}C-Bridge
Quelle: Eigenes Bild.

ten. Es ist für den Benutzer möglich, 256 verschiedene Sensoren des jeweils selben Typs zu verwenden. Da die Sensoren durch die CAN-I^{2}C-Bridge mit einer ID versehen sind, ist es dem Benutzer möglich, die angezeigten Werte in der GUI dem einzelnen Sensor zuzuordnen. Abbildung 51 zeigt die Anzeige eines per CAN-I^{2}C-Bridge angeschlossenen Temperatursensors mit der CAN-I^{2}C-Bridge-ID 160.

7.5.2 Software

Die auf dem Mikrocontroller laufende Software wurde mithilfe der Arduino IDE umgesetzt. Hierbei gibt es zwei Design-Vorgaben:

- Die Methode `void setup()` wird beim Start des Mikrocontrollers genau einmal aus-

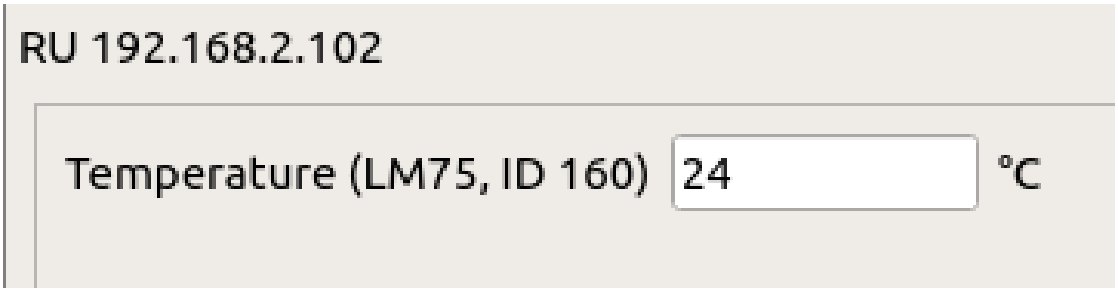


Abbildung 51: Anzeige eines per CAN-I^{2}C-Bridge verbundenen Temperatursensors mit Sensor-ID

geführt. Hier finden sich Initialisierungen, etwa für die serielle Schnittstelle zur Kommunikation mit einem Terminal per UART.

- Die Methode `void loop()` wird innerhalb einer Endlosschleife immer wieder aufgerufen.

Da die Service-Discovery auch das Entfernen und Anschließen eines Sensors (oder Aktors) von der CAN-I^{2}C-Bridge berücksichtigen soll, finden sich innerhalb von `setup()` lediglich Initialisierungen zur Bereitstellung von Kommunikationsschnittstellen wie UART, SPI und I^{2}C sowie für den MCP2515. Für Sensoren ist dies nicht möglich, da ein Wechseln des Sensors im laufenden Betrieb möglich sein soll. Daher findet die entsprechende Initialisierung erst innerhalb von `loop()` statt, nachdem anhand der I^{2}C-Adresse der entsprechende Sensortyp festgestellt wurde.

Listing 6 zeigt den Aufbau der Methode `loop()`. Hier wird in jedem Schleifendurchgang zunächst die ID der CAN-I^{2}C-Bridge ermittelt. Anschließend wird geprüft, ob eine Anfrage zur Übermittlung von Sensordaten (und Aktortypen) vorliegt. Ist dies der Fall, wird anhand der I^{2}C-Adresse des angeschlossenen Gerätes dessen Typ bestimmt, ggf. eine Abfrage durchgeführt und der Sensortyp mit Daten oder der Aktortyp an die Remote Unit übertragen. Alternativ wird geprüft, ob ein Steuerungskommando für einen Aktor vorliegt, welches in diesem Fall umgesetzt wird.

Aktoren an der CAN-I^{2}C-Bridge

Im Folgenden wird beschrieben, wie die Aktoren an der CAN-I^{2}C-Bridge betrieben werden.

- **LCD**
  Das genutzte LCD ist das in Abschnitt 6.1.3 beschriebene, welches mit einem PCF8574

(siehe Unterunterabschnitt 7.3.2, I^{2}C-Adresse ist 0x27) angesteuert und per I^{2}C angeschlossen wird. Das verwendete Display hat eine Größe von 16 $\times$ 2 Zeichen. Es sind jedoch auch andere Varianten verfügbar, beispielsweise solche mit 4 Zeilen oder mit einer Zeilenlänge von 20 Zeichen. Für die Benutzung andersartiger Displays bedarf es nur geringer Anpassung.
Während der Umsetzungsphase ergab sich das Problem, dass das Display zwar 32 Zeichen anzeigen kann, per CAN jedoch maximal 8 Bytes übertragen werden. Daher wird bei Nutzung der CAN-I^{2}C-Bridge der Displayinhalt auf vier Pakete (CAN-IDs 0x200000 + id, ... , 0x230000 + id; id mithilfe des PCF8574 an Adresse 0x38 ermittelt, siehe oben) aufgeteilt, in der CAN-I^{2}C-Bridge zwischengespeichert und mit einem weiteren Befehl (CAN-ID 0x2F0000 + id) zum Display übertragen. Bei einer Anpassung für die genannten anderen Displaytypen müssten entsprechend mehr CAN-Pakete gesendet werden, bevor der Text an das LCD übertragen wird.

- **Servomotor**
  Um den Servomotor zu steuern, wird eine CAN-Message mit der ID 0x300000 + id gesendet, die den Sollwert des PWM-Controllers im Datenfeld beinhaltet.

7.6 Nutzung der CAN-I^{2}C-Bridge

Der Anschluss von Sensoren an der CAN-I^{2}C-Bridge erfolgt mithilfe einer SUB-D-Steckverbindung genau wie beim Anschluss einer Remote Unit. Einzig zu beachten ist, dass mehrere CAN-I^{2}C-Bridges bei Verwendung gleichartiger Sensoren unterschiedliche IDs haben müssen, welche mithilfe eines DIP-Schalters eingestellt werden. Abbildung 46 zeigt beispielhaft die Vergabe der IDs 1,...,5.

Algorithm 6: LOOP()

```
1  read ID from Adress 0x38 (PCF8574)
2  if CAN-Message was read then
3      if CAN-Message is request then
4          for each I2C-Adress a (except 0x38) do
5              bool succ = try to read Byte
6              if succ then
7                  switch a do
8                      case 0x48 do
9                          //Temperature Sensor
10                         //No Initialization Needed
11                         typeOffset = 0x0100
12                         read Temperature
13                         Send CAN-Message with Temperature and CAN-ID =
                            typeOffset + ID
14                     case 0x53 do
15                         //Accelerometer
16                         Initialize Sensor
17                         typeOffset = 0x0300
18                         Read Accelerations ax, ay, az
19                         Send CAN-Message with three values and CAN-ID = typeOffset
                            + ID
20                     case 0x27 do
21                         //LCD
22                         Initialize Sensor
23                         typeOffset = 0x0500
24                         Send CAN-Message with no Data and CAN-ID = typeOffset +
                            ID
25                     ...
26     else if CAN-Message is first part of LCD-Data then
27         store 8 Bytes to (global) &buffer
28     else if CAN-Message is second part of LCD-Data then
29         store 8 Bytes to (global) &buffer+8
30     ...
31     else if CAN-Message is „print-to-lcd“-Message then
32         Print Buffer to LCD
33     else if CAN-Message is „Set-Servo“-Message then
34         Set PWM-Value to value within CAN-Message
```

8 Evaluation

Um das Protokoll zu evaluieren, werden veschiedene Szenarien in mehreren Messreihen umgesetzt. Hierzu gehören der Test von Sensoren und Aktoren, sowohl per I^2C als auch per CAN-I^2C-Bridge angebunden, sowie die Service Discovery auf Netzebene.
Zur Darstellung der Ergebnisse wurden die Log-Dateien mit den Telemetriedaten mithilfe von Excel aufbereitet. Screenshots und Fotos des Aufbaus zeigen, inwieweit das erwartete Ergebnis erreicht wurde.

8.1 Service Discovery auf Remote-Unit-Ebene

Die Service-Discovery auf Remote Unit-Ebene wird getestet, indem in mehreren Testläufen verschiedene Kombinationen aus Sensoren und Aktoren, jeweils mit und ohne CAN-I^2C-Bridge angeschlossen und die empfangenen Telemetriepakete mit dem Erwartungswert verglichen werden.

8.1.1 Messreihe 1: I^2C-Sensoren

Es wird zunächst der Sensor-Thread gestartet, woraufhin nacheinander

1. LM70
2. TSL2581
3. BMP280

angeschlossen werden. Anschließend wird der SensorThread gestoppt.

Erwartetes Ergebnis
Es wird erwartet, dass Telemetriepakete in folgender Reihenfolge aufgezeichnet werden:

- Zu Beginn werden zwei Telemetriepakete ([1,1] und [1,7][22]) empfangen, die den Empfang und die erfolgreiche Ausführung des Kommandos „StartSensorThread“ quittieren.
- Anschließend werden Pakete ohne Application Data empfangen.
- Nach einigen Sekunden werden Pakete empfangen, die als Application Data den Eintrag 0x01 (ID des Temperatursensors) und und einen im Bereich zwischen 0x17 und 0x1A (Temperatur während der Messung) enthalten.

[22] [1,7] meint Service Type 1 und Service Subtype 7.

- Weitere Sekunden später werden Pakete empfangen, die als Application Data zuerst eine 0x03 (der TSL2581 hat eine niedrigere I^2C-Adresse und steht daher zuerst im Paket) und einen niedrigen uint32_t-Wert sowie anschließend die Daten aus dem letzten Punkt enthalten.
- Kurze Zeit später werden Pakete wie soeben empfangen, die jedoch zusätzlich die ID 0x0F, einen float32_t-Wert, die ID 0x10 und einen weiteren float32_t-Wert enthalten.
- Abschließend wird die Quittierung des Befehls „StopSensorThread“ empfangen.

Ebenso wird erwartet, dass eine entsprechende Anzeige im RemoteUnitWidget erscheint.

Messprotokoll

Im Folgenden (Abbildung 52) wird das Messprotokoll als Screenshot der mithilfe von Excel aufbereiteten CSV-Datei gezeigt.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V
1	Receive Time	IP	APID	ServiceType	ServiceSubType	SeqCount	AppData[0]	...														
2	07.11.2018 17:20:30	192.168.178.10	1	1	1	0	18	1	c0	1												
3	07.11.2018 17:20:30	192.168.178.10	1	1	7	1	18	1	c0	1												
4	07.11.2018 17:20:30	192.168.178.10	1	80	97	0																
5	07.11.2018 17:20:32	192.168.178.10	1	80	97	1																
6	07.11.2018 17:20:33	192.168.178.10	1	80	97	2																
7	07.11.2018 17:20:35	192.168.178.10	1	80	97	3																
8	07.11.2018 17:20:36	192.168.178.10	1	80	97	4	1	0														
9	07.11.2018 17:20:37	192.168.178.10	1	80	97	5	1	0														
10	07.11.2018 17:20:38	192.168.178.10	1	80	97	6	1	16														
17	07.11.2018 17:20:47	192.168.178.10	1	80	97	000d	1	16														
18	07.11.2018 17:20:49	192.168.178.10	1	80	97	000e	3	0	0	0	4	1	16									
21	07.11.2018 17:20:53	192.168.178.10	1	80	97	11	3	0	0	0	0a	1	16									
22	07.11.2018 17:20:56	192.168.178.10	1	80	97	12	3	0	0	0	8	1	16	0f	44	7c	58	64	41	a8	d2	eb
27	07.11.2018 17:21:09	192.168.178.10	1	80	97	17	3	0	0	0	0b	1	16	0f	44	7c	54	89	41	a9	1b	fb
28	07.11.2018 17:21:10	192.168.178.10	1	1	1	2	18	1	c0	1												
29	07.11.2018 17:21:10	192.168.178.10	1	1	7	3	18	1	c0	1												
30	07.11.2018 17:21:11	192.168.178.10	1	80	97	18	3	0	0	0	0b	1	16	0f	44	7c	58	2c	41	a9	6f	7b

Abbildung 52: Telemetriepakete bei Messreihe 1

Die Anzeige im Widget wird in Abbildung 53 gezeigt.

Auswertung

Die ersten Zeilen zeigen das erwartete Ergebnis. Bei den ersten beiden Temperaturmessungen wird der fehlerhafte Wert 0 übertragen. Hier wird davon ausgegangen, dass der Temperatursensor eine gewisse Zeit zur Initialisierung benötigt. Die weiteren Temperaturwerte liegen im erwarteten Bereich. Nach einigen Sekunden werden vier Bytes Daten Telemetrie vom Helligkeitssensor - in der Application Data vor dem Temperatursensor - übertragen. Anschließend sind alle drei Sensoren zu erkennen. Anzumerken ist, dass der BMP280 sowohl Luftdruck als

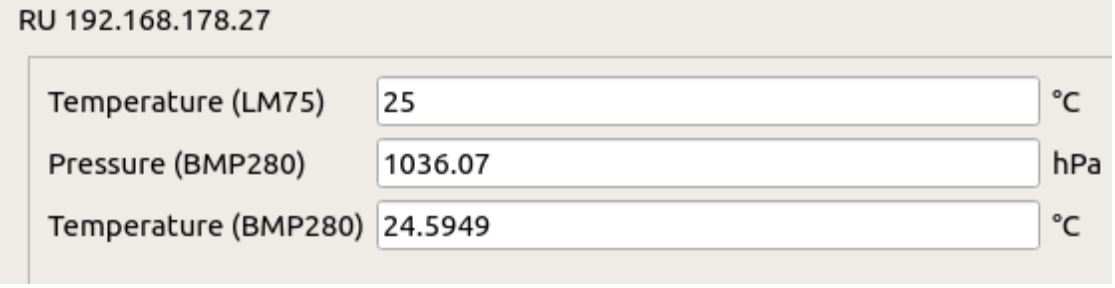


Abbildung 53: Anzeigeelemente der GUI bei Messreihe 1

auch Temperatur misst und daher acht Bytes Daten überträgt. Nachdem der Befehl zum Beenden des Threads empfangen und ausgeführt wurde, durchläuft der Thread ein letzes Mal die in ihm vorhandene Schleife (die Prüfung, ob der Thread noch läuft ist zu Beginn der Schleife). Daraufhin werden keine weiteren Pakete empfangen.
Der Test wurde erfolgreich absolviert.

8.1.2 Messreihe 2: I²C-Aktoren

Es wird zunächst der Sensor-Thread gestartet, woraufhin nacheinander

1. das LC-Display
2. der Servomotor am PCA9685

angeschlossen werden. Es wird ein Text („ABCXYZ123890....TEST-ANZEIGE0001“) an das Display gesendet und anschließend der Servomotor nacheinander zu beiden Endpositionen (Minumum- und Maximumauslenkung) sowie in eine mittlere Position bewegt. Anschließend wird der SensorThread gestoppt.

Erwartetes Ergebnis

In den Telemetriepaketen finden sich zunächst Acknowledgements für das Starten des Sensorthreads und RemoteUnitService-Pakete ohne Application Data. Nach Anschluss des Displays findet sich in der Application Data die ID 0x07 (für das LCD), gefolgt von der APID, dem Service Type und den Service Subtype, unter dem das LCD angesprochen werden kann[23]: 0x0001 (APID), 0x80 (Service Type des RemoteUnitService) und 0x85 (Service Subtype für das Ansteuern des LCD). Analog finden sich nach dem Anschluss des PWM-Moduls Werte mit ID 0x08, APID 0x0001, Service Type 0x80 und Service Subtype 0x86.
In der GUI werden Anzeigeelemente für das LC-Display (Textfeld) und den PWM-Controller (Slider) angezeigt.
Nach dem Absenden des Textes erfolgen Acknowledgements ([1,1] und [1,7]). Das Auslenken des Motors führt ebenfalls zu Acknowledgements. Es werden weiterhin Telemetriepakete des Remote Unit Service mit Angaben der angeschlossenen Sensoren übertragen. Zuletzt finden sich Bestätigungen für das Stoppen des SensorThreads.
Der eingegebene Text wird korrekt angezeigt: In der ersten Zeile steht „ABCXYZ123890....“ und in der zweiten „TEST-ANZEIGE0001“. Der Motor bewegt sich (entgegen dem Uhrzeigersinn) in die eine Endposition und anschließend (im Uhrzeigersinn) in die entgegengesetzte. Zuletzt nimmt er eine Position - in etwa mittig - zwischen den Endpositionen ein. Darüber hinaus zeigen die Debugausgaben der Remote Unit die Zeilen „value: 0“ (Minimumwert), „value: 255“ sowie eine mit einem Wert um 128.

Messprotokoll

Im Folgenden (Abbildung 54) wird das Messprotokoll der eingehenden Telemetriepakete als Screenshot der mithilfe von Excel aufbereiteten CSV-Datei gezeigt. Abbildung 55 zeigt die

[23] Diese werden von der Remote Unit festgelegt.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
1	Receive Time	IP	APID	ServiceType	ServiceSubType	SeqCount	AppData[0]	...								
2	08.11.2018 17:12:46	192.168.178.10	1	1	1	0	18	1	c0	1						
3	08.11.2018 17:12:46	192.168.178.10	1	1	7	1	18	1	c0	1						
4	08.11.2018 17:12:46	192.168.178.10	1	80	97	0										
7	08.11.2018 17:12:50	192.168.178.10	1	80	97	3										
8	08.11.2018 17:12:51	192.168.178.10	1	80	97	4	7	0	1	80	85					
9	08.11.2018 17:12:52	192.168.178.10	1	80	97	5	7	0	1	80	85					
10	08.11.2018 17:12:53	192.168.178.10	1	80	97	6	7	0	1	80	85	8	0	1	80	86
14	08.11.2018 17:12:59	192.168.178.10	1	80	97	000a	7	0	1	80	85	8	0	1	80	86
15	08.11.2018 17:12:59	192.168.178.10	1	1	1	0	18	1	c0	0						
16	08.11.2018 17:12:59	192.168.178.10	1	1	7	1	18	1	c0	0						
17	08.11.2018 17:13:00	192.168.178.10	1	80	97	000b	7	0	1	80	85	8	0	1	80	86
20	08.11.2018 17:13:03	192.168.178.10	1	1	1	2	18	1	c0	0						
21	08.11.2018 17:13:03	192.168.178.10	1	1	7	3	18	1	c0	0						
22	08.11.2018 17:13:04	192.168.178.10	1	80	97	000e	7	0	1	80	85	8	0	1	80	86
23	08.11.2018 17:13:04	192.168.178.10	1	1	1	4	18	1	c0	0						
24	08.11.2018 17:13:04	192.168.178.10	1	1	7	5	18	1	c0	0						
25	08.11.2018 17:13:05	192.168.178.10	1	80	97	000f	7	0	1	80	85	8	0	1	80	86
27	08.11.2018 17:13:07	192.168.178.10	1	1	1	6	18	1	c0	0						
28	08.11.2018 17:13:07	192.168.178.10	1	1	7	7	18	1	c0	0						
32	08.11.2018 17:13:10	192.168.178.10	1	1	1	8	18	1	c0	0						
33	08.11.2018 17:13:10	192.168.178.10	1	1	7	9	18	1	c0	0						
34	08.11.2018 17:13:11	192.168.178.10	1	80	97	14	7	0	1	80	85	8	0	1	80	86
37	08.11.2018 17:13:14	192.168.178.10	1	1	1	2	18	1	c0	1						
38	08.11.2018 17:13:15	192.168.178.10	1	1	7	3	18	1	c0	1						

Abbildung 54: Telemetriepakete bei Messreihe 2

Anzeigeelemente zur Steuerung des LC-Displays und des Servomotors, nachdem beide an die Remote Unit angeschlossen wurden.

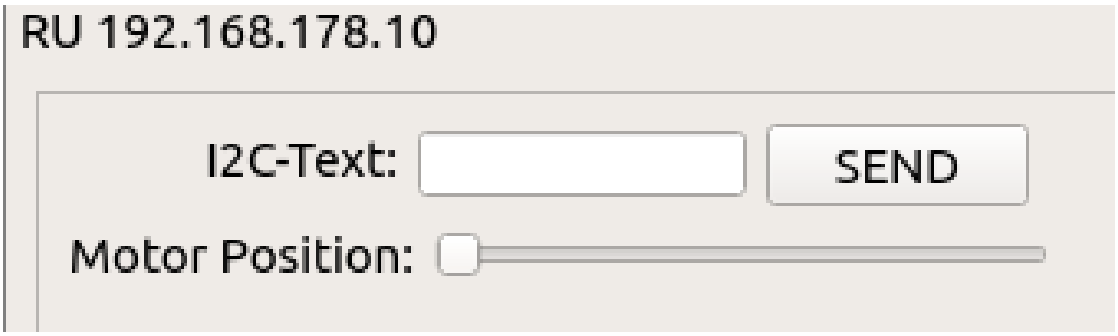


Abbildung 55: Anzeige der Bedienelemente für LCD und PWM-Controller

Abbildung 56 zeigt das LCD, nachdem der Text versendet wurde. Der PWM-Slider stand bereits auf seiner Minimalposition, sodass ein Ziehen und Schieben auf die Position ganz links nicht das Qt-Signal einer Änderung auslöste[24]. Erst ein leichtes Schieben nach rechts

[24] Mithilfe des Universal TC Packet Widgets wurde im Anschluss an den Test manuell der Befehl zur Aus-

Abbildung 56: LCD-Anzeige nach Absenden des Textes

sowie ein erneutes Schieben in die Minimalposition ergaben zwei Bewegungen, die letzte zur minimalen Auslenkung des Servomotors. Die nächsten Telekommandos sorgten dafür, dass sich der Motor erst in seine Maximalposition und dann in eine etwa mittige Position bewegte. Die Debugausgaben der Remote Unit lauteten „value: 4“, „value: 0“, „value: 255“ und „value: 120“.

Auswertung

Die empfangene Telemetrie entspricht den Erwartungen. Ebenfalls den Erwartungen entsprachen die Anzeigen in der GUI sowie die Anzeige des Textes auf dem LCD. Die nicht sofort ansteuerbare minimale Auslenkung des Servomotors ergibt sich aus einer Einschränkung des Qt-Frameworks: Entweder werden bei einer Bewegung viele Signals ausgelöst, auch wenn der Button der Maus nicht losgelassen wurde oder es wird ausschließlich bei einer Veränderung gegenüber dem ursprünglichen Wert ein Signal ausgelöst. Die erste Variante wurde auch untersucht, führte jedoch zu einer Flut an Telekommandos, da innerhalb einer Sekunde mehrere Male eine Veränderung des Wertes festgestellt wurde. Bei der zweiten Variante wird hingegen kein Telekommando gesendet, wenn der neue Sliderwert dem alten entspricht, obwohl der Slider zwischenzeitlich bewegt wurde. An dieser Stelle wird daher in Kauf genommen, dass der Motor minimal bewegt werden muss, bevor er seine Minimalposition erreichen kann, da die Bedienung mit dem Slider generell sehr ungenau ist und eher ein Proof-of-Concept darstellt[25]. **Der Test wurde erfolgreich absolviert.**

lenkung auf Position 0 gesendet, indem ein Telekommando mit Application Data 0x00 an die APID 1, [128,134] gesendet wurde. Der Motor fuhr in seine Minimalpostion.

[25] Diese Einschränkung wurde zu Testzwecken umgangen, indem ein einzelnes Telekommando mit dem Wert 0 gesendet wurde.

8.1.3 Messreihe 3: Sensoren an CAN-I^{2}C-Bridge

Innerhalb dieser Messreihe werden zwei gleichartige Sensoren (LM70) mithilfe von zwei CAN-I^{2}C-Bridges an einer Remote Unit betrieben. Beide CAN-I^{2}C-Bridges sind vorab mit dem Sensor verbunden und haben die voreingestellten Bridge-IDs 42 und 84. Zunächst wird der SensorThread gestartet, woraufhin die beiden CAN-I^{2}C-Bridges mit Sensoren nacheinander angeschlossen werden. Zuletzt wird der SensorThread gestoppt.

Erwartetes Ergebnis

Es wird erwartet, dass beide Temperatursensoren in der GUI angezeigt werden, d.h. mit ihren IDs und sinnvollen Temperaturwerten. Innerhalb der Telemetriepakete sollte sich dieses widerspiegeln, indem - abgesehen von leeren Paketen und Acknowledgements wie bei den ersten beiden Messreihen - die ID 0x0A (für einen per CAN-I^{2}C-Bridge betriebenen LM70), die Bridge-ID 0x2A und ein Temperaturwert im Bereich zwischen 0x15 und 0x1A empfangen werden sowie dasselbe im Anschluss erneut, jedoch mit Bridge-ID 0x54.
Es wird weiterhin erwartet, dass die Acknowledgements für das Starten und Stoppen des SensorThreads korrekt empfangen werden.

Messprotokoll

Im Folgenden (Abbildung 57) werden das Messprotokoll als Screenshot der mithilfe von Excel aufbereiteten CSV-Datei sowie die Anzeigeelemente innerhalb der GUI (siehe Abbildung 58) gezeigt.

	A	B	C	D	E	F	G	H	I	J	K	L
1	Receive Time	IP	APID	ServiceType	ServiceSubType	SeqCount	AppData[0]	...				
2	09.11.2018 15:08:13	192.168.178.10	1	1	1	0	18	1	c0	1		
3	09.11.2018 15:08:13	192.168.178.10	1	1	7	1	18	1	c0	1		
4	09.11.2018 15:08:14	192.168.178.10	1	80	97	0						
7	09.11.2018 15:08:17	192.168.178.10	1	80	97	3						
8	09.11.2018 15:08:18	192.168.178.10	1	80	97	4	0a	2a	15	0a	2a	15
9	09.11.2018 15:08:19	192.168.178.10	1	80	97	5	0a	2a	15			
15	09.11.2018 15:08:27	192.168.178.10	1	80	97	000b	0a	2a	15			
16	09.11.2018 15:08:28	192.168.178.10	1	80	97	000c	0a	2a	15	0a	54	16
22	09.11.2018 15:08:35	192.168.178.10	1	80	97	12	0a	2a	15	0a	54	16
23	09.11.2018 15:08:36	192.168.178.10	1	1	1	2	18	1	c0	1		
24	09.11.2018 15:08:36	192.168.178.10	1	1	7	3	18	1	c0	1		
25	09.11.2018 15:08:36	192.168.178.10	1	80	97	13	0a	2a	15	0a	54	16

Abbildung 57: Telemetriepakete bei Messreihe 3

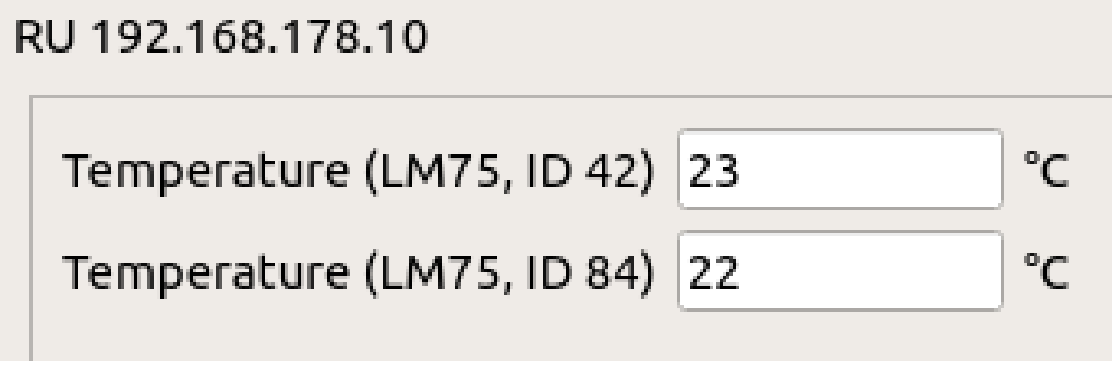


Abbildung 58: Anzeigeelemente bei Durchführung von Messreihe 3

Auswertung

Die GUI zeigt das erwartete Ergebnis: Zwei LM70-Sensoranzeigen, jeweils mit den IDs 42 und 84 und plausiblen Temperaturwerten. Das Telemetrieprotokoll zeigt ebenfalls die erwarteten Daten: Acknowledgements für das Starten des SensorThreads, Pakete ohne ApplicationData, während noch keine Sensoren (bzw. CAN-I^{2}C-Bridges) angeschlossen sind, Pakete mit der Angabe, einen Temperatursensor mit Bridge-ID 42 gefunden zu haben[26], Pakete mit Daten zweier Sensoren an CAN-I^{2}C-Bridges und die Acknowledgements für das Stoppen des SensorThreads. Beim letzten Paket wurde - wie in Messreihe 1 - die Messung gestartet, bevor das Kommando zum Stopp des SensorThreads eingegangen war und vollständig ausgeführt.

Der Test wurde erfolgreich absolviert.

[26] Das erste eingehende CAN-Paket wird zweimal angezeigt. Es wird davon ausgegangen, dass dies ein Timing-Problem ist, welches möglicherweise vom Arduino ausgeht.

8.1.4 Messreihe 4: Aktoren an CAN-I^{2}C-Bridge

Innerhalb dieser Messreihe werden nacheinander zwei gleichartige Aktoren (LCDs) über CAN-I^{2}C-Bridges (mit eingestellten IDs 42 und 84) an eine Remote Unit angeschlossen, nachdem der SensorThread gestartet wurde. Es werden jeweils unterschiedliche Nachrichten („42...00000000000111111111111111“ und „84...00000000001111111111111111“) an die LCDs gesendet. Anschließend wird der SensorThread beendet.

Erwartetes Ergebnis

Neben den Acknowledgements für das Starten und Stoppen des Sensor-Threads sowie für das Schreiben auf die LCDs sollen Telemetriepakete des Remote-Unit-Service empfangen werden, die das Vorhandensein von erst keinem, dann einem und anschließend zwei LCDs anzeigen. Die LCDs sollen den Text anzeigen, der ihnen geschickt wird.

Messprotokoll

In der Tabelle (Abbildung 59) ist - neben den Acknowledgements zum Starten und Stoppen des SensorThreads (Zeilen 1-2 sowie 35-36) - zu erkennen, dass Service-Discovery-Pakete zunächst Informationen zu keinem, dann zu einem (zuerst dreimal hintereinanter zu demselben; daraufhin nur einmal) und dann zu zwei LCDs enthalten. Im Paket in Zeile 10 sind die Bytes für den Aktortyp 13 (0x0D), zwei Bytes für die APID 0x0001 , ein Byte für den Service Type 0x81 (129 = CAN-LCD-Service) und eines für den Service Subtype (0x2A, entspricht der ID der CAN-I^{2}C-Bridge) zur Ansteuerung des Displays vorhanden. Kurz darauf finden sich Pakete mit Informationen zu zwei per CAN-I^{2}C-Bridge angeschlossenen Displays (IDs 0x2A=42_{10} und 0x54=84_{10}, in unkonstanter Reihenfolge). Zeilen 20-21 sowie 26 und 28 zeigen die Acknowledgements für das Schreiben auf die LCDs.
Abbildung 60 zeigt die Remote Unit mit den beiden angeschlossenen LCDs. An den dazwischengeschalteten CAN-I^{2}C-Bridges sind links die ID 42 und rechts die ID 84 gesetzt. Auf dem LCD, welches an die CAN-I^{2}C-Bridge mit ID 42 angeschlossen ist, finden sich die Zeilen „42...00000000000“ und „1111111111111111“. Auf dem anderen LCD finden sich die Zeilen „84...00000000001“ und „1111111111111111“.
Abbildung 61 zeigt die Anzeigeelemente in der GUI, nachdem beide LCDs angeschlossen wurden (Der Cursor wurde zur besseren Sichtbarkeit nach links versetzt). Erkennbar sind die IDs der CAN-I^{2}C-Bridges 84 und 42.

Auswertung

Die empfangenen Telemetriepakete entsprachen den Erwartungen mit der Ausnahme, dass das Telemetriepaket in Zeile 9 den Sensor dreifach anzeigt. Dieses führt jedoch nicht zu einer

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
1	Receive Time	IP	APID	ServiceType	ServiceSubType	SeqCount	AppData[0]	...													
2	10.11.2018 17:25:59	192.168.178.10	1	1	1	0	18	1	c0	1											
3	10.11.2018 17:25:59	192.168.178.10	1	1	7	1	18	1	c0	1											
4	10.11.2018 17:25:59	192.168.178.10	1	80	97	0															
8	10.11.2018 17:26:05	192.168.178.10	1	80	97	4															
9	10.11.2018 17:26:06	192.168.178.10	1	80	97	5	0d	0	1	81	2a	0d	0	1	81	2a	0d	0	1	81	2a
10	10.11.2018 17:26:08	192.168.178.10	1	80	97	6	0d	0	1	81	2a										
11	10.11.2018 17:26:09	192.168.178.10	1	80	97	7	0d	0	1	81	2a										
12	10.11.2018 17:26:10	192.168.178.10	1	80	97	8	0d	0	1	81	54	0d	0	1	81	2a					
19	10.11.2018 17:26:20	192.168.178.10	1	80	97	000f	0d	0	1	81	2a	0d	0	1	81	54					
20	10.11.2018 17:26:20	192.168.178.10	1	1	1	0	18	1	c0	0											
21	10.11.2018 17:26:21	192.168.178.10	1	1	7	1	18	1	c0	0											
22	10.11.2018 17:26:21	192.168.178.10	1	80	97	10	0d	0	1	81	54	0d	0	1	81	2a					
25	10.11.2018 17:26:25	192.168.178.10	1	80	97	13	0d	0	1	81	54	0d	0	1	81	2a					
26	10.11.2018 17:26:26	192.168.178.10	1	1	1	2	18	1	c0	0											
27	10.11.2018 17:26:26	192.168.178.10	1	80	97	14	0d	0	1	81	2a	0d	0	1	81	54					
28	10.11.2018 17:26:27	192.168.178.10	1	1	7	3	18	1	c0	0											
29	10.11.2018 17:26:28	192.168.178.10	1	80	97	15	0d	0	1	81	2a	0d	0	1	81	54					
34	10.11.2018 17:26:35	192.168.178.10	1	80	97	001a	0d	0	1	81	54	0d	0	1	81	2a					
35	10.11.2018 17:26:36	192.168.178.10	1	1	1	2	18	1	c0	1											
36	10.11.2018 17:26:36	192.168.178.10	1	1	7	3	18	1	c0	1											

Abbildung 59: Telemetriepakete bei Messreihe 4

Beeinträchtigung in der Anzeige des Steuerungselementes für das LCD (es wurde nur einmal angezeigt). Es wird davon ausgegangen, dass an dieser Stelle der Sendepuffer des MCP2515 noch Daten aus den vorangegangenen Abfragen enthielt. Diese wurden aufgrund eines nicht empfangenen CAN-Acknowledgements nicht verworfen (diese Remote Unit hatte zu diesem Zeitpunkt noch keinen zweiten CAN-Controller). Daher wurden alle drei Anfragen aus dem Sendepuffer hintereinander beantwortet, sodass entsprechend dreimal der selbe Aktor erkannt wurde. Die GUI zeigt die erwarteten Anzeigeelemente für die LCDs. Die Anzeige auf den LC-Displays in Abbildung 60 zeigen die gesendeten Texte.

Der Test wurde erfolgreich absolviert.

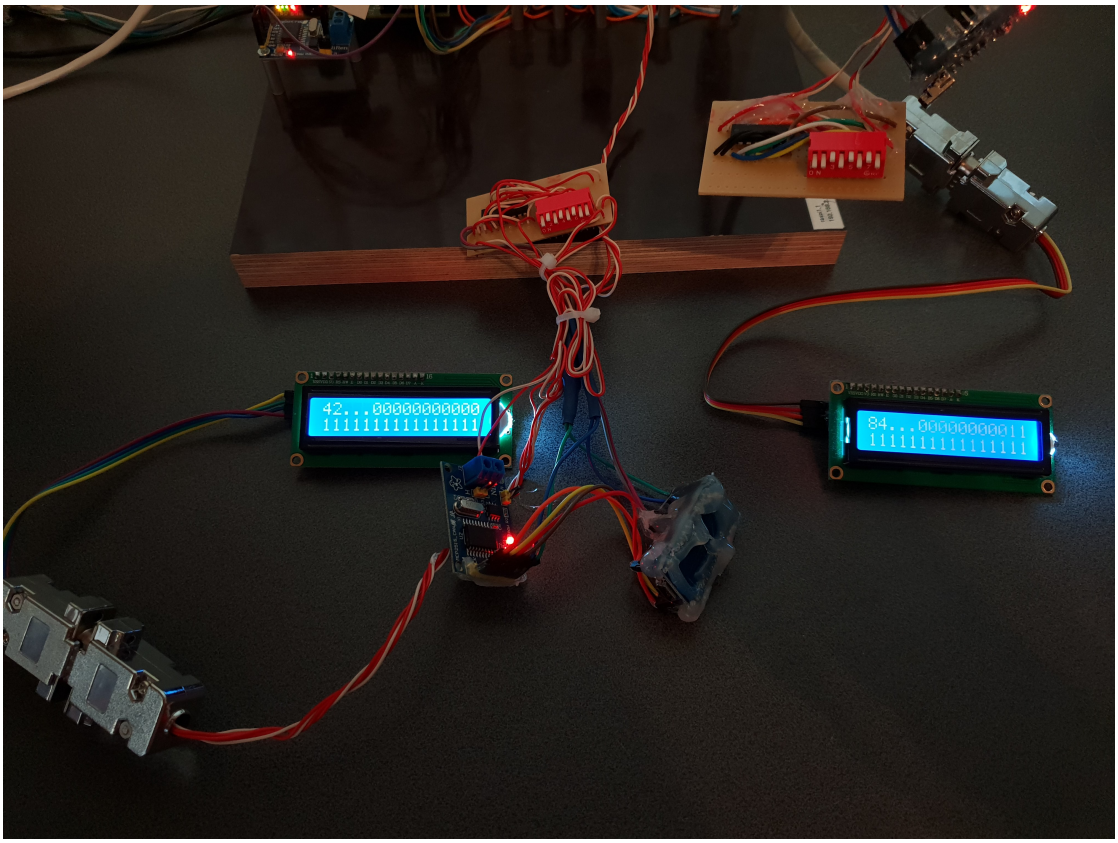


Abbildung 60: Aufbau der Messreihe 4

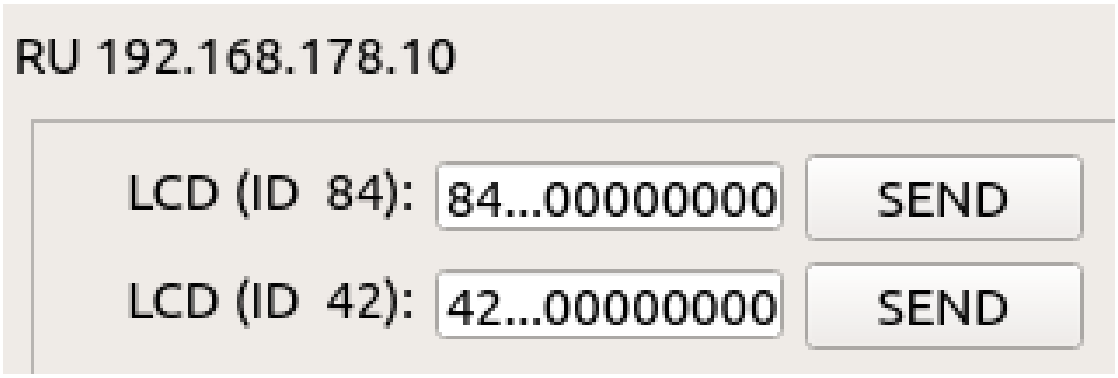


Abbildung 61: Anzeige innerhalb der GUI bei Messreihe 4

8.1.5 Messreihe 5: Aktoren und Sensoren mit und ohne CAN-I^{2}C-Bridge

Innerhalb dieser Messreihe werden jeweils eine CAN-I^{2}C-Bridge mit einem LCD sowie einem LM70 und per I^{2}C ein LM70, ein TSL2561 und ein LCD angeschlossen. Somit sind alle fünf Anschlüsse belegt, per CAN-I^{2}C-Bridge ein Aktor und ein Sensor verbunden und per I^{2}C zwei Sensoren und ein Aktor angeschlossen.
Es wird zunächst der Sensor-Thread gestartet, woraufhin nacheinander

1. LM70 per CAN-I^{2}C-Bridge mit CAN-I^{2}C-Bridge ID 84
2. LCD per CAN-I^{2}C-Bridge mit CAN-I^{2}C-Bridge ID 42
3. LM70 per I^{2}C
4. TSL2561 per I^{2}C
5. LCD per I^{2}C

angeschlossen werden. Es wird zunächst ein Text auf das per CAN-I^{2}C-Bridge angeschlossene („42ABCDEFGHIJKLMNOPQRSTUVWXYZ1234“) und anschließend auf das per I^{2}C verbundene („I2C.lcd.1234567890AbCdEfGhIjKlMn“) LC-Display gesendet. Anschließend wird der SensorThread gestoppt.

Erwartetes Ergebnis

Es wird erwartet, dass die Telemetriepakete - abgesehen von den Acknowledgements für das Starten und Stoppen des SensorThreads sowie das zweimalige Schreiben auf ein LCD - nacheinander Informationen zu einem LM70 (mit CAN-I^{2}C-Bridge-ID 0x54), einem LCD (mit CAN-I^{2}C-Bridge-ID, APID 0x0001, Service Type 0x81 und Service Subtype 0x2A) sowie per I^{2}C angeschlossenen LM70, TSL2561 und einem LCD (mit APID 0x0001, Service Type 0x80 und Service Subtype 0x85) enthalten. Weiterhin sollen alle genannten Sensoren und Aktoren ein entsprechendes Anzeigefeld innerhalb der GUI erhalten und plausible Werte enthalten.

Messprotokoll

In Abbildung 62 wird das Messprotokoll als Screenshot der mithilfe von Excel aufbereiteten CSV-Datei gezeigt. Erkennbar ist - neben den Acknowledgements für das Starten und Stoppen des SensorThread sowie das Schreiben von Text auf die beiden LCDs -, dass zunächst keine Informationen zu Sensoren und Aktoren in der Application Data auftauchen. In Zeile 11 sind Informationen zum LM70 in dreifacher Ausführung zu erkennen, woraufhin diese Information nur noch einmal vorhanden ist. In den folgenden Telemetriepaketen kommen nacheinander Informationen zum per CAN-I^{2}C-Bridge angeschlossenen LCD (Typ 0x0D, APID 0x0001,

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
1	Receive Time	IP	APID	ServiceType	ServiceSubType	SeqCount	AppData[0]	...																		
2	10.11.2018 18:38:50	192.168.178.10	1	1	1	0	18	1	c0	1																
3	10.11.2018 18:38:50	192.168.178.10	1	1	7	1	18	1	c0	1																
4	10.11.2018 18:38:50	192.168.178.10	1	80	97	0																				
10	10.11.2018 18:38:57	192.168.178.10	1	80	97	6																				
11	10.11.2018 18:38:59	192.168.178.10	1	80	97	7	0a	54	18	0a	54	18	0a	54	18	0a	54	17								
12	10.11.2018 18:39:00	192.168.178.10	1	80	97	8	0a	54	18																	
19	10.11.2018 18:39:08	192.168.178.10	1	80	97	000f	0a	54	18																	
20	10.11.2018 18:39:09	192.168.178.10	1	80	97	10	0d	0	1	81	2a	0a	54	18												
23	10.11.2018 18:39:12	192.168.178.10	1	80	97	13	0d	0	1	81	2a	0a	54	18												
24	10.11.2018 18:39:14	192.168.178.10	1	80	97	14	0d	0	1	81	2a	0a	54	18	1	0										
35	10.11.2018 18:39:26	192.168.178.10	1	80	97	001f	0d	0	1	81	2a	0a	54	18	3	0	0	0	24	1	18					
39	10.11.2018 18:39:33	192.168.178.10	1	80	97	23	0d	0	1	81	2a	0a	54	18	3	0	0	0	1b	1	18					
40	10.11.2018 18:39:34	192.168.178.10	1	80	97	24	0d	0	1	81	2a	0a	54	18	7	0	1	80	85	3	0	0	0	1e	1	18
63	10.11.2018 18:40:08	192.168.178.10	1	80	97	003b	0d	0	1	81	2a	0a	54	18	7	0	1	80	85	3	0	0	0	1d	1	18
64	10.11.2018 18:40:08	192.168.178.10	1	1	1	0	18	1	c0	0																
65	10.11.2018 18:40:08	192.168.178.10	1	1	7	1	18	1	c0	0																
66	10.11.2018 18:40:09	192.168.178.10	1	80	97	003c	0d	0	1	81	2a	0a	54	18	7	0	1	80	85	3	0	0	0	1d	1	18
82	10.11.2018 18:40:32	192.168.178.10	1	80	97	004c	0d	0	1	81	2a	0a	54	18	7	0	1	80	85	3	0	0	0	1c	1	18
83	10.11.2018 18:40:34	192.168.178.10	1	1	1	2	18	1	c0	0																
84	10.11.2018 18:40:34	192.168.178.10	1	1	7	3	18	1	c0	0																
85	10.11.2018 18:40:34	192.168.178.10	1	80	97	004d	0d	0	1	81	2a	0a	54	18	7	0	1	80	85	3	0	0	0	1c	1	18
102	10.11.2018 18:40:59	192.168.178.10	1	80	97	005e	0d	0	1	81	2a	0a	54	18	7	0	1	80	85	3	0	0	0	1c	1	18
103	10.11.2018 18:41:00	192.168.178.10	1	1	1	2	18	1	c0	1																
104	10.11.2018 18:41:00	192.168.178.10	1	1	7	3	18	1	c0	1																

Abbildung 62: Telemetriepakete bei Messreihe 5

Service Type 0x81, Service Subtype 0x2A; entspricht ID 42), dem TSL2561 (Typ 0x03) und dem per I^2C angeschlossenen LCD (Typ 0x07, APID 0x0001, Service Type 0x80 und Service Subtype 0x85) an.
Die GUI (Abbildung 63) zeigt die den Sensoren und Aktoren zugehörigen Anzeigeelemente. Im Messaufbau (Abbildung 64) sind - nach Anschluss aller Sensoren und Aktoren sowie dem Senden der Nachrichten an die LCDs - die angezeigten Texte („42ABCDEFGHIJKLM“ / „NOPQRSTUVWXYZ1234“ und „I2C.lcd.12345678“ / „90AbCdEfGhIjKlMn“) zu erkennen. Anzumerken ist, dass nach dem Anschluss des LM70 per CAN-I^2C-Bridge etwa eine Sekunde lang zwei Anzeigeelemente für den genannten Temperatursensor erkennbar waren. Weiterhin ist in Zeile 24 erkennbar, dass der erste gemessene Temperaturwert 0°C beträgt.

Auswertung

Die empfangenen Telemetriepakete entsprachen den Erwartungen mit der Ausnahme, dass das Telemetriepaket in Zeile 11 den Sensor dreifach anzeigt. In diesem Messdurchgang führte dies kurz zu einer doppelten Anzeige. Dieses kann im Rahmen einer Initialisierungsphase hingenommen und sollte entsprechend für den Anwender vermerkt werden. Ebenfalls fehlerhaft ist die Temperaturanzeige von 0°C, die ebenfalls nur eine Sekunde andauerte. Dies ist - wie bereits in der ersten Messreihe - wahrscheinlich auf eine Initialisierung des Sensors zurückzuführen. Die weitere Anzeige von Bedienelementen entsprach den Erwartungen: Alle Sensoren

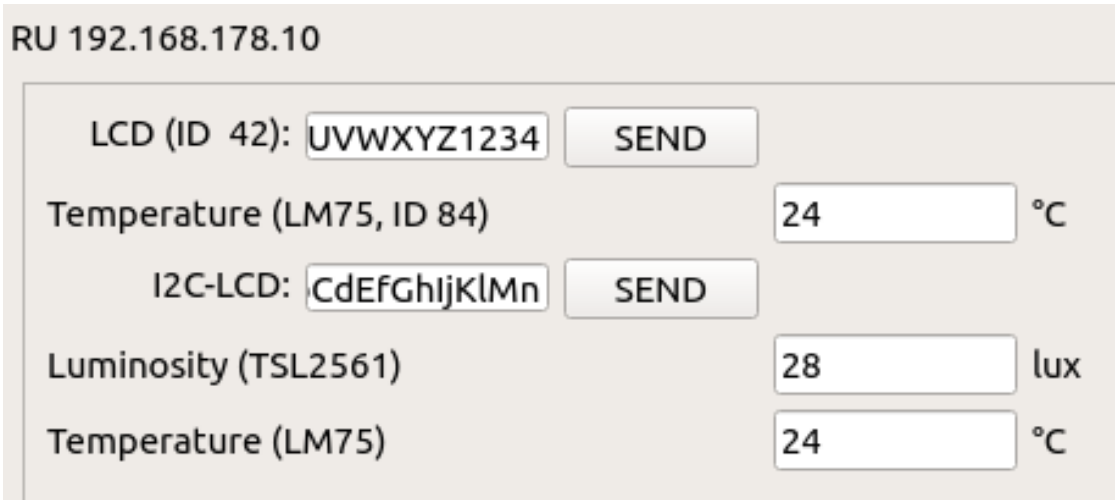


Abbildung 63: Anzeigeelemente in der GUI bei Messreihe 5

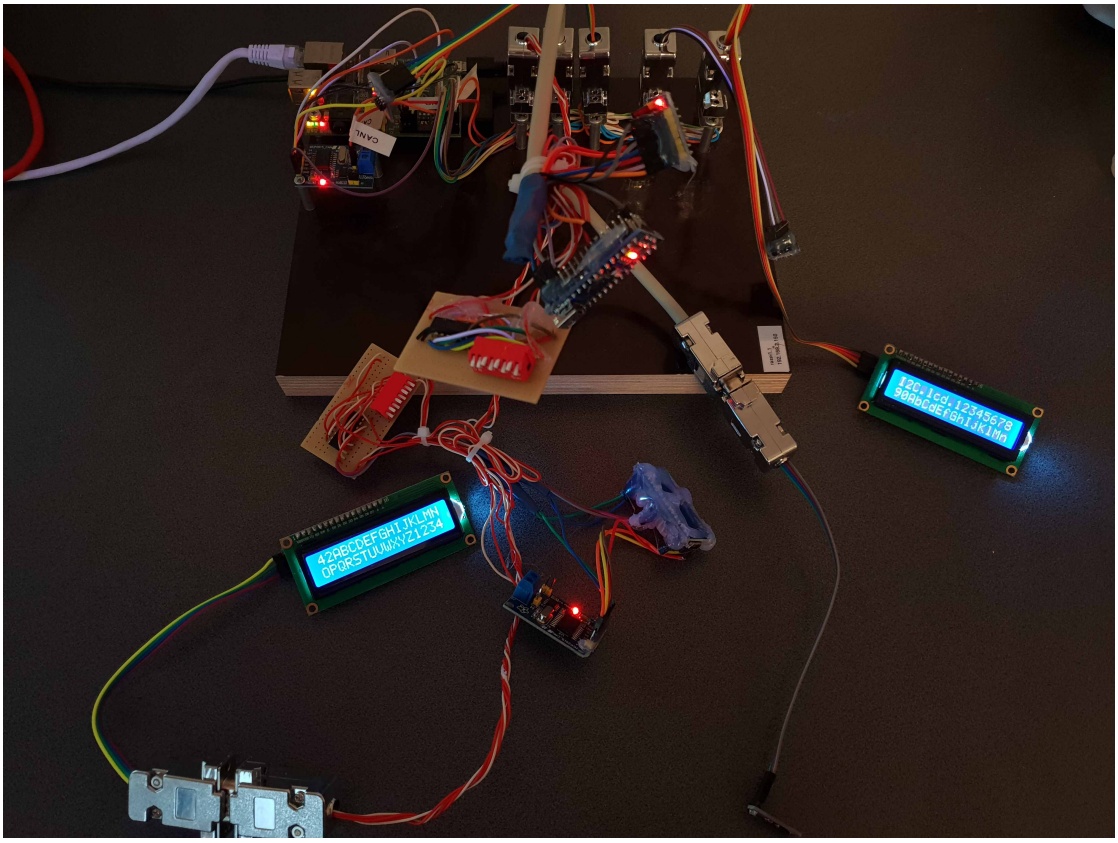


Abbildung 64: Aufbau bei Messreihe 5

und Aktoren wurden korrekt angezeigt und lieferten plausible Werte. Ebenfalls korrekt war die Anzeige der übertragenen Texte an die LC-Display.
Der Test wurde erfolgreich absolviert.

8.2 Service Discovery auf Netzebene

Die Service-Discovery soll auf Netzebene alle Remote Units im Subnetz finden und deren Daten korrekt in der GUI anzeigen. Es sollen die Aktoren angesprochen werden, die durch die Anzeige im Widget einer Remote Unit zugeordnet wurden.

8.2.1 Messreihe 6: Drei RUs mit Sensoren und Aktoren per CAN und I²C

Die Service-Discovery auf Netzebene wird getestet, indem drei Remote Units mit Sensoren und Aktoren, teilweise mit CAN-I²C-Bridges verbunden werden und die empfangene Telemetrie mit dem Erwartungswert verglichen wird. Die Sensoren sind bereits mit dem Aufbau verbunden, bevor ein Service Discovery Command geschickt wird. Anschließend wird der Sensor-Thread gestartet. In diesem Testdurchlauf wird die bisher nicht getestete Tiny RTC einmal per CAN-I²C-Bridge und einmal per I²C genutzt. Es wird auf alle drei (zwei per I²C, eines per CAN-I²C-Bridge) angeschlossenen LC-Displays ein Text geschrieben, der auf die angeschlossene Remote Unit schließen lässt. Hierzu wird der letzte Teil der Remote-Unit-IP in den gesendeten Text eingebunden.

Erwartetes Ergebnis

Es wird erwartet, dass alle drei Remote Units mit jeweils zwei Acknowledgements auf das Service Discovery Command reagieren und die GUI für jede ein Widget anlegt. Nach dem Starten des Sensor-Threads zeigen alle Widgets die Anzeigeelemente zu den Sensoren und Aktoren, die mit den Remote Units verbunden sind. Jedes der LCDs zeigt einen Text, der korrekt auf die IP-Adresse der Remote Unit oder auf die ID der CAN-I²C-Bridge schließen lässt.
Es wird erwartet, dass die Telemetriepakete - abgesehen von den Acknowledgements für das Starten des SensorThreads sowie das dreimalige Schreiben auf ein LCD - Informationen zu den verwendeten Sensoren, Aktoren und I²C-Bridges enthalten. Nach dem Service-Discovery-Command folgte eine Pause von mehreren Sekunden, sodass eine Pause auch in den eingegangenen Telemetriepaketen erkennbar ist.

Messprotokoll

Abbildung 65 zeigt sechs Acknowledgements für das empfangene Service-Discovery-Command, davon jeweils zwei pro Remote Unit. Ebenso zeigen sich etwa 30 Sekunden später jeweils zwei Acknowledgements für das Starten des Sensor-Threads. Die Telemetriepakete haben die erwartete Form. Dasselbe gilt für die Acknowledgements für das Schreiben auf die LCDs.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	AA	AB	AC	AD	AE	AF	AG	AH	AI	AJ	AK
1	Receive Time	IP	APID	ServiceType	ServiceSubType	SeqCount	AppData[0]	...																														
2	18.11.2018 01:29:45	192.168.178.26	1	1	1	6	18	1	c0	1																												
13	18.11.2018 01:30:14	192.168.178.27	1	1	7	9	18	1	c0	1																												
14	18.11.2018 01:30:15	192.168.178.26	1	80	97	18	16	54	26	14	1	12	0b	12	7	0	1	80	85	1	18	9	0	44	0	62	ff	18	0f	44	81	83	8c	41	bf	ce	5b
15	18.11.2018 01:30:15	192.168.178.27	1	80	97	18	7	0	1	80	85	15	0	1d	1	12	0b	12	0f	44	81	7e	e3	41	b3	fe	57										
16	18.11.2018 01:30:15	192.168.178.10	1	80	97	17	7	0	1	80	85	3	0	0	0	16	1	1a	0f	44	81	a8	fa	41	cf	5a	32										
56	18.11.2018 01:30:46	192.168.178.26	1	80	97	26	16	54	14	1f	1	12	0b	0	7	0	1	80	85	1	18	9	0	44	0	60	ff	18	0f	44	81	80	3d	41	bf	d8	c0
57	18.11.2018 01:30:47	192.168.178.27	1	80	97	26	7	0	1	80	85	15	1f	1d	1	12	0b	12	0f	44	81	7e	3d	41	b4	8	c6										
58	18.11.2018 01:30:47	192.168.178.10	1	80	97	25	0a	31	1a	7	0	1	80	85	3	0	0	0	18	1	1a	0f	44	81	a7	27	41	cf	12	6a							
59	18.11.2018 01:30:48	192.168.178.26	1	1	1	2	18	1	c0	0																											
60	18.11.2018 01:30:49	192.168.178.26	1	1	7	3	18	1	c0	0																											
61	18.11.2018 01:30:49	192.168.178.26	1	80	97	27	16	54	16	1f	1	12	0b	12	7	0	1	80	85	1	18	9	0	45	0	61	ff	19	0f	44	81	83	45	41	bf	c3	f7
62	18.11.2018 01:30:49	192.168.178.27	1	80	97	27	7	0	1	80	85	15	22	1d	1	12	0b	12	0f	44	81	7f	68	41	b4	3c	f2										
63	18.11.2018 01:30:49	192.168.178.10	1	80	97	26	0a	31	1a	7	0	1	80	85	3	0	0	0	16	1	1a	0f	44	81	a6	4a	41	cf	12	6a							
64	18.11.2018 01:30:51	192.168.178.26	1	80	97	28	16	54	18	1f	1	12	0b	12	7	0	1	80	85	1	18	9	0	46	0	61	ff	19	0f	44	81	80	ca	41	bf	ed	89
97	18.11.2018 01:31:16	192.168.178.26	1	80	97	33	16	54	31	1f	1	12	0b	12	7	0	1	80	85	1	18	9	0	44	0	62	ff	19	0f	44	81	81	f7	41	bf	f7	ee
98	18.11.2018 01:31:16	192.168.178.27	1	80	97	33	7	0	1	80	85	15	1	1e	1	12	0b	12	0f	44	81	80	0e	41	b4	32	83										
99	18.11.2018 01:31:16	192.168.178.10	1	80	97	32	0a	31	1a	7	0	1	80	85	3	0	0	0	18	1	1a	0f	44	81	a7	97	41	cf	45	af							
100	18.11.2018 01:31:18	192.168.178.10	1	1	1	2	18	1	c0	0																											
101	18.11.2018 01:31:18	192.168.178.10	1	1	7	3	18	1	c0	0																											
102	18.11.2018 01:31:18	192.168.178.26	1	80	97	34	16	54	33	1f	1	12	0b	12	7	0	1	80	85	1	19	9	0	44	0	61	ff	1a	0f	44	81	83	34	41	bf	e3	25
103	18.11.2018 01:31:18	192.168.178.27	1	80	97	34	7	0	1	80	85	15	3	1e	1	12	0b	12	0f	44	81	80	0e	41	b4	32	83										
104	18.11.2018 01:31:19	192.168.178.10	1	80	97	33	0a	31	1a	7	0	1	80	85	3	0	0	0	16	1	1a	0f	44	81	a8	75	41	cf	45	af							
114	18.11.2018 01:31:27	192.168.178.26	1	80	97	38	16	54	0	20	1	12	0b	12	7	0	1	80	85	1	19	9	0	44	0	61	ff	1a	0f	44	81	81	f7	41	bf	f7	ee
115	18.11.2018 01:31:27	192.168.178.27	1	80	97	38	7	0	1	80	85	15	0c	1e	1	12	0b	12	0f	44	81	7e	dd	41	b4	9a	da										
116	18.11.2018 01:31:27	192.168.178.10	1	80	97	37	0a	31	1a	0a	31	1a	7	0	1	80	85	3	0	0	0	18	1	1a	0f	44	81	a6	e8	41	cf	6e	b4				
117	18.11.2018 01:31:29	192.168.178.27	1	1	1	2	18	1	c0	0																											
118	18.11.2018 01:31:30	192.168.178.27	1	1	7	3	18	1	c0	0																											
119	18.11.2018 01:31:30	192.168.178.26	1	80	97	39	16	54	2	20	1	12	0b	12	7	0	1	80	85	1	19	9	0	45	0	60	ff	19	0f	44	81	81	47	41	c0	21	81
120	18.11.2018 01:31:30	192.168.178.27	1	80	97	39	7	0	1	80	85	15	0e	1e	1	12	0b	12	0f	44	81	7f	83	41	b4	90	6b										
121	18.11.2018 01:31:30	192.168.178.10	1	80	97	38	0a	31	1a	7	0	1	80	85	3	0	0	0	16	1	1a	0f	44	81	a7	6d	41	cf	83	36							
127	18.11.2018 01:31:34	192.168.178.26	1	80	97	003b		16	54	7	20	1	12	0b	12	7	0	1	80	85	1	18	9	0	44	0	61	ff	19	0f	44	81	81	9f	41	c0	0c	b8
128	18.11.2018 01:31:35	192.168.178.27	1	80	97	003b		7	0	1	80	85	15	13	1e	1	12	0b	12	0f	44	81	7f	55	41	b4	af	b8									
129	18.11.2018 01:31:35	192.168.178.10	1	80	97	003a	0a	31	1a	7	0	1	80	85	3	0	0	0	16	1	1a	0f	44	81	a5	9a	41	cf	3b	6e							

Abbildung 65: Telemetriepakete bei Messreihe 6

In Abbildung 66 ist zu erkennen, dass für jede Remote Unit ein Widget erstellt wurde (das der Übersicht halber noch von Hand nach rechts verschoben wurde). Die Widgets zeigen die verwendeten Sensoren sowie plausible Daten. Das Absenden der Texte für die LCDs führte zur Anzeige eines Textes, aus dem die letzte Stelle der IP der jeweiligen Remote Unit hervorgeht.

Auswertung

Die Anzeige der zu den Remote Units gehörenden Widgets entsprach den Erwartungen. Ebenfalls wurden alle Sensoren mit plausiblen Messwerten und Aktoren korrekt angezeigt. Das Bedienen der Aktoren führte zum erwarteten Ergebnis.

Der Test wurde erfolgreich absolviert.

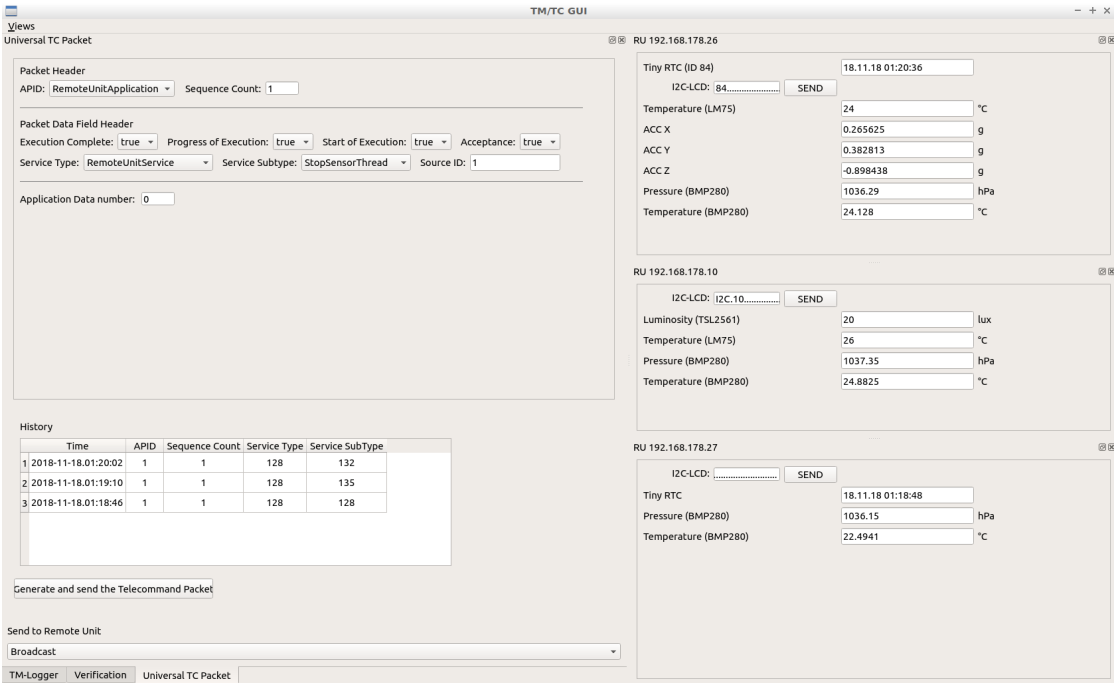


Abbildung 66: Anzeige innerhalb der GUI bei Messreihe 6

8.3 Zusammenfassung der Ergebnisse

Innerhalb dieses Kapitels wurde gezeigt, dass die Anforderungen an das „Service-Discovery-Protokoll für Netze heterogener Sensoreinheiten“ erfüllt wurden. In verschiedenen Messreihen wurde gezeigt, dass die Service Discovery sowohl auf Remote-Unit-Ebene als auch auf Netzebene funktioniert. Sowohl Sensoren als auch Aktoren und der Anschluss per I^2C und per CAN-I^2C-Bridge wurden überprüft.

Das Protokoll zur Service-Discovery für Netze heterogener Sensoreinheiten wurde erfolgreich getestet.

9 Inbetriebnahme

In diesem Abschnitt werden die notwendigen Schritte beschrieben, um die in dieser Arbeit verwendete Hard- und Software in Betrieb zu nehmen. Insbesondere wird hierbei auf das Kompilieren der Software sowohl auf GUI- als auch auf RU-Seite eingegangen.
Für das Klonen des Git-Repositories ist ein FB3-Account / Uni-Account der Universität Bremen[27] sowie eine Erlaubnis des Autors für den Zugriff auf das Repository[28] notwendig.

9.1 GUI

Um die GUI zu kompilieren und auszuführen, werden im Folgenden die hierfür notwendigen Schritte angegeben. Als Betriebssystem wird in dieser Beschreibung ein neu installiertes Lubuntu 18.10[29] verwendet. Dieses läuft innerhalb einer virtuellen Maschine mit VMware Workstation 14 Player sowie installiertem Paket „open-vm-tools“.

Innerhalb eines Terminals werden zum Kompilieren notwendige Pakete installiert.

```
dev@dev-pc:~$ sudo apt-get install scons qt4-qmake libqt4-dev
```

Anschließend wird das Repository dieser Arbeit geklont.

```
dev@dev-pc:~$ mkdir git
dev@dev-pc:~$ cd git
dev@dev-pc:~/git$ git clone https://gitlab.informatik.uni-bremen.de/struck/
    raspi-demo.git
```

Das Kompilieren und Linken der GUI erfordert die kompilierten Outpost-Bibliotheken. Hierfür wird in den Ordner der GUI gewechselt und anschließend werden die Bibliothek-Dateien erstellt.

```
dev@dev-pc:~/git$ cd raspi-demo/gui/tmtc_gui/
dev@dev-pc:~/git/raspi-demo/gui/tmtc_gui$ scons
```

Anschließend wird mithilfe von qmake das Makefile erstellt und make aufgerufen.

```
dev@dev-pc:~/git/raspi-demo/gui/tmtc_gui$ qmake -qt=5 tmtcGUI.pro
dev@dev-pc:~/git/raspi-demo/gui/tmtc_gui$ make
```

Nach diesem Prozess wurde eine ausführbare Datei „tmtcGUI“ erstellt, die mithilfe des folgenden Befehls die GUI startet.

```
dev@dev-pc:~/git/raspi-demo/gui/tmtc_gui$ ./tmtcGUI
```

[27] siehe http://www.informatik.uni-bremen.de/t/Accounts
[28] Für den Zugriff wird um eine E-Mail an malte.struck@uni-bremen gebeten.
[29] http://cdimage.ubuntu.com/lubuntu/releases/cosmic/release/lubuntu-18.10-desktop-amd64.iso

9.2 Remote Unit

In diesem Abschnitt wird beschrieben, wie die Remote Unit mithilfe eines Raspberry Pi in Betrieb genommen wird. Zunächst werden die notwendigen Schritte dargelegt, den Raspberry Pi bezüglich der Hardware auszurüsten, d.h. es wird gezeigt, wie einzelne Leitungen des GPIO-Anschlusses genutzt werden. Diese dienen zum Anschluss des CAN-Controllers (mit -Transceiver), der Spannungsversorgung der Sensoren (oder CAN-I^2C-Bridges) sowie der Bereitstellungen der I^2C-Leitungen. Beispielhaft wird eine SUB-D-Buchse zum Anschluss eines Sensors oder einer Bridge genutzt.
Anzumerken ist, dass die GPIO-Pins eines Raspberry Pi, Modell 1B, gezeigt werden. Da alle dort vorhandenen Leitungen auch an einem Raspberry Pi 2 oder 3 vorhanden sind, ist der Anschluss dort identisch.

9.2.1 Vorbereitung der SD-Karte

Zum Bespielen der SD-Karte unter Windows wurde das Programm *Win32 Disk Imager* genutzt, welches unter `https://www.heise.de/download/product/win32-disk-imager-92033` kostenlos heruntergeladen werden kann. Als Image wurde Raspbian Stretch Lite (basierend auf Debian Stretch) vom April 2018 genutzt. Ein aktuelles Image ist unter `https://www.raspberrypi.org/downloads/raspbian/` zu finden. Mithilfe von Win32 Disk Imager wird dieses Image auf eine SD-Karte kopiert. Verwendet wurde eine 16GB-SD-Karte. Abbildung 67 zeigt die Oberfläche des Programms beim Beschreiben der SD-Karte. Damit der Raspberry Pi per SSH bedient werden kann, muss im Wurzelverzeichnis der SD-Karte (genaugenommen der FAT32-Partition mit der Bezeichnung „boot“) eine leere Datei mit dem Namen „ssh“ angelegt werden, da SSH bei Raspbian standardmäßig abgeschaltet ist. Die nun vorbereitete SD-Karte kann in den Raspberry Pi eingelegt werden, woraufhin dieser durch Einstecken der Stromversorgung gestartet wird.

9.2.2 Vorbereiten von Linux

Nachdem der Raspberry Pi mit Strom versorgt und mit dem lokalen Netz verbunden wurde, muss dessen IP-Adresse etwa durch Ablesen aus der DHCP-Lease-Tabelle des Routers ermittelt werden, sodass eine Verbindung per SSH aufgebaut werden kann. Unter Windows wird hierfür das Programm „PuTTY“ verwendet. Nach Anerkennung des (neuen) SSH-Fingerprints kann ein Login mit dem Benutzernamen „pi“ und dem Passwort „raspberry“ erfolgen:

```
login as: pi
pi@192.168.2.104's password:
Linux raspberrypi 4.14.34+ #1110 Mon Apr 16 14:51:42 BST 2018 armv6l
```

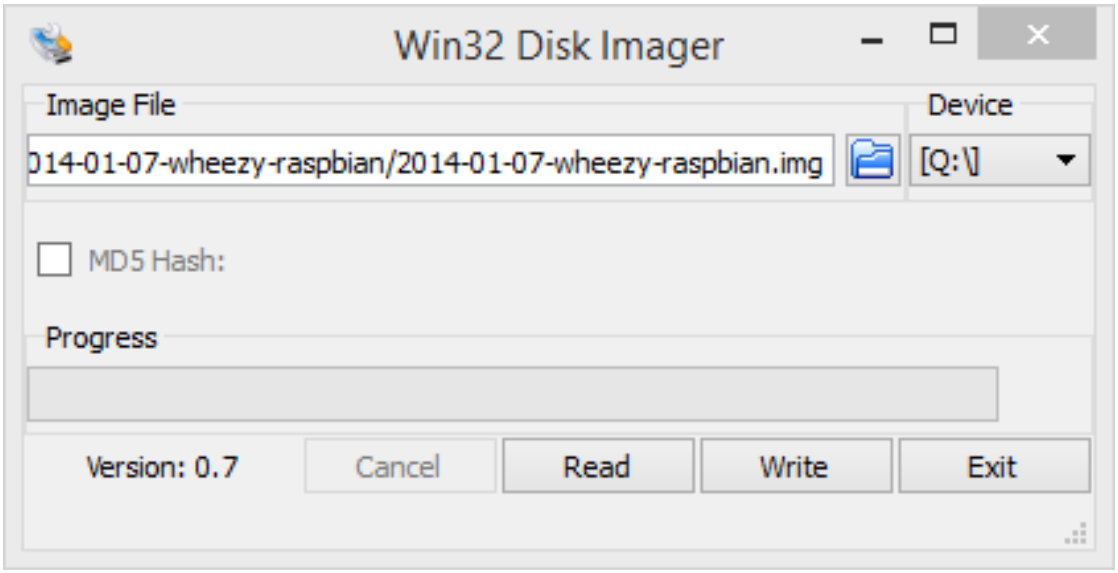


Abbildung 67: Das Programm Win32 Disk Imager

```
The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.

SSH is enabled and the default password for the 'pi' user has not been
    changed.
This is a security risk - please login as the 'pi' user and type 'passwd' to
    set a new password.

pi@raspberrypi:~ $
```

Nun müssen die I^2C- sowie die SPI-Schnittstelle aktiviert werden, indem die Datei `/boot/config.txt` (etwa mit dem Editor „nano“) mit Root-Rechten editiert wird:

```
pi@raspberrypi:~ $ cd /boot/
pi@raspberrypi:/boot $ sudo nano config.txt
```

Die Rauten zu Beginn der Zeilen

```
#dtparam=i2c_arm=on
#dtparam=spi=on
```

müssen entfernt werden, sodass sie die entsprechenden Zeilen nicht mehr als Kommentar markieren. Somit stehen die beiden genannten Schnitstellen nach einem Neustart zur Verfügung. Damit die CAN-Schnittstelle als Netzwerkinterface genutzt werden kann, müssen weiterhin folgenden Zeilen in die Datei eingefügt werden:

```
dtoverlay=mcp2515-can0,oscillator=8000000,interrupt=25
dtoverlay=spi-bcm2835-overlay
```

Anschließend muss die Datei `/etc/network/interfaces` um folgende Zeilen erweitert werden, sodass der CAN-Controller beim Hochfahren automatisch als Netzwerkgerät initialisiert wird.

```
auto can0
iface can0 can static
        bitrate 500000
```

Nach einem Neustart mit

```
pi@raspberrypi:~ $ sudo reboot now
```

oder einer kurzen Trennung des Raspberry Pi vom Stromnetz ist dieser nun für die folgenden Schritte vorbereitet.

9.2.3 Softwarepakete, Klonen und Kompilieren

Standardmäßig beinhaltet Raspbian Stretch Lite nicht alle zum Kompilieren benötigten Bibliotheken und Software-Werkzeuge. Diese müssen mithilfe von Konsolenbefehlen nachinstalliert werden. Die Versionsverwaltung *git*, das Build-Tool *scons* sowie die Library *libi2c-dev* können mithilfe des Paketmanagers `apt` nachinstalliert werden, die Bibliothek *wiringPi* muss von Github geklont, kompiliert und installiert werden.

```
pi@raspberrypi:~ $ sudo apt-get update
pi@raspberrypi:~ $ sudo apt-get install git scons libi2c-dev
pi@raspberrypi:~ $ mkdir git
pi@raspberrypi:~ $ cd git
pi@raspberrypi:~/git $ git clone git://git.drogon.net/wiringPi
pi@raspberrypi:~/git $ cd wiringPi
pi@raspberrypi:~/git/wiringPi $ ./build
```

Mithilfe des Befehls

```
pi@raspberrypi:~/git/wiringPi $ sudo modprobe i2c-dev
```

wird das I^{2}C-Modul auf dem Raspberry Pi gestartet. Durch

```
pi@raspberrypi:~/git/wiringPi $ cd ~/git
pi@raspberrypi:~/git $ git clone https://gitlab.informatik.uni-bremen.de/
    struck/raspi-demo.git
```

wird das Projekt in den Ordner ~/git/raspi-demo geklont. Nachdem das Repository geklont wurde, kann es mithilfe von

```
pi@raspberrypi:~/git $ cd raspi-demo/ru/
pi@raspberrypi:~/git/raspi-demo/ru $ scons
```

kompiliert werden. Dies kann, insbesondere auf dem Einkern-Rechner Raspberry Pi 1, mehrere Minuten in Anspruch nehmen.

9.2.4 Starten der Software

Um die Software zu starten, muss das kompilierte Binary-File ru gestartet werden, welches sich im Unterordner ./build/ befindet. Dies gelingt mithilfe von

```
pi@raspberrypi:~/git/raspi-demo/ru $ ./build/ru
```

Anschließend wartet der UDP-Server der Remote Unit auf eingehende Verbindungen von der GUI.

9.3 Fehlerbehandlung

Installieren von Packages in der VM schlägt fehl
Bei der benutzten Version von Lubuntu 18.10 kann es zu Fehlern beim Installieren der Packages (scons, qt,...) durch nicht aquirierbare Locks kommen. Abhilfe findet sich unter https://itsfoss.com/could-not-get-lock-error/, bei der zunächst störende Prozesse ausfindig gemacht und anschließend beendet werden.

Segmentation Fault bei Ausführung der GUI
Beim Ausführen der GUI kann es zu einem Segmentation Fault kommen, dessen Ursache in Zusammenhang mit dem laufenden X-Server sowie der Grafikschnittstelle OpenGL steht[30]. Abhilfe schafft das Starten der GUI innerhalb des Debuggers GDB[31]

```
dev@dev-pc:~/git/raspi-demo/gui/tmtc_gui$ gdb tmtcGUI.pro
...
(gdb) r
```

[30] Dies ist in der Ausgabe des Memory-Leak-Checkers Valgrind zu erkennen.
[31] GDB kann mit sudo apt-get install gdb installiert werden. Dass die Ausführung innerhalb von GDB keinen Segmentation Fault zur Folge hat, ist zum Einen der Grund dafür, dass dessen Ursache nicht exakt lokalisierbar ist, und zum Anderen ein Workaround. Anmerkung: Das Starten der GUI innerhalb von Valgrind führt zu einem Segmentation Fault von Valgrind (sic!).

Telekommandos erreichen die Remote Unit nicht

VMWare nutzt standardmäßig NAT, um eine Verbindung nach außen zu schaffen. Hierbei erhält die VM eine IP-Adresse von VMWare aus einem Subnetz, das sich von dem der Host-Maschine unterscheidet. Im Menü (von VMWare) kann unter Player, Removable Devices, Network Adapter, Settings die Variante „Bridged“ eingestellt werden, bei der die VM eine IP-Adresse von demselben DHCP-Server wie die Host-Maschine erhält. Alternativ kann mit einem Rechtsklick auf das Netzwerksymbol unten rechts in der Info-Leiste (der VM) und einem Klick auf „edit connections“ manuell eine IP-Adresse aus dem Subnetz der Remote Unit eingetragen werden.

I²C-Fehler auf der Remote Unit

Es kann bei der Remote Unit vorkommen, dass die Verbindung zum I²C-Bus bei einem Neustart nicht hergestellt wird. Dies zeigt sich durch die Meldung

```
pi@raspberrypi:~/git/raspi-demo/ru $ ./build/ru
RaspiClock-Konstruktor
RemoteUnitService-Konstruktor
SensorScanner-Konstruktor
Failed to open the bus.pi@raspberrypi:~/git/raspi-demo/ru $
```

welche angibt, dass ein Zugriff auf den I²C-Bus nicht möglich ist. Mithilfe der erneuten Eingabe von

```
pi@raspberrypi:~/ $ sudo modprobe i2c-dev
```

kann das Problem gelöst werden.

Fehler beim Einbinden des CAN-Bus der Remote Unit

Bei Remote Units kann es vorkommen, dass die automatische Einbindung des CAN-Controllers als Netzwerkgerät nicht zuverlässig durchgeführt wird. Mithilfe des Befehls

```
pi@raspberrypi:~/ $ ifconfig
```

werden angeschlossene Netzwerkadapter angezeigt. Befindet sich hierunter keiner mit der Bezeichnung `can0`, kann die Verbindung hierzu manuell mithilfe von

```
pi@raspberrypi:~/ $ sudo ip link set can0 up type can bitrate 500000
```

hergestellt werden.

Fehler beim Schreiben auf den CAN-Bus der Remote Unit

Wenn an der Remote Unit keine CAN-I²C-Bridge angeschlossen ist und gleichzeitig der SensorThread gestartet wurde, führt dies dazu, dass der Schreibpuffer für den CAN-Bus vollläuft.

Wird nun eine CAN-I^{2}C-Bridge angeschlossen, werden die Pakete aus dem Puffer gesendet. Ist dies nicht der Fall und lautet die entsprechende Ausgabe

```
CanSender::sendPacket id = 000000FF, data = (nil), length = 0
write: No buffer space available
```

so kann die CAN-Verbindung mithilfe von

```
pi@raspberrypi:~/ $ sudo ip link set can0 down
pi@raspberrypi:~/ $ sudo ip link set can0 up type can bitrate 500000
```

erneut hergestellt werden.

10 Fazit

Um die für diese Mission gesteckten Ziele zu erreichen, wurden verschiedene Teilschritte umgesetzt:

- Es wurde ein Protokoll entworfen, das es ermöglicht, Informationen über die angeschlossenen Sensortypen zusammen mit den Messdaten zu versenden.
- Das ursprünglich geplante Service-Discovery-Protokoll wurde erweitert, um Aktoren einzubeziehen.
- Mithilfe eines speziellen Telekommandos in einem Broadcast können alle im Subnetz vorhandenen Remote Units aufgefunden werden.
- Die GUI stellt zur Umsetzung des Protokolls dynamische Anzeigeelemente zur Verfügung: Jede Remote Unit erhält beim Auffinden ein generiertes Widget, in dem Bedienelemente zu den an der Remote Unit angeschlossenen Sensoren und Aktoren platziert werden.
- Die Einschränkung des I^2C-Busses durch einen kleinen Adressraum und nicht weitreichend konfigurierbare Adressierung wurde aufgehoben, indem eine Komponente entworfen und implementiert wurde, die es erlaubt, annähernd beliebig viele Sensoreinheiten zu nutzen.

Die durch den Titel dieser Arbeit gesetzten Ziele wurden nicht nur erfüllt, sondern in einigen Punkten übertroffen. Es finden sich verschiedene Anknüpfungspunkte, das in dieser Arbeit verfolgten Ziel der Generizität und der Abstraktion in einem weiteren Kontext zu erreichen. Mit dieser Arbeit wird eine Plattform bereitgestellt, die prototypisch Kosten reduzieren und den wissenschaftlichen Prozess deutlich vereinfachen kann, sodass

die Grenzen,
in denen wir die Welt verstehen können,
nur ein Monument unseres Zeitalters
und nicht unseres Geistes sind.

11 Ausblick

Für diese Master Thesis ergeben sich interessante Anschlussprojekte, die an den im Vorfeld sowie während der Implementierung gesetzten Grenzen des zu entwickelnden und wissenschaftlich evaluierten Produktes anknüpfen.

11.1 Verallgemeinerte Abfrage

Die in dieser Arbeit genutzten Sensoren benötigen als Zwischenschicht zwischen Service und (I^2C-) Hardware einen Treiber, d.h. mithilfe bestimmter Befehle müssen die Geräte so eingestellt werden, dass sie die Messung beginnen, durchführen und anschließend ihre Ergebnisse zurückschreiben. Dabei werden bei I^2C-Sensoren bestimmte Register mit Werten gefüllt, die etwa angeben, wie lange ein Messergebnis integriert oder mit welcher Verstärkung das Signal vor der Digitalisierung aufbereitet wird. Dies schränkt den Einsatz des Service-Discovery-Protokolls prinzipiell auf Sensoren ein, die vorher ausgesucht wurden und für die ein Treiber zur Verfügung steht.
Mithilfe einer Protokollerweiterung, die der Remote Unit (und in der Folge auch der CAN-I^2C-Bridge) mitteilt, wie bisher unbekannte bzw. nicht von vornherein definierte Sensoren abzufragen sind, könnten beliebige Sensoren verwendet werden, ohne dass ein spezieller Treiber vorhanden sein muss. Beispielhaft soll folgender Pseudocode zeigen, wie dies umzusetzen wäre. Dieses Skript könnte nun an die Remote Unit übertragen werden (oder auf die CAN-

```
1  FUNC(BYTE[8]) READ-NEW-SENSORTYPE //Function returns 4 Bytes of Data;
2  SET-TYPE 0x81 //Sensortype-ID of this Sensor is now 129;
3  TELEMETRY[8] //Reserve 8 Bytes for Data;
4  SET-I2C-ADDR 0x39;
5  SET-REG 0x01 0xF0 //Set Register #1 to value 240;
6  SET-REG 0x02 0x00;
7  READ-WORD 4 0x00 TELEMETRY //Read 4 Bytes;
8  SET-REG 0x02 0x01;
9  READ-WORD 4 0x00 TELEMETRY+4 //Read 4 more Bytes, don't overwrite;
10 RETURN TELEMETRY END FUNC
```

I^2C-Bridge), sodass ein neuer Sensortyp angeschlossen, gesteuert und ausgelesen wird. Eine entsprechend gesetzte Sensortype-ID (die vor den 8 Bytes Messdaten im Telemetriepaket übertragen wird) sorgt dafür, dass die GUI über den neu eingetragenen Sensor informiert wird.
Analog hierzu müsste ein entsprechendes Auswertungsskript auf GUI-Seite die Daten entge-

gennehmen, ggf. auswerten und umrechnen und für eine adäquate Anzeige sorgen.

11.2 Übertragungsskript

Bisher lief der Sensor-Thread so ab, dass nach dem Abfragen aller gefundenen Sensoren und dem Senden des entsprechenden Telemetriepaketes ein definierter Timeout (in dieser Arbeit 1 Sekunde) abgewartet wird, bevor eine neue Messreihe mit Datenübertragung beginnt. Dies impliziert, dass das Abfrageintervall für alle Sensoren dasselbe ist. Die Abfrage eines Temperatursensors (beispielsweise zur Überwachung der stromversorgenden Komponenten) muss sicherlich seltener geschehen als die Messung der Auslenkung eines Seismometers, für die eher eine Messhäufigkeit in der Größenordnung von kHz sinnvoll wäre. Sicherlich wäre eine Programmierung der Remote Unit möglich, sodass verschiedene Messdaten mit unterschiedlicher Frequenz gesendet würden. Dies widerspricht jedoch dem grundsätzlichen Gedanken dieser Arbeit, dem Benutzer, etwa einem wissenschaftlichen Team, eigene Programmierung (die über die eines einfach zu haltenden Skriptes hinaus geht) zu ersparen. Ein mögliches Skript könnte wie folgt aussehen.

Algorithm 7: Skript zur Abfrage eines neuen Sensors

```
1 SET-INTERVAL 100 //Repeat this script every 100ms;
2 BUFFER 16 //Allocate 16 Bytes for App-Data of Telemetry Packet;
3 SET-SERVICETYPE 140 //Set Service Type of Telemetry Packet to 140;
4 SET-SUBTYPE 1 //Set Service Subtype of Telemetry Packet to 1;
5 BUFFER += READ-NEW-SENSORTYPE //Sensor Data is stored to Buffer, Buffer
    Pointer is auto-incremented;
6 BUFFER += READ-ANOTHER-SENSORTYPE;
7 SEND-DATA //Sends Telemetry Packet to GUI;
```

Wie im Listing 7 zu erkennen ist, baut diese Variante auf den Ideen des vorangegangenen Vorschlags zur Erweiterung des Protokolls (siehe Unterabschnitt 11.1) auf. Allerdings ist dort festgelegt, dass ein Telemetriepaket stets nur einen Messdatensatz enthält. Gerade bei großen Datenmengen, etwa wie sie bei der Abfrage eines Seismometers anfallen (beispielhaft 24 Bit Genauigkeit in drei Achsen bei 1 kHz Abfragerate $\hat{=} 3 \cdot 3 \cdot 1000$ Bytes/s $= 9000$ Byte/s), ist es sinnvoller, Daten zunächst zu sammeln, um sie anschließend gemeinsam zu versenden.

Algorithm 8: Skript zum Senden mehrerer Einzelmesswerte in einem Paket

```
1  SET-INTERVAL 32//Run this Code every 32ms
2  BUFFER 288 //31 Measurements of 3 x 3 Bytes
3  SET-SERVICETYPE 140
4  SET-SUBTYPE 2
5  FOR for I IN 1 TO 32 DO do
6  |  SET-INTERVAL 1 // Run this Code every 1ms
7  |  BUFFER += READ-SEIMO-X
8  |  BUFFER += READ-SEIMO-Y
9  |  BUFFER += READ-SEIMO-Z
10 |  DELAY 0.91 //Wait for 0.91 ms (1ms - deterministic measure time)
11 END-FOR
12 SEND-DATA //Asynchronously, maybe during delay
```

11.3 Andere Hardware-Basis

Um das Service-Discovery-Protokoll einzusetzen, ist es nicht zwingend erforderlich, einen Raspberry Pi einzusetzten. Notwendig für den Einsatz sind nur ein SoC mit einer Netzwerkschnittstelle sowie Anschlüssen für Sensoren bzw. Aktoren. Beispielsweise ist hier der ESP8266 (vgl. [Sys18]) zu nennen, der ein 2,4 GHz-WLAN-Modul (bis zu 72.2 Mbps) beinhaltet und über mehrere frei programmierbare I/O-Pins verfügt, über die eine Kommunikation per I^2C möglich ist. Er ist mit einem Preis von unter 2€ pro Modul besonders kostengünstig, bietet dafür aber einen 32-Bit Prozessor (Tensilica L106), der mit bis zu 160 MHz getaktet werden kann.

11.4 Skalierbarkeit

Aufgrund der hardwareseitigen Beschränkung auf fünf Sensoren/Aktoren pro Remote Unit konnten keine Tests durchgeführt werden, bei denen mehr als diese Zahl genutzt wurde. Denkbar ist es, dass der Code angepasst werden muss, um der größeren Anzahl von Abfragen pro Durchlauf gerecht zu werden. Innerhalb des Service-Discovery-Threads wird davon ausgegangen, dass die Abfrage der Sensoren (und Feststellung der Aktoren) keine Zeit beansprucht, sodass das Messintervall durch das nach dem Senden durchgeführte `SLEEP` festgelegt ist.Ebenso wurde bei der Abfrage von per CAN-I^2C-Bridges angeschlossenen Sensoren zwar die Datenstruktur so groß gewählt, dass 255 Sensoren/Messwerte empfangen werden können, allerdings wurden die gesetzen Timeouts für das Empfangen von Nachrichten nicht bei mehr angeschlossenen Geräten getestet. Es müsste also bei großer benutzer Sensor-/Aktorzahl ge-

testet werden, ob dieser Timeout angepasst werden muss, sodass keine Pakete verloren gehen.

12 Anhang

12.1 Quellcode

Der Quellcode sowie die PDF-Version dieses Dokumentes befindet sich auf der beiliegenden CD-ROM.

Inhalt der CD-ROM

Die Ordnerstruktur auf der CD-Rom wird im Folgenden beschrieben.

```
/
├── gui
├── lib
├── mc
├── ru
└── thesis
```

- **gui** enthält den Quellcode der entwickelten GUI.
- **lib** enthält die Bibliotheken OUTPOST-CORE, OUTPOST-SATELLITE und die scons-build-utils
- **mc** enthält den Quellcode für die CAN-I^2C-Bridge
- **ru** enthält den Quellcode für die Remote Unit
- **thesis** enthält diese Arbeit im PDF-Format

12.2 Robex *Reloaded*

ROBEX *Reloaded*

Malte Christian Struck[1]

[1] **DLR e.V., Standort Bremen**

Ideensammlung zur Fortführung des Robex-Projektes

1. **Automatische Service Discovery**
   - **Die GUI soll nach Start nach allen Geräten im Subnetz suchen und anzeigen.**
   - **Ein Paket soll als Broadcast an das gesamte Subnetz gesendet werden.**
   - **Geräte antworten mit einem speziellen Telemetriepaket, aus dem deren Fähigkeiten hervorgehen**
   - **Ziel: Skalierbarkeit**
2. **Authentifizierung / Autorisierung**
   - **Die GUI soll sich bei Geräten anhand eines Zertifikats als berechtigt authentifizieren.**
   - **Geräte sollen die Echtheit anhand einer Zertifikatskette prüfen können.**
   - **Zertifikate sollen für mehrere Command Center ausgestellt werden.**
   - **Die Verbindung soll verschlüsselt sein.**
3. **Geräte für kleines Geld**
   - **Auf Basis von ESP8266 (o.Ä.) für 1-2€ pro Stück sollen viele kleine Sensoren konstruiert werden, etwa Temperatursensoren, die Batteriebetrieben einige Tage halten und in Intervallen Sensordaten übermitteln. Eckdaten ESP8266: WLAN, 160Mhz CPU, 4MByte Flash.**
4. **Android App**
   - **Mithilfe einer Android-App (etwa auf einem Tablet) soll das Robex-System bedient werden können, um nicht schweres Gerät (Toughbook) schleppen zu müssen.**
   - **Android Tablets bieten ausreichend Leistung für notwendige Berechnungen / Anzeigen**
5. **Seismometer-Implementierung**
   - **Bisher zeichnet Seismometer mit 250 Hz auf, da die Rechen- und Übertragungsleistung nicht ausreicht. Dies ist für sinnvolle Messungen zu niedrig.**
   - **Mithilfe eines RasPi könnten die nowendigen 2000Hz ermöglicht werden.**
6. **Zusammenarbeit mit konkreten Nutzern**
   - **Es soll eine Partnerschaft mit potentiellen Nutzern angestrebt werden. Etwa könnte mit dem Alfred-Wegener-Institut in Bremerhaven zusammengearbeitet werden, um zum Einen Robex an die Antarktis zu bringen und zum Anderen zu erfahren, wie die Nutzungsvorstellungen aussehen und entsprechend Inhalte des Projektes (Bedienung, Funktion) angepasst werden können.**
7. **…**

12.3 Festgelegte CAN-ID-Offsets für Sensoren und Aktoren

typeOffset (ID=LSB)	Typ
0x00 00 00 00	RU
0x00 00 01 00	LM70 (TEMP)
0x00 00 02 00	TSL2561 (LUMI)
0x00 00 03 00	ADXL345 (ACCEL)
0x00 00 04 00	SHT21 (HUMI / TEMP)
0x00 00 05 00	PCF8574 (LCD)
0x00 00 06 00	PCA9685 (PWM)
0x00 00 07 00	BMP280 (TEMP / PRES)
0x00 00 08 00	RESERVED
0x00 00 09 00	TINY RTC
0x00 00 0A 00	RESERVED
0x00 00 0B 00	RESERVED
0x00 00 0C 00	RESERVED
0x00 00 0D 00	RESERVED
0x00 00 0E 00	RESERVED
0x00 00 0F 00	RESERVED
...	
0x00 20 00 00	SET LCD BUFFER BYTE 0-7
0x00 21 00 00	SET LCD BUFFER BYTE 8-15
...	
0x00 2E 00 00	SET LCD BUFFER BYTE 112-119
0x00 2F 00 00	WRITE LCD BUFFER TO DISPLAY
0x00 30 00 00	SET CAN PWM
0x00 31 00 00	SET TINY RTC

12.4 Innerhalb der TM/TC-GUI veränderte und implementierte Dateien

- `lib/dispatcher_matcher_pattern/listener_interface.h`
  Zwei QT-Signale für die erbenden Klassen, insbesondere den `RemoteUnitMatcher`, wurden hinzugefügt.

- `lib/dock_widgets/universal_tc_packet_widget.cpp/h`
  Die VerificationTable wurde entfernt und in ein eigenes Widget eingebettet.
  Es wurde eine Liste und eine ComboBox mit den gefundenen IPs der Remote Units hinzugefügt. Ein Senden an eine spezifische IP ist nun möglich.

- `lib/gui/view_elements/tc_packet_header_layout.cpp`
  Standardmäßig werden alle Acknowledgements von den Remote Units angefordert.

- `lib/gui/view_elements/TmLoggerTable.cpp/h`
  Ein Telemetrielogger zeigt alle eigegangenen Telemetriepakete in einer Tabelle.

- `lib/gui/view_elements/verification_table_view_with_ip.cpp/h`
  In einer Tabelle werden eingehende Acknowledgements mit RU-IP angezeigt.

- `src/data/enums.h`
  APID, Service Types und Service Subtypes der Remote-Units-Services wurden mit Bezeichnung eingetragen.

- `src/gui/datafields/PwmSlider.cpp/h`
  Es wurde ein Bedienelement für PWM-gesteuerte Servomotoren implementiert.

- `src/gui/datafields/TextField.cpp/h`
  Es wurde ein Bedienelement für LCDs implementiert.

- `src/gui/dock_widgets/remote_unit_widget.cpp/h`
  Es wurde ein Widget implementiert, das die einer RU zugehörigen Sensoren und Aktoren anzeigt.

- `src/gui/dock_widgets/TmLoggerWidget.cpp/h`
  Es wurde ein Widget implementiert, das alle eingehenden Telemetriepakete in einer Tabelle anzeigt.

- `src/gui/dock_widgets/verificationWidget.cpp/h`
  Die Anzeige der Acknowledgements wurde in ein eigenes Widget eingebunden und um die Anzeige der IP-Adresse der sendenden Remote Unit ergänzt.

- `src/gui/window/window.cpp/h`
  Das Hauptfenster instanziiert die DockWidgets, legt den Main-Dispatcher an und verwaltet die Liste mit den IPs der gefundenen Remote Units.

- `src/gui/matchers/remote_unit_matcher.cpp/h`
  Der RemoteUnitMatcher verarbeitet die Telemetriepakete einer bestimmten Remote Unit. Er informiert das zugehörige RemoteUnitWidget mithilfe eines QT-Signals über Veränderungen.

- `src/gui/matchers/TmLoggerMatcher.cpp/h`
  Der TmLoggerMatcher empfängt Telemetriepakete aller Remote Units, speichert diese in einer Datei und aktualisiert die Tabelle des Telemetrielogger-Widgets.

- `src/gui/matchers/verificationMatcher.cpp/h`
  Der VerificationMatcher wurde angepasst, sodass auch die IP der sendenden Remote Unit in der Tabelle mit Acknowledgements angezeigt wird.

- `src/net/*`
  Dieser Ordner enthält den Verteilungsstack für eingehende Telemetriepakete (Receiver, ConnectionManager, ConnectionWrapper), den MainDispatcher, ein Interface für am MainDispatcher angeschlossene Matcher und einen UDP-Sender für Telekommandos.

- `tmtcGUI.pro / project.pro`
  Aus diesen Dateien generiert QMake ein Makefile für das Kompilieren. Neu hinzugefügte Klassen wurden hier eingetragen.

Literatur

[ada18] adafruit. *TSL2561 Luminosity Sensor*. 2018. URL: https://cdn-learn.adafruit.com/downloads/pdf/tsl2561.pdf (besucht am 18.11.2018).

[AG08] Arduino AG. *Arduino Nano (V2.3) User Manual*. 2008. URL: https://www.arduino.cc/en/uploads/Main/ArduinoNanoManual23.pdf (besucht am 22.10.2018).

[als16] alselectro. *Serial LCD I2C Module–PCF8574*. 2016. URL: https://alselectro.wordpress.com/2016/05/12/serial-lcd-i2c-module-pcf8574/ (besucht am 18.11.2018).

[BH14] A. Badach und E. Hoffmann. *Technik der IP-Netze: Internet-Kommunikation in Theorie und Einsatz*. Hanser, Carl, 2014. ISBN: 9783446439764.

[Dan] Frank Dannemann. *S2TEP. Small Satellite Technology Experiment Platform*. URL: https://www.dlr.de/irs/en/desktopdefault.aspx/tabid-12525/21846_read-49985/ (besucht am 02.10.2018).

[DEV14] ANALOG DEVICES. *Datasheet ADXL345*. 2014. URL: https://www.analog.com/media/en/technical-documentation/data-sheets/adxl345.pdf (besucht am 18.11.2018).

[Ele14] Elecrow. *Tiny RTC*. 2014. URL: https://www.openhacks.com/uploadsproductos/tiny_rtc_-_elecrow.pdf (besucht am 18.11.2018).

[Gmb16] Unbekant (ME-Meßsysteme GmbH). *Grundlagen zum CAN Bus*. 2016. URL: https://www.me-systeme.de/inhalte/grundlagen/canbus/kb-canbus.pdf (besucht am 22.09.2018).

[Gmb18a] Vector Informatik GmbH. *Einführung in CAN. Data Frame*. 2018. URL: https://elearning.vector.com/mod/page/view.php?id=47 (besucht am 15.11.2018).

[Gmb18b] Vector Informatik GmbH. *Einführung in CAN. Priorisierung*. 2018. URL: https://elearning.vector.com/mod/page/view.php?id=55 (besucht am 15.11.2018).

[Gre] Fabian; et al. Greif. *OUTPOST (github repository)*. URL: https://github.com/DLR-RY/outpost-core (besucht am 02.10.2018).

[Mic07] Microchip. *MCP2515 Datasheet*. 2007. URL: http://ww1.microchip.com/downloads/en/devicedoc/21801e.pdf (besucht am 11.10.2018).

[New08] element14 Newark Farnell. *Arduino Nano*. 2008. URL: http://www.farnell.com/datasheets/1682238.pdf (besucht am 22.10.2018).

[NXP07] NXP. *LM75A*. 2007. URL: https://www.nxp.com/docs/en/data-sheet/LM75A.pdf (besucht am 18.11.2018).

[NXP13] NXP. *PCF8574; PCF8574A*. 2013. URL: https://www.nxp.com/docs/en/data-sheet/PCF8574_PCF8574A.pdf (besucht am 18.11.2018).

[NXP15] NXP. *PCA9685*. 2015. URL: https://cdn-shop.adafruit.com/datasheets/PCA9685.pdf (besucht am 18.11.2018).

[Plu13] Plupp. *Canbus levels*. 2013. URL: https://commons.wikimedia.org/wiki/File:Canbus_levels.svg (besucht am 22.10.2018).

[rau] rnwissen. todo: autor raussuchen. *IIC, Inter-IC bzw. Inter Integrated Circuit Bus*. URL: http://rn-wissen.de/wiki/index.php?title=I2C.

[Sem03] Philips Semiconductors. *TJA 1050 Datasheet*. 2003. URL: https://www.nxp.com/docs/en/data-sheet/TJA1050.pdf (besucht am 11.10.2018).

[Sen18] Bosch Sensortex. *BMP280 Digital Pressure Luminosity Sensor*. 2018. URL: https://cdn-learn.adafruit.com/downloads/pdf/tsl2561.pdf (besucht am 18.11.2018).

[SM98] Inc. Siemens Microelectronics. *Controller Area Network*. 1998. URL: http://ecee.colorado.edu/~mcclurel/CANPRES.pdf (besucht am 15.11.2018).

[SS03] European Cooperation for Space Standardization. *Space engineering. Ground systems and operations - Telemetry and telecommand packet utilization*. 2003. URL: http://ecss.nl/get_attachment.php?file=standards/ecss-e/ECSS-E-70-41A30Jan2003.pdf (besucht am 17.11.2018).

[SS10] European Cooperation for Space Standardization. *Space engineering. Spacecraft on-board control procedures*. 2010. URL: http://ecss.nl/get_attachment.php?file=standards/ecss-e/ECSS-E-ST-70-01C16April2010.pdf (besucht am 15.11.2018).

[SS16] European Cooperation for Space Standardization. *Space engineering. Telemetry and telecommand packet utilization. (ECSS-E-ST-70-41C)*. 2016. URL: http://ecss.nl/get_attachment.php?file=2016/06/ECSS-E-ST-70-41C15April2016.pdf (besucht am 15. 11. 2018).

[Sys18] Espressif Systems. *ESP8266EX Datasheet*. 2018. URL: https://www.espressif.com/sites/default/files/documentation/0a-esp8266ex_datasheet_en.pdf (besucht am 02. 10. 2018).

[Tro13] Sebastian Trowitzsch. *BEESAT-2 Systemübersicht*. 2013. URL: https://www.raumfahrttechnik.tu-berlin.de/menue/forschung/aktuelle_projekte/beesat_2/raumsegment/ (besucht am 15. 11. 2018).

[TW12] A.S. Tanenbaum und D.J. Wetherall. *Computernetzwerke*. Always learning. Pearson, 2012. ISBN: 9783868941371. URL: https://books.google.de/books?id=uIyZMAEACAAJ.

[Unb14] Unbekannt. *SG90 9g Micro Servo*. 2014. URL: http://www.ee.ic.ac.uk/pcheung/teaching/DE1_EE/stores/sg90_datasheet.pdf (besucht am 18. 11. 2018).

[wul14] wulala. *D-Sub*. 2014. URL: https://cdn.pixabay.com/photo/2014/01/02/12/07/dsub-237547_960_720.png (besucht am 17. 11. 2018).

[Zie13] Werner Ziegelwanger. *Raspberry Pi: GPIO Schnittstelle - Teil 1*. 2013. URL: https://developer-blog.net/raspberry-pi-gpio-schnittstelle-teil-1/ (besucht am 17. 11. 2018).